Formal Specification of a Small Example Based on GKS

D. A. DUCE and E. V. C. FIELDING
Rutherford Appleton Laboratory
and
L. S. MARSHALL
University of Manchester

Implicit regeneration is a fundamental concept of the Graphical Kernel System (GKS), an ISO International Standard, but it is difficult to understand as presented in the English language specification of GKS. Consequently, it is a good example to use in determining whether formal specification techniques can be used effectively to describe and clarify graphics concepts of this kind. The problem is first motivated informally with a description of GKS concepts and terminology. A formal specification of implicit regeneration using a simplified model is then presented, and the notation that is used for its formalization (VDM) is described. Finally, properties of implicit regeneration are formulated and the specification is proved to conform to these properties. This demonstrates the applicability of formal specification to graphics software, because a sufficiently precise description of a complicated concept is provided that enables its consistency to be checked against an intuitive understanding of the concept as derived from the GKS document.

Categories and Subject Descriptors: D.2.1 [Software Engineering]: Requirements Specifications; F.3.1 [Logics and Meanings of Programs]: Specifying, Verifying and Reasoning about Programs; I.3.4 [Computer Graphics]: Graphics Utilities

General Terms: Design, Verification

Additional Key Words and Phrases: Abstract data type, bundled attributes, constructive specification, implicit regeneration

1. INTRODUCTION

There is much interest, currently, in the development of formal techniques for the design and specification of software. This paper explores the application of one such technique, the Vienna Development Method (VDM) [13], to the design of graphics software. The description of implicit regeneration in the Graphical Kernel System (GKS) [11, 12], an ISO International Standard for computer graphics programming, has been chosen as the basis for an example because the framework required for its description encompasses the major concepts on the output side of GKS. Implicit regeneration is a concept that readers of the GKS

Authors' addresses: D. A. Duce and E. V. C. Fielding, Rutherford Appleton Laboratory, Chilton, Didcot, Oxfordshire, England OX11 0QX; L. S. Marshall, Department of Computer Science, University of Manchester, Manchester, England M13 9PL.
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1988 ACM 0730-0301/88/0700-0180 $01.50
document have found difficult to understand. In this paper, an informal description is followed by a precise formal specification. From the specification, properties of implicit regeneration are formulated and proved, showing that the behavior conforms to that implied by the informal description. Thus the paper illustrates the power of the formal technique to give a concise description of a difficult concept and provide the ability to reason about a design.

2. THE PROBLEM

GKS provides a functional interface between an application program and a configuration of graphical input and output devices at a device-independent level of abstraction. The mechanism of implicit regeneration in GKS was chosen for formal specification because it is the key to understanding attribute binding in GKS, and in particular the power of bundled attributes, which, as others have argued, is one of the key innovations for providing device independence in GKS [1, 17]. The terms implicit regeneration and attribute binding, as well as related GKS terminology and concepts required for this description, are described informally below.

Various simplifications are made in order to focus on the key concepts of implicit regeneration in a small example. A more complete system can be described within a similar framework, as shown in [4], [6], and [7].

2.1 Pictures and Picture Structure in GKS

Pictures in GKS are constructed from six different basic building blocks, termed output primitives, which are abstractions of the basic actions that a graphical output device can perform (e.g., drawing a line). The GKS output primitives are polyline, polymarker, text, fill area, cell array, and generalized drawing primitive. Associated with each output primitive is a set of parameters, which defines a particular instance of the primitive. As there is a common model of attribute binding and implicit regeneration shared by all the primitives, only the polyline primitive is used here to illustrate the definition of this "generic" framework. The polyline primitive has as parameters the coordinates of a list of vertices defining its geometry and is a connected sequence of line segments between these vertices. This paper does not specify the geometry of the GKS output primitives; this topic is treated in [7].

GKS also provides an application with the capability of structuring a picture by optionally grouping output primitives into units called segments. Segments are stored and may be manipulated in certain ways as a single entity, but may not be nested. Thus pictures may consist of primitives and segments leading to the situation depicted in Figure 1.

With respect to implicit regeneration, there are cases in GKS where primitives inside segments are treated differently from those outside segments. The specification given below attempts to model these differences.

2.2 Workstations and Coordinate Systems in GKS

The physical output device on which the pictures are displayed is presented to the applications programmer through a workstation, which is a GKS abstraction of physical device hardware. An application program may direct output to more
than one workstation simultaneously, but the specification models only the case of a single workstation.

