GKS Inquiry Functions within PROLOG

P. Sykes and R. Krishnamurti

1 Introduction

GKS, the international standard for 2D graphics software, provides a set of functionalities which are specified in a language independent manner. However, for GKS to be used, a binding must be defined for some host programming language. To date a FORTRAN binding [1] has been accepted, and proposals for bindings in Pascal [8] and Ada are under consideration in ISO. Possible bindings for C [7] and ALGOL 68 [5] have also been proposed.

A language binding for the PROLOG programming language is currently under development [9], and a draft version is being implemented within an enhanced version of the C-PROLOG interpreter [6] running on UNIX. GKS is designed to be implemented in the natural programming language of the host system and to have a language dependent layer as an interface to each of the other programming languages on the system. Our PROLOG implementation, therefore, forms an interface to a library of GKS functions written in the C programming language.

Different PROLOG implementations may have different syntactical rules for differentiating between variables and atoms. The convention adopted in this binding document is that adopted by the C-PROLOG interpreter. Variables start with an upper case letter or an underscore, atoms start with a lower case letter or may be any string enclosed within single quotes.

The draft ANSI standard GKS document [1, 3] specifies guidelines for language bindings. These essentially constrain a binding, in effect, to provide a one-to-one mapping of GKS abstract functions to atomic language functions, and to specify data types corresponding to the GKS abstract data types. The rules also require the binding to observe good software engineering principles, a requirement which we

P. R. Bono et al. (eds.), *GKS Theory and Practice*

have taken to mean that the functions names be mnemonic and that the parameter lists be kept to manageable proportions. Within a PROLOG environment it is possible for the programmer to define multiple predicates having the same name but with differing argument lists. That is, the arguments may differ in length and/or type. This feature has been utilized to provide the binding with a degree of flexibility. This is illustrated in this paper in one area, namely that of the GKS inquiry functions.

In their paper on a C binding, Rosenthal and ten Hagen [7] added two more rules to the list, namely,

(a) The GKS specification should not be interpreted literally as to prevent the application programmer making use of the full range of the host language's facilities.

This rule is particularly relevant to a PROLOG graphics binding. Prolog is a declarative language. Any binding in PROLOG must either be declarative or must at least look declarative. A PROLOG binding that forces procedural programming techniques on the applications programmer will not find widespread acceptance by the PROLOG community.

(b) The GKS document should not be interpreted literally as to force inefficient techniques on the implementor.

PROLOG provides a flexible environment for programming that may be attributed to many factors among which are the following. First, arguments to a PROLOG predicate are not strongly data typed. That is, for example, some clauses of a PROLOG predicate may have arguments that are instanced to simple constants or atoms whilst others may have the same arguments instanced to compound terms or structures. Second, arguments to a predicate do not have fixed scope in that they may, in general, serve as either input or output. Third, PROLOG permits definitions for predicates with the same name that differ in parameter lengths. In other words, a predicate is uniquely specified only by both its name and its parameter length (arity). A consequence of this is that GKS functions may be used in ways which were not foreseen in the original standard specifications.

Most PROLOG implementations whether they are compilers or interpreters are written partially in some host language and partially in PROLOG. The simplest way to implement a PROLOG binding is to write it essentially in PROLOG rather than in the host language of the Prolog compiler/interpreter. This ensures that the binding is specified in a manner that makes it more natural to use within a PROLOG environment.

2 The PROLOG Binding

The PROLOG under development at EdCAAD provides for this mapping of GKS functions to PROLOG predicates. We have adopted a naming convention for the predicates that does not err on the side of being too terse, yet is still reasonably compact. All the GKS predicates have the prefix **gk_** and all GKS inquiry predicates have the prefix **gk_q_**. Wherever possible and without ambiguity as to the intended functionality, the predicate names are abbreviated. The naming convention is given

in the document describing the suggested binding [9].

The data types in the standard are merely tools for describing the semantics of the standard. They should be replaced by actual data types conforming to the host language. PROLOG has no context independent notion of data typing. Also PROLOG has no context independent semantics for operators, though the operator syntax must be strictly obeyed in a PROLOG term. For example, < is a PROLOG infix operator. Any expression involving < must be of the form LHS < RHS where LHS and RHS are valid PROLOG terms. However, PROLOG will not interpret this expression as the conditional

LHS "less than" RHS

unless it is stated as a PROLOG goal. Moreover, PROLOG permits overloading of operator type. Thus, for example, the operator + is both prefix and infix. There is no reason why it can't be declared postfix as well.

We have found it convenient to employ some PROLOG operators, for instance, *X:Y* to describe coordinate pairs, and *Attribute = Value* to name parameters in lengthy argument lists. In many cases we have parameters that are structures; for example, the polyline representation is denoted by the PROLOG functor *line(Id, Type, Width, Colour)*. In fact, the structured parameters may themselves have arguments which need not be atomic, for example, data record items.

