Separating Graphics from Application in the Design of User Interfaces

F. NEELAMKAVIL* AND O. MULLARNEY
Interactive Graphics Systems Group, Department of Computer Science, Trinity College Dublin, Ireland

Until recently work in the area of User Interface development has been almost entirely* ad hoc. *Implementors have had to rely on their own judgement not only in the area of the interface appearance and dynamics, but also in the internal structure of interaction management. Considerable confusion exists in defining what constitutes a good User Interface, and in the management processes required to construct them. Insufficient software foundations and a lack of formalisms, tools and methodologies are contributing factors. To some extent these problems have been addressed by the User Interface Management System (UIMS) community, primarily for those working on large systems. This paper describes a methodology, developed as part of the design of a User Interface Manager, aimed at providing implementors with a structure for building interfaces which are separable from their underlying applications.

Received July 1988, revised May 1989

1. INTRODUCTION

Arguably the most important area of an interactive software system is the user interface. The user's satisfaction relies heavily on their appreciation of the nature of the underlying application and of the interface itself[7] and without proper structure in the interface, much of the functionality of a system will be hidden. The effect of good interface design is to increase productivity, user satisfaction and to facilitate full exploitation of the system beneath. Well structured interfaces should offer consistency in methods used to complete different tasks, strong reflection of the nature of the application, and also allow the user to configure the interface to suit individual requirements.

Traditionally, the user interface was the last part of the application development process. It has now been accepted that the user interface is more important than just an additional component; it is an integral part of the whole system. While modern structured software techniques have made some contributions towards systematic development of user interfaces, the progress has been comparatively slow mainly due to the complex, expensive, dynamic and highly specialised nature of graphics technology and a lack of widely accepted international graphics standards.

In the past, graphic interfaces to application software have been developed using *ad hoc* and low level techniques. The interface is frequently buried within the application code in a manner which both hinders maintenance, and makes the reuse of the methods within a different application certainly time consuming, and often impractical.[15] Even in non-graphic interfaces these methods have resulted in inconsistent interaction methods and badly structured systems. The production and maintenance costs are high for systems using interactive graphics interfaces, demanding up to 50% of entire system development time[22] and as much as 60% of maintenance costs, and so tools and methods are critical to the development of quality interfaces.

In order to raise interface development from an art to a science, research has been plentiful[14,9] in the area of User Interface Management Systems (UIMSs), tools which aim to provide support for both the development and execution of interactive graphic interfaces. In the course of our work on a User Interface Manager we developed a methodology for the production of separable graphics interfaces on top of structured, object oriented, applications. Our goal was to provide a general structure for the construction of graphic interfaces around applications, interfaces which are both separable from the application and rapidly reconfigurable.

2. INTERACTION TECHNIQUE LIBRARIES

The traditional method of developing user interfaces is to let the users build their own interfaces from scratch, using general graphical libraries such as GINO-F,[6] CORE[1] and GKS.[8] The disadvantages of this approach include: discouraging people from building user interfaces, generating bad interfaces and producing inconsistent interaction styles.

The next major development was in providing a set of tools, called Interaction Technique Libraries (ITLs), to help the graphics interface developer in building user interfaces. These are collections of (editable) logical input and output devices, providing developers with 'off the shelf' techniques for both collection and display of data by graphic means. These devices not only cut development time, as not all of the interface needs to be written from scratch, but can also improve the quality of an interface as the devices should be consistent in their design. Indeed, consistency between interaction methods within a single interface, and across a range of interfaces, is a critical area in user satisfaction.[7] In addition, the containment of the graphics code within a library naturally separates it from the internals of the underlying application, making the tasks of debugging and reconfiguring the interface considerably easier. However, these tools are designed to ease the task of implementation, and take no account of the underlying structure. The appearance of the interfaces constructed by this approach can strongly reflect the conceptual model of the designer of the toolkit and may not suit all applications or users.

* To whom correspondence should be addressed.

3. USER INTERFACE MANAGEMENT SYSTEMS

The name User Interface Management System (UIMS)[20] was coined to describe a system which plays a comparable rôle in interface construction as compilers do in code production. They consist of two main toolsets, the first to assist in the construction of the interface, and the second providing runtime support. UIMSs take a formal description of the interface structure, a form of dialogue specification (Fig. 1), and from this they build the application specific component of the interface. This specification contains information on the flow of control and data through the interface, and may also contain some information on its appearance. The UIMS converts this into an executable form, linking it to an application, graphics system and possibly an ITL. Forms of extended Backus-Naur Form (BNF), Augmented Transition Networks (ATNs) as well as less formal approaches have been used as methods for the specification of the dialogue, though they can be shown to overlap in the types of dialogue they can define.[13]

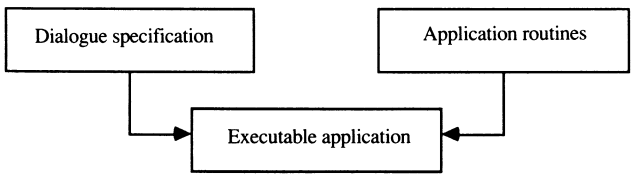

Figure 1. Generating and combining Interface and Application.