The application program specifies coordinate data in the parameters of an output primitive in *world coordinates* (WCs), a Cartesian coordinate system. The transformation pipeline in GKS has two stages. In the first, data defined in world coordinates are transformed to a common coordinate system, *normalized device coordinates* (NDCs), by a window to viewport mapping termed a *normalization transformation*. This transformation allows different parts of a picture defined in different world coordinate systems to be composed into a single NDC picture. Primitives in segments are stored in segment store after transformation to normalized device coordinates. The second stage, the *workstation transformation*, again a window to viewport mapping, allows different regions of the NDC picture to be viewed on different workstations. This transformation transforms normalized device coordinates to *device coordinates* (DCs). As transformations are not the primary concern of this paper, the specification ignores both world coordinates and clipping, and the range of NDC coordinates is not restricted.

2.3 Appearance of Primitives—Attributes

The appearance of output primitives in GKS is determined by their parameters and additional data, termed *aspects*. In the case of the polyline primitive these aspects are linetype (solid, dashed, dotted, dashed-dotted, or implementation dependent), linewidth scale factor, and polyline color index. The specification of Section 4 looks at only the two aspects of linetype and linewidth (the latter a simplification of linewidth scale factor) and assumes that the workstation supports any linetype or linewidth values requested.

Although there are two basic schemes for specifying aspects in GKS, termed *individual specification* and *bundled specification*, the former does not interact
with implicit regeneration and so only bundled specification is considered here. Individual specification and the relation between bundled and individual specification is addressed in [4].

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 of 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. If a polyline is to be displayed with a polyline index for which a representation has not been defined, the representation of polyline index 1 is used. A representation for polyline index 1 is always defined.

The point of interest here is that each workstation has its own polyline bundle table which allows the application to control the appearance of polylines created with the same polyline index independently on each workstation on which they are displayed, using the capabilities of the workstation.

GKS provides a function to set the value of polyline index modally:

```
SET POLYLINE INDEX(PLI)
```

as well as a function to set the representation of a bundle index on a particular workstation:

```
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: color index.

When a polyline is created, the current value of the polyline index is bound to the primitive (primitive attribute binding) and cannot subsequently be changed. Bundles are bound to primitives when they are displayed (workstation attribute binding).

The concepts of a transformation pipeline and attribute binding in GKS are depicted in Figure 2.

Now, consider the program fragment:

```
SET POLYLINE REPRESENTATION(1, 1, 3, 1.0, 4)
SET POLYLINE INDEX(1)
POLYLINE(N, X, Y)
```

The polyline defined by the N data points in the arrays X, Y is drawn with polyline index 1. On workstation 1, polyline index 1 corresponds to linetype 3, linewidth scale factor 1.0, and color index 4.

An interesting question is, “What happens if the representation of polyline index 1 on workstation 1 is subsequently changed?”

2.4 Implicit Regeneration

The GKS model allows a change in a representation to have a retrospective effect. 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.

GKS allows for the fact that some workstations may be able to perform such changes dynamically, while others may need to redraw the complete picture from a stored representation to effect the changes. It also allows the application to control when such redrawing (regeneration) takes place. As the only storage of primitives in GKS is the segment store, regenerations are performed using the contents of the segment store. 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.

For each class of picture change that can potentially require a regeneration, each workstation description table contains an entry to indicate whether the change is performed immediately (IMM) or requires a regeneration (IRG). When a change to the picture is requested, if the change can be made dynamically, it is made. If not, an implicit regeneration is signaled. GKS allows the application to control when the regeneration will occur through the setting of another flag, implicit regeneration mode. The possible values for this flag 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 or IMPLICIT REGENERATION MODE is set to ALLOWED.

It suffices to consider only changes in polyline representation in the specification that follows, since other changes such as changes of the workstation transformation, for example, can be modeled similarly.

3. FORMAL SPECIFICATION

The purpose of a specification is to state what a system is to do, not how it is to be done. It does this by describing the internal state of the system in an implementation-independent way by the use of abstract data types. These are data types characterized only by the operations allowed over them.

The two main approaches to specifying abstract data types are the algebraic approach, in which the operations of an abstract data type are specified by implicitly relating them to each other, using, for example, algebraic equations, and the constructive approach, in which the operations of an abstract data type are specified explicitly in terms of some precise discipline such as set theory.

Algebraic approaches are used in [2], [3], [9], [10], and [15], and a comparison of an algebraic approach with a constructive approach appears in [5]. This paper describes and uses a limited subset of the Vienna Development Method (VDM), with a somewhat simplified notation that is restricted to familiar mathematical syntax. VDM is an example of the constructive or model-based approach and models abstract data types in terms of mathematically tractable entities such as basic types (integers, denoted by \( \mathbb{N} \); reals, denoted by \( \mathbb{R} \)) and tuples, sets, lists, and mappings. The method and applications are described in detail in [8], [13], and [14].

The choice of VDM was motivated by its widespread use in Europe and because it is particularly well suited for directly formulating and proving statements about the behavior of a specification because of the way operations are specified.

A VDM specification has three components:

1. a model of the state,
2. invariants on the state,
3. operations over the state.
The state definition describes the structure of the class of objects representing the state in terms of basic and constructed types. The operations are defined implicitly by predicates, which allow relations and thus nondeterminacy to be specified. The operations given in this paper, however, do not require this generality and, in fact, reduce to functions. Operation definitions given here have the general type

\[ \text{State} \times \text{Inputs} \rightarrow \text{State} \]

and are described by two predicates: a \textit{precondition} and a \textit{postcondition}. The former is a predicate over \textit{State} and \textit{Inputs} and defines the conditions under which the operation produces a valid result. The latter is a predicate over \textit{State} (the initial state), \textit{Inputs}, and \textit{State} (the final state), which defines the effect of the operation.

4. THE COMPLETE SPECIFICATION
The various concepts comprising the simplified GKS system described above are captured in a VDM specification by a combination of the choice of data type to represent the state, the invariants over the state, and the definitions of the operations over this state.