Lastly, it should be noted that PROLOG clauses are logical implications that either *succeed* (when **true**) or *fail* (when **false**) and take one of the two following forms:

```
/* 1 */
        Goal.

/* 2 */
        Goal :- condition1
                condition2
                .
                .
                .
                conditionn
```

In the first case, 'Goal' is treated as a fact which succeeds whenever its arguments, if any, are matched. In the second case, 'Goal' succeeds only if each condition 1 through n succeeds and fails otherwise. Each condition, in turn, is a PROLOG goal.

3 Inquiry Functions

The GKS inquiry functions return information about the current state of GKS. In a conventional von Neumann language the value of the return parameter would be tested and the program would continue as required. The PROLOG equivalent of this may be described as

```
gk_q_function (Var),    /★ get value of Var ★/
test (Var, value),      /★ succeeds if Var is value ★/
.
.
etc                     /★ carry on only if test succeeds ★/
```

A more natural implementation would require the inquiry and the test to work in one go. Thus, we have

```
gk_q_function (value),    /★ succeeds if the inquired function matches value ★/
```

This is the preferred form where the returned value is usually one in a set of enumeration types. A typical PROLOG application would then have several clauses of the form:

```
inquire_and_do :-
                    gk_q_function (value 1),
                    !,
                    do_action_1.

inquire_and_do :-
                    gk_q_function (value 2),
                    !,
                    do_action_2.
.
.

inquire_and_do :-
                    gk_q_function (value n),
                    !,
                    do_action_n.
```

The effect of this is that PROLOG would first determine if the result of the inquiry was $value_1$ and if so then it would 'do_action_1'. The goal can fail in two ways. Either if the inquiry failed in which case the next 'inquire_and_do' clause is tried, or the 'do_action' clause fails in which case the goal fails. The cut (!) operator acts as a *barrier* to prevent PROLOG from backtracking and trying other 'inquire_and_do' clauses in the event of a successful inquiry. PROLOG would repeat this process with each of the 'inquire_and_do' clauses in the given order until either one succeeds or the entire goal fails.

Many GKS inquiry functions return several values. In a conventional von Neumann language each of these parameters is specified by its position in the parameter list which has a predetermined length. In PROLOG it is possible to allow the programmer to state which of the parameters he is interested in and in a similar manner to the example above to state what he expects it to be. Furthermore the parameter may be specified by name and not by position.

The general format of an inquiry function is

gk_q_GKS_FUNCTION(*Attribute* = *Value*)

or

gk_q_GKS_FUNCTION([List of *Attribute* = *Value* terms])

The list may contain as many different attributes as required.

4 Examples

Consider the function INQUIRE WORKSTATION NUMBERS. This returns three small integers corresponding to the maximum number of workstations that are simultaneously open, active or have associated segments. The FORTRAN binding implements this as

SUBROUTINE GQWKM (ERR, MXOPWK, MXACWK, MXASWK)

which returns in the four arguments an error indicator and the three maximums. (In the PROLOG binding an error situation corresponds to a failure of the inquiry goal.)

The obvious equivalent PROLOG predicate is:

gk_q_ws_max([open = *Mxop*, active = *Mxac*, assoc = *Mxas*]),

However, if all the parameters are not required the PROLOG binding allows the applications programmer to use this inquiry function in the following ways:

gk_q_ws_max(open = *Mxop*),

to ask the maximum number of open workstations;

gk_q_ws_max(open = 3),

will succeed only if the maximum number of open workstations is 3;

gk_q_ws_max(*M* = 3),

will instantiate *M*, in turn via 'backtracking', to each one of open, active or assoc provided the corresponding maximum number of workstations equals 3, and fails otherwise;

gk_q_ws_max([open = 3, active = *Mxac*]),

succeeds only if the maximum number of open workstations is 3 and instantiates *Mxac* to the maximum number of active workstations.

The last case is:

gk_q_ws_max(*L*),

instantiates *L* to a list with the three members of the form *Attribute* = *Value*.

Combinations of the above cases are also permitted. For instance,

gk_q_ws_max([open = 3 | *L*]),

will succeed if the maximum number of open workstations is 3 and instantiates *L* to

a list of the other parameters. The order of the arguments is not important. Thus, the inquiry

gk_q_ws_max([open = *Mxop*, active = *Mxac*])

is the same as

gk_q_ws_max([active = *Mxac*, open = *Mxop*])

The above mechanism works equally well with enumerated data types. For instance, consider the GKS functionality INQUIRE SEGMENT ATTRIBUTES which returns for a given segment name, its transformation matrix, relative priority, and three enumerated types which correspond to the visibility, highlighting and detectability of the segment. The PROLOG implementation allows the programmer to form the query in such a way that his code is not cluttered with unwanted variables. For instance, the goal

gk_q_seg(*SEG*, [norm = *MAT*, detect = yes]),

will return the transformation matrix in *MAT* for segment *SEG* only if it is detectable.

Some of the GKS inquiry functions return so many arguments that even this implementation would be unwieldy. For instance, the functionalities INQUIRE CURRENT PRIMITIVE ATTRIBUTE VALUES and INQUIRE CURRENT INDIVIDUAL ATTRIBUTE VALUES have 11 and 13 arguments respectively. The arguments relate to the various graphics primitives - for example, polyline, text etc - and their attributes.