The specification is particularly focused on the following concepts:

1. an intermediate picture in a common coordinate system (NDC space),
2. pictures structured by grouping primitives into segments,
3. segments in NDC coordinates stored in a segment store,
4. primitive creation and binding of attributes at the NDC level,
5. binding of representations to primitives at the DC level,
6. the behavior of implicit regeneration.

Because GKS does not specify the order in which primitives are to be displayed, the data type multiset is used to represent pictures in the GKS specification below. This data type is denoted by \texttt{mset} and is like the set data type, with the exception that duplicates are allowed in \texttt{mset} whereas they are removed in sets.

4.1 The State
The state of a formal specification of the example system described above must capture the concepts of NDC space, the segment store, and a workstation.

These concepts are captured in the state by having one component that models the NDC picture, a second component that models the segment store, and four components that model the concept of a workstation. The components modeling a workstation are an abstract model of the DC picture that is displayed on the display surface, the polyline bundle table, the bundle modification flag, and the implicit regeneration mode, the last two of which control the effects of polyline bundle representation change and when regeneration is to occur.

This leads to the definition for the state of the system shown in Figure 3.

4.1.1 Description. States of the system are described by objects of type \textit{GKS}, which is defined to be a tuple with first component of type \textit{NDC\_Picture}, second
component of type `DC_Picture`, third component of type `Segment_Store`, fourth component of type `Polyline_Bundle_Table`, and fifth and sixth components two flags of types `Bundle Modification_Flag` and `Implicit_Regeneration_Mode`.

The `NDC_Picture` is modeled as a set of objects of type `Component`, where `Component` may be either of type `NDC_Polyline`, or of type `Segment` ("|" denotes alternation). This allows pictures in NDC space to be constructed from both segments and primitives outside segments. The type `NDC_Polyline` is an ordered pair consisting of a list of real coordinate pairs (the coordinates of its vertices) and a `Polyline_Index`, which will be bound to the polyline at the time of its creation. Segments are modeled, as might be expected, by sets of primitives, and segment store is modeled as a set of segments.

The DC picture that is displayed on the workstation is modeled similarly to the NDC picture, as a set of DC polylines. At the DC level, a polyline is also described as a list of real coordinate pairs, a `Bundle`, which enables the concept of a bundle being bound to the primitive at display time to be captured, and a `Polyline_Index`.

The data type used to model the polyline bundle table is a mapping from a polyline index to a bundle (a linetype, linewidth pair). A mapping is a finite function in which the pairing of domain and range elements is constructed explicitly.
The type `BundleModification_Flag` can take the values `IRG` and `IMM`. The possible values of the type `ImplicitRegeneration_Mode` are similarly enumerated.

The DC picture description has been designed so that it is an abstraction of what is displayed on the display surface of a device. It records both the geometric and aspect data from which a visible image might be rendered and so describes the appearance of the picture to be displayed. It also contains the information required to capture the concept of a workstation dynamically updating the appearance of the displayed picture without reference to the segment store.

It is possible to compare the equality of pictures at the DC level with this data type. Two DC pictures are said to be equivalent if they contain exactly the same number of DC polylines and if corresponding DC polylines have equal lists of points, linetypes, and linewidths. Two DC pictures are said to be equal if they are equivalent and if corresponding DC polylines have equal polyline indices.

Note that this design is not concerned with modeling 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), whereas others would have no stored representation whatsoever, other than the picture on the physical display surface (e.g., a pen plotter).

4.2 Invariants

The invariants are constraints on the values that may be assumed by the components of the state and must be preserved by operations over the state.

The precise details of the transformation from NDC to DC coordinates are not important here. One fixed transformation function is used throughout this paper and its type is as follows:

\[ t : NDC\_Points \rightarrow DC\_Points \]