It is possible to implement these as single predicates each with a list of ten or more arguments, some of which are points, some integers, some names, some enum types and some lists; or as in the FORTRAN binding and indeed as suggested in other bindings, to implement these as separate inquiry functions, one for each attribute. The method chosen for the PROLOG binding is to allow the application to specify which attribute of some primitive is required. As in the previously considered inquiry function, it is desirable to have the goal succeed if the return value matches what is expected. Therefore in this example, the PROLOG goal

gk_q_line(index = *LI*)

will succeed with the variable *LI* instantiated to the current polyline index. Similar goals can be specified for the other primitives. The general form of the primitive attribute inquiry function takes the form:

gk_q_<primitive> (List of one or more *Attribute* = *Value* terms)

where <primitive> is one of the GKS output primitives. The attribute(s) depends on the primitive.

In the example above, the attributes are atomic constants. It is possible to have attributes which are structures. For instance, to inquire the line type aspect source flag we can invoke the predicate:

gk_q_asf(line(type) = *Flag*).

This form allows the application programmer to program goals such as:

```
/* Gather 'in a bag' the primitives and their indices whose asf's are bundled
   'bag' has three arguments (Element, Condition, Bag)
   =.. is a PROLOG operator that takes Prim with Index to form
                    the term Prim(Index) */

inquire_bundled_asf(Bag):-
                    bag((Prim, Index),
                        (gk_q_asf(Attribute = bundled),
                         Attribute =.. [Prim, Index] ),
                        Bag).
```

The last example we consider also deals with structured parameters. Consider the function INQUIRE PREDEFINED PRIMITIVE REPRESENTATION. For each primitive, namely, polyline, polymarker, text etc most GKS implementations hold in the workstation description table a structure representing the primitive representation. For a polyline, this representation has four attributes, namely, the polyline index, the line type, the line width scale factor and polyline colour index. While the applications programmer may wish to query a particular attribute, from an implementation standpoint this would require accessing the internal GKS tables once for each attribute queried. In general it is faster for our implementation for the C-GKS internal structures to be accessed once and for the applications programmer to extract the particular attributes of interest. This is easily done in PROLOG with the use of the don't care variable '_'. Thus, the call,

```
gk_q_rep( WS, line(ID, _, _, Colour) )
```

will instantiate *Colour* to the polyline colour index only if polyline index equals *ID* on workstation *WS*. Other variations can easily be described.

5 Concluding Remarks

In this paper we have attempted to show that it is possible to define a PROLOG binding for GKS in a manner that makes declarative graphics programming a viable proposition yet at same time conforming to the guidelines laid down by the GKS standards specifications. The few examples presented in this paper highlight some of the potential flexibility that PROLOG achieves through the use of named attributes and structured arguments. Moreover, this flexibility is achieved without sacrificing both the readability and conciseness of the application programmers code.

Since we have only barely hinted at implementation details, it should be remarked that the interface to the GKS functions as illustrated by the examples above can be written entirely in PROLOG though at the present time this is likely to result in an unreasonably slow implementation. Our particular implementation is written in PROLOG and calls GKS routines written in C. (A full implementation [4] of C-PROLOG/ GKS is now available.) The C-GKS system [2] on which our implementation is based utilizes macros for the inquiry functions which in turn take as arguments pointers to the various GKS tables. It is a relatively straightforward matter to translate PROLOG attribute names to the C table pointers.

Acknowledgements

This work has been carried out as part of the ACORD project supported by the ESPRIT programme.

References

1. Anon, "Special GKS Issue," *Computer Graphics* (1984).
2. M. Bakker (ed.), *The GKS Reference Manual,* Stichting Mathematisch Centrum, Stichting Computer Grafiek, Systeem Experts b.v. (1986).
3. G. Enderle, K. Kansy, and G. Pfaff, *Computer Graphics Programming: GKS - The Graphics Standard,* Springer-Verlag (1984).
4. R. Krishnamurti (ed.), "Prolog/ GKS Reference Manual," Technical Report, EdCAAD, University of Edinburgh (1987).
5. R. R. Martin and C. Anderson, "A proposal for an ALGOL 68 Binding of GKS," *Computer Graphics Forum* **4**(1), pp.43-57 (1985). (Reproduced in this Volume.)
6. F. C. N. Pereira, "C-Prolog User's Manual," Technical Report, EdCAAD, University of Edinburgh (revised 1984).
7. D. S. H. Rosenthal and P. J. W. ten Hagen, "GKS in C," pp. 359-370 in *Eurographics '82*, ed. D. S. Greenaway and E. A. Warman, North-Holland (1982).
8. M. Slater, "Pascal Interface for GKS 7.2," BSI OIS/5/WG5/207, BSI Working Group on Computer Graphics (1984).
9. P. Sykes and R. Krishnamurti, "A Proposal for a Prolog binding to GKS," Technical Report, EdCAAD, University of Edinburgh (revised 1985).