In defining the invariants, the names of both the state and its components are required; so the following clause provides these and holds over the definitions in this section.

\[
\text{let } mk\_GKS(ndep, dep, ss, pbt, bmf, ir) = gks \text{ in }
\]

The function `mk_GKS` is an example of a constructor function and constructs an object of type `GKS` from the named components. In VDM, such functions are implicitly assumed to be available for use in constructing and decomposing constructed data types. The name of a constructor for a type is the type name prefixed by `mk_`. We follow the convention that type names have capitalized initial letters and the names of instances are in lowercase.

**Invariant (1).** All segments in the NDC picture must be stored in the segment store, and vice versa:

\[
ss = \{ s \mid s \in ndep \land \text{is\_type\_Segment}(s) \}
\]

(1)

Invariant (2). Polylines that are in the NDC picture are also in the DC picture:

\[
\text{dcp} = \{ \text{mk}_\text{DC-Polyline}(t(\text{pts}), b, \text{pi}) \mid \text{mk}_\text{NDC-Polyline}(\text{pts}, \text{pi}) \in \text{nscp} \\
\lor (\text{mk}_\text{NDC-Polyline}(\text{pts}, \text{pi}) \in s \land \text{is-type-Segment}(s) \\
\land s \in \text{nscp}) \text{ for some } b \in \text{Bundle} \}
\]

(2)

4.3 Operations

The operations over this state are \text{add-polylines}, which captures the GKS polyline function for primitives outside segments; \text{add-segment}, which captures the GKS functions concerned with collecting primitives into segments; \text{redraw-all-segments}, which represents all the GKS functions that cause the picture displayed on the workstation to be updated; and \text{set-polyline-representation}, which associates a bundle representation with a polyline index in the polyline bundle table and captures the effects of changing a polyline bundle representation on polylines already displayed on the workstation.

Both the \text{add-polylines} and \text{add-segment} operations are abstractions of an amalgamation of GKS functions. In GKS, the polyline index is set modally, but in the specification given here, the appropriate polyline indices are provided as arguments to the \text{add-polylines} and \text{add-segment} operations (in the latter case bound to the polylines in the list of NDC polylines given as argument). Thus \text{add-polylines} is equivalent to the sequence of GKS functions (when no segment is open):

\text{SET POLYLINE INDEX}(\ldots)
\text{POLYLINE}(\ldots)

and \text{add-segment} is equivalent to the sequence of GKS functions:

\text{CREATE SEGMENT}(\ldots)
\text{SET POLYLINE INDEX}(\ldots)
\text{POLYLINE}(\ldots)
\ldots
\text{SET POLYLINE INDEX}(\ldots)
\text{POLYLINE}(\ldots)
\ldots
\text{CLOSE SEGMENT}

The GKS function \text{ASSOCIATE SEGMENT WITH WORKSTATION} is also a partial analogy for \text{add-segment}.

Initialization of the state is not described, as this is straightforward. Full specifications of the operations are shown in Figure 4.

4.3.1 Description. The \text{let} clause preceding the operation definitions applies to all subsequent operation definitions. The first line serves to name the initial state and its components. The names of the final state and its components that result from an operation are obtained by decorating the names given in this line of the \text{let} clause with a prime ('). This is done in the second line of the \text{let} clause.
let mk\_GKS(ndcp, dcp, ss, pbt, bmf, ir) = gks and
    mk\_GKS(ndcp', dcp', ss', pbt', bmf', ir') = gks' in

add\_polyline: GKS × NDC\_Points × Poliline\_Index → GKS
add\_polyline(gks, pts, pi, gks') \triangleq
post ndcp' = \{mk\_NDC\_Polyline(pts, pi) \cup ndcp ∧
    dcp' = \{mk\_DC\_Polyline((pts), bundle(pi, pbt), pi) \cup dcp
bundle: Poliline\_Index × Poliline\_Bundle\_Table → Bundle
bundle(pi, pbt) \triangleq if pi ∈ dom pbt then pbt(pi) else pbt(1)

add\_segment: GKS × Segment → GKS
add\_segment(gks, s, gks') \triangleq
post ndcp' = \{s\} \cup ndcp ∧
    ss' = \{s\} \cup ss ∧
    dcp' = to\_dc(s, pbt) \cup dcp

in\_seg: Segment × Poliline\_Bundle\_Table → DC\_Picture
in\_seg(s, pbt) \triangleq \{mk\_DC\_Polyline((pts), bundle(pi, pbt), pi) \cup mk\_NDC\_Polyline(pts, pi) ∈ s\}

redraw\_all\_segments: GKS → GKS
redraw\_all\_segments(gks, gks') \triangleq
post dcp' = regenerate(ss, pbt) ∧
    ndcp' = in\_segment(ndcp)

recreate: DC\_Picture × Poliline\_Bundle\_Table → DC\_Picture
recreate(dcp, pbt) \triangleq \{mk\_DC\_Polyline(dcp, bundle(pbt), pi) \cup
    mk\_DC\_Polyline(dcp, b, pi) ∈ dcp for some b ∈ Bundle\}

in\_seg: NDC\_Picture → NDC\_Picture
in\_seg(ndcp) \triangleq \{\{s\} \in ndcp ∧ is\_type\_Segment(s)\}

set\_polyline\_representation: GKS × Poliline\_Index × Linetype × Linewidth → GKS
set\_polyline\_representation(gks, pi, lt, lw, gks') \triangleq
post pbt' = pbt + \{pi → mk\_Bundle(lt, lw)\} ∧
    (bmf = IMM ⇒ ndcp' = ndcp ∧
    dcp' = recreate(dcp, pbt')) ∧
    (bmf = IRG ∧ ir = ALLOWED ⇒
    ndcp' = in\_segment(ndcp) ∧
    dcp' = regenerate(ss, pbt') ∧
    (bmf = IRG ∧ ir = SUPPRESSED ⇒ ndcp' = ndcp ∧
    dcp' = dcp)

Fig. 4. The operations.

The first line of an operation definition is its signature, which defines the types of the arguments and the result of the operation. The second line (following the \(\triangleq\)) simply names the objects of the types given in the signature. Thus in `add\_polyline`, `gks`, the initial state, is of type `GKS`, the argument `pts` is of type `NDC\_Points` (a list of NDC coordinate pairs), the argument `pi` is of type `Polyline\_Index`, and the result of the operation, `gks'`, the final state, is also of type `GKS`.

The preconditions of `add\_polyline` is `true` and so has been omitted. This is the case with the preconditions of all the operations. The postcondition describes

the effect of the operation. In `add polyline` the postcondition states that as a polyline is created, an object of type `NDC_Polyline` is formed (by `mk_NDC_Polyline(pts, pi)`) from the NDC list and polyline index supplied as arguments and is added to the set `ndcp`, representing the NDC picture. The construction of an NDC polyline captures the concept of the polyline index being bound to the polyline at the time of its creation. A corresponding polyline is displayed in DC space by forming an object of type `DC_Polyline` and adding it similarly to the DC picture. This DC polyline is formed from the transformed NDC coordinates (obtained by applying the function `t` to the NDC coordinates) and the bundle obtained by the auxiliary function `bundle` and the polyline index. The function `bundle` either applies the polyline bundle table mapping to the polyline index given as argument to yield the associated bundle (denoted by `pbt(pi)`) or returns the default bundle associated with polyline index 1 if that polyline index is undefined (i.e., if `pi` is not in the domain of the polyline bundle table). Components of the state not modified by an operation are by convention omitted from the postcondition (in this case `ss`, `pbt`, `ir`, and `imm` are not modified and are hence omitted).

The postcondition of `add segment` states that the segment `s` is simply added to the sets representing the NDC picture and the segment store. It uses the function `to dc` to generate the DC picture corresponding to the segment given as argument. The function `to dc` constructs a set of DC polylines corresponding to the set of NDC polylines comprising the segment. The coordinates of each NDC polyline are transformed to DC coordinates, and the representation of each polyline index obtained from the polyline bundle table is bound to the DC polyline. A flattening and loss of the segment structure occurs in adding the DC equivalent of the NDC segment to the DC picture as the set of DC polylines resulting from the application of the `to dc` function is added by the union operator `∪` to the previously displayed picture.

The `redraw all segments` operation constructs a new DC picture from the segment store that has the effect of deleting primitives outside segments from the NDC picture. The function `regenerate` traverses the set of sets representing the segment store. It uses the function `to dc` to generate from the NDC polylines in each segment the corresponding DC polylines, with both polyline indices and bundles bound, and constructs a DC picture without segment structure. The function `in segment` delivers the segments in the NDC picture and is used to describe the effect of removing primitives outside segments (i.e., the retention of only primitives stored in segments).

The `set polyline representation` operation describes the addition to the polyline bundle table of the new representation specified for the polyline index. The operator `+` adds `pi → mk_Bundle(lt, lw)` to the mapping representing the polyline bundle table, overriding any previous value associated with `pi`. The function `recreate` describes the effect of an immediate change to a DC picture by effectively rebinding the bundles to the DC polylines in the DC picture. It uses the polyline index value contained in each DC polyline to look up the new bundle representation in the polyline bundle table mapping. This is done without reference to the segment store.

Note that the contents of the DC picture are not always derivable from the contents of the NDC picture and the current polyline bundle table and that their
relationship cannot be expressed as an invariant. If the dynamic modification is
IRG and the implicit regeneration mode is SUPPRESSED, changing represen-
tations only applies to subsequently created primitives. Previously created primit-
ives will continue to be displayed with representations that might not be in the
current polyline bundle table. However, all primitives appearing in the NDC
picture also appear in the DC picture (invariant (2)).

The segment store is directly derivable from the NDC picture and the two are
related by invariant (1). VDM does not assume that invariants hold by definition.
In VDM, the postconditions of operations explicitly construct both the NDC
picture and the segment store and any invariants should be proved to be true.
Other constructive techniques, such as Z [16], assume that invariants hold by
definition and would not construct both the NDC picture and the segment store,
but would define the segment store in terms of the NDC picture by means of an
invariant. There are arguments for both views: the latter view is more concise,
but also more indirect.

5. BEHAVIOR

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

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

Suppose there is a third system in which the workstation requires a picture
regeneration. If regeneration mode for the workstation of the third system 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 can be shown that the specification does indeed conform to this intuitive
behavior. Three GKS systems with appropriate settings of the dynamic modifi-
cation and implicit regeneration mode flags are considered, and the intuitive
behavior described above is stated formally in a theorem below.

THEOREM. Consider systems gks0, gks1, and gks2, in which

\begin{align*}
\text{let} & \quad \text{mk\_GKS}(\text{nncp}, \text{ncp}, \text{ss}, \text{pbt}, \text{bmf}_0, \text{ir}_0) = \text{gks}_0 \quad \text{and} \\
\text{mk\_GKS}(\text{nncp}, \text{ncp}, \text{ss}, \text{pbt}, \text{bmf}_1, \text{ir}_1) = \text{gks}_1 \quad \text{and} \\
\text{mk\_GKS}(\text{nncp}, \text{ncp}, \text{ss}, \text{pbt}, \text{bmf}_2, \text{ir}_2) = \text{gks}_2 \quad \text{and} \\
\text{bmf}_0 = \text{IMM} \quad \text{and} \\
\text{bmf}_1 = \text{IRG} \quad \text{and} \quad \text{ir}_1 = \text{ALLOWED} \quad \text{and} \\
\text{bmf}_2 = \text{IRG} \quad \text{and} \quad \text{ir}_2 = \text{SUPPRESSED}
\end{align*}

that is, their NDC picture, DC picture, segment store, and polyline bundle table components are identical; then, if $gks_0$, $gks'_0$ and $gks_1$, $gks'_1$ are related by

\[
\text{set polyline representation}(gks_0, pi, lt, lw) = gks'_0,
\]
\[
\text{set polyline representation}(gks_1, pi, lt, lw) = gks'_1,
\]

and

\[
\text{redraw all segments}(\text{set polyline representation}(gks_2, pi, lt, lw)) = gks''_0,
\]

then

\[
dcp'_0 = dcps'_1 = dcps''_1 \quad \text{iff} \quad ndcp = ss = \{s \mid s \in ss\}.
\]

That is, the resulting DC pictures in the three systems are equal if and only if the NDC picture is composed entirely of segments. The value of $ir_0$ is immaterial.

**PROOF.** The proof given here follows directly from the definitions of the operations and the invariants. Proving that the operations maintain the invariants is straightforward, and the details are not given here.

Consider first systems $gks_0$ and $gks_1$. It has to be proved that

\[
\text{ndcp} = ss \Rightarrow dcps'_0 = dcps'_1,
\]
\[
\text{ndcp} \neq ss \Rightarrow dcps'_0 \neq dcps'_1.
\]

From the postcondition of \text{set polyline representation},

\[
pbt'_0 = pbt + [i \mapsto mk\_Bundle(lt, lw)],
\]
\[
pbt'_1 = pbt + [i \mapsto mk\_Bundle(lt, lw)].
\]

Thus $pbt'_0 = pbt'_1$. In the remainder of the proof $pbt'$ is used to denote this value.

Also from the postcondition of \text{set polyline representation},

\[
dcp'_0 = \text{recreate}(dcp, pbt'),
\]
\[
dcp'_1 = \text{regenerate}(ss, pbt').
\]

Substituting these into eqs. (3) and (4), we then need to prove that

\[
\text{ndcp} = ss \Rightarrow \text{recreate}(dcp, pbt') = \text{regenerate}(ss, pbt'),
\]
\[
\text{ndcp} \neq ss \Rightarrow \text{recreate}(dcp, pbt') \neq \text{regenerate}(ss, pbt').
\]

From invariant (2) the NDC and DC pictures are related by

\[
dcp = [mk\_DC\_Polyline(t(pts), b, pi) \mid mk\_NDC\_Polyline(pts, pi) \in \text{ndcp}
\]
\[
\lor (mk\_NDC\_Polyline(pts, pi) \in s \land is\_type\_Segment(s) \land s \in \text{ndcp})
\]
\[
\text{for some } b \in \text{Bundle}.
\]
Consider applying the recreate function to (7):

\[
\text{recreate } \left( \{ \text{mk DC Polyline}(t(pts), b, pi) \mid \text{mk NDC Polyline}(pts, pi) \in \text{ndcp} \right) \\
\quad \lor \left( \{ \text{mk NDC Polyline}(pts, pi) \in s \land \text{is type Segment}(s) \land s \in \text{ndcp} \right) \\
\quad \text{for some } b \in \text{Bundle}, pbt’ \left) \\
\quad = \{ \text{mk DC Polyline}(t(pts), bundle(pi, pbt’), pi) \\
\quad \mid \text{mk DC Polyline}(t(pts), b, pi) \\
\quad \in \{ \text{mk DC Polyline}(t(pts), b, pi) \\
\quad \mid \text{mk NDC Polyline}(pts, pi) \in \text{ndcp} \\
\quad \lor \{ \text{mk NDC Polyline}(pts, pi) \in s \\
\quad \land \text{is type Segment}(s) \land s \in \text{ndcp} \right) \\
\quad \text{for some } b \in \text{Bundle}, pbt’ \} \\
\quad = \{ \text{mk DC Polyline}(t(pts), bundle(pi, pbt’), pi) \\
\quad \mid \text{mk NDC Polyline}(pts, pi) \in \text{ndcp} \\
\quad \lor \{ \text{mk NDC Polyline}(pts, pi) \in s \\
\quad \land \text{is type Segment}(s) \land s \in \text{ndcp} \right) \\
\quad \text{for some } b \in \text{Bundle}, pbt’ \} \\
\quad \cup \{ \text{mk DC Polyline}(t(pts), bundle(pi, pbt’), pi) \\
\quad \mid \text{mk NDC Polyline}(pts, pi) \in \text{ndcp} \\
\quad \land \text{is type Segment}(s) \land s \in \text{ndcp} \right) \\
\quad = \text{regenerate}(ss, pbt’). \tag{8}
\]

Hence (8) can be written

\[
= \{ \text{mk DC Polyline}(t(pts), bundle(pi, pbt’), pi) \\
\quad \mid \text{mk NDC Polyline}(pts, pi) \in \text{ndcp} \} \\
\quad \cup \text{regenerate}(ss, pbt’). \tag{9}
\]

As we are using multisets, this expression is equal to

\[
\text{regenerate}(ss, pbt’)
\]

iff

\[
\{ \text{mk DC Polyline}(t(pts), bundle(pi, pbt’), pi) \\
\quad \mid \text{mk NDC Polyline}(pts, pi) \in \text{ndcp} \} = \emptyset,
\]

which is true iff

\[
\{ \text{mk NDC Polyline}(pts, pi) \mid \text{mk NDC Polyline}(pts, pi) \in \text{ndcp} \} = \emptyset.
\]
If $n_{dep} = ss$, then by invariant (1)

$$\{ mk\_NDC\_Polyline(pts, pi) \mid mk\_NDC\_Polyline(pts, pi) \in n_{dep} \} = \emptyset,$$

proving (5) and thus (3).

If $n_{dep} \neq ss$, then

$$\{ mk\_NDC\_Polyline(pts, pi) \mid mk\_NDC\_Polyline(pts, pi) \in n_{dep} \} \neq \emptyset,$$

proving (6) and thus (4). Thus we have shown that

$$d_{cp}' = d_{cp}, \quad \text{iff} \quad n_{dep} = ss.$$

We now need to show that

$$d_{cp}'' = d_{cp}, \quad \text{iff} \quad n_{dep} = ss.$$

From the postcondition of set\_polyline\_representation,

$$d_{cp}'' = d_{cp} \land p_{bt}'' = p_{bt} + \{ i \mapsto mk\_Bundle(lt, lw) \} = p_{bt}'' \land ss'' = ss.$$

From the postcondition of redraw\_all\_segments,

$$d_{cp}'' = \text{regenerate}(ss'', p_{bt}'') = \text{regenerate}(ss, p_{bt}) = d_{cp}'.$$

By the first part of the proof,

$$d_{cp}' = d_{cp}, \quad \text{iff} \quad n_{dep} = ss,$$

and hence

$$d_{cp}'' = d_{cp}, \quad \text{iff} \quad n_{dep} = ss. \quad \square$$

6. CONCLUSIONS

The example given here shows how a specification of implicit regeneration in GKS could be given. The specification here clearly states what effects are to be achieved by an implementation. For example, if any polyline bundle table representation is changed, and dynamic modification for this is IRG, the effect to be achieved is as if a regeneration were performed, even if no primitives in the picture have been created with the corresponding polyline index, or the new representation value is exactly the same as the old value. It is important to note that the specification defines the result of the operation, not the mechanism by which it is to be achieved in an implementation. The specification does not mandate that an implementation perform a regeneration from some physical storage and certainly does not preclude the implementation from detecting circumstances in which a regeneration is unnecessary in the sense that the specified effect can be achieved without one.

Formal specification is no recipe for avoiding errors in design, nor does it provide a recipe for producing good designs. What it does provide is a precise
medium that can be used by a designer for communicating a design to others unambiguously. Because it is formally based, it is possible to check whether the design corresponds to the behavior expected by formulating and proving theorems. Further advantages over a natural language description are that it is more concise and that it is also possible for questions of the form “What happens if . . . ?” to be formulated precisely and answered.

We believe that the specification that has been presented above captures the essential concepts that are the key to producing a complete formal specification of GKS. As a result we feel that problems that will arise in attempting to specify the whole of GKS formally are likely to be concerned with keeping the specification readable rather than with difficulties in being able to formalize concepts. The success of this application of a formal method to the description of a difficult, but fundamental, concept of GKS will, it is hoped, encourage the use of formal techniques in the description of emerging graphics standards and also in the further development and improvement of these techniques themselves.

ACKNOWLEDGMENTS

We wish to thank Bob Hopgood, Dale Sutcliffe and Julian Gallop for their many useful comments on an earlier draft of this paper, Alan Kinroy for his assistance with proofreading, and the referees for their constructive criticisms which have significantly improved this paper.

REFERENCES


Received November 1984; revised May 1986; accepted February 1988