The aim of UIMS research is to produce interface compilers, collecting the mechanisms of interaction into a single system, and apply techniques appropriate to the communication defined in the dialogue description. With the burden of graphics programming and device management removed from the designer's shoulders, work in interface development can concentrate on good design of the form of the communication, using formal methods to express concepts rather than methods.

One model of the structure of user interface is that proposed by Foley and Van Dam[10,11] which breaksdown the interface into four major parts: the *conceptual*, *lexical*, *syntactic* and *semantic* designs. This model corresponds quite closely with the Seeheim model of UIMS's (Fig. 2), which was developed at the Graphical Input Interaction Technique (GIIT) meeting[23] and is described in Green (1985).[12]

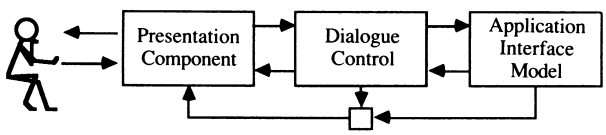

Figure 2. UIMS structure.

Although not initially proposed as a model for implementation, but rather a logical breakdown of the functionality, this scheme has been used in the development of a number of research UIMSs. The lower three levels of Foley and Van Dam's model are clearly partitioned in the different modules of this UIMS. The Presentation Component manages the lexical level of the interaction, controlling all screen images, whether they are representations of application objects or the interaction devices of the system. It is driven by the Dialogue Control module, the syntactic level of the model. This component orders the interaction by requesting the Presentation Component to update the screen or to initiate interaction through a given device. The results of device interactions are passed back from the Presentation Component to Dialogue Control where they are collected, before being bundled as a call to the Application Interface Model. This is the semantic level of the UIMS, containing descriptions of the application data types and a description of all user callable routines. The lower pipeline in the diagram illustrates that it may be necessary to have application graphics passed directly to the Presentation Component, for example in CAD systems, where elements of the graphics display are not directly connected to the dialogue.

A number of UIMS's have been designated and built, with greater or lesser success, some based on Interactive Automata[17–19] and others based on event driven methodology.[12,5,9] UIMSs are large systems and have their own drawbacks: specification is a complex and arduous task, and there is no guarantee that what will emerge is what is required. The best interfaces are still produced by the equivalent of assembly language programming, with ITLs filling the role of macro facilities.

4. THE PAPILLON SYSTEM

Our methodology came about during the design of a Configurable Graphics Subsystem for Computer Integrated Manufacturing (CIM) under the Papillon project, supported by the Commission of the European Communities (CEC). We needed a system configurable in terms of both underlying application and also interface structure and appearance. One of the first reactions on examination of the then existing systems was the realisation of the generally unstructured nature of interface construction. Therefore, it was decided to carry out a thorough examination of methods for separation of the interface from the application, as this was clearly a necessary condition for a separable and easily maintainable interface. The system design was being conducted using the Vienna Development Method (VDM)[2,3] for specification, with implementation in ADA. The design also stems from the Abstract Data Type (ADT) form of Object Oriented Design (OOD).[4] The formalism of VDM and the structure of ADA are both suited to the design of large, modular systems of the kind we envisaged. This design produces structured hierarchical models and applications, and it was felt unreasonable to lose this structure in a UIMS which did not follow this design.

Published literature on UIMS design did not contain much reference to application structure, regarding it as the least important element of the system. In general UIMSs have been developed independently of sound formal methods, and are somewhat unstructured in implementation; naturally this leads to high maintenance costs for the tools themselves. Our aim was to develop tools to assist in the tasks of both building an application and developing an interface on top of it, and therefore the first step was to investigate possible structures for the construction of prototypes of the intended system, and tools to accelerate that task.

The current version of the Papillon prototype system is application independent and is portable and adaptable

with respect to different applications and hardware. This is achieved through modular design and the use of modern software engineering methods. The set of tools provided simplifies the task of building applications and supports the automatic generation of user interfaces without direct coding. The re-use of components is an important feature of the system and different graphical user interfaces can be easily generated and evaluated at run time with no need for recompilation.

5. THE PAPILLON MODEL

The Papillon model was developed in the search for a methodology for producing structured interface to applications constructed using ADTs. The model rearranges the components of the Seeheim model (Fig. 2) to provide a separable, object oriented interface structure and consists of five modules (Fig. 3), linked together to form a complete interface to a single object or ADT. The acronyms come from the logical function of each module, the CONtrol, the REQuirements, the SEMantics and the REPresentation. Each of the REQ, SEM and REP packages have a similar structure, and they implement the three identifiable stages of command execution – acquiring the data, performing the command and providing feedback. The application interface MODEL is a representation of the application from the viewpoint of the user interface and therefore defines the semantics of the application. This set of five modules must exist for each ADT in the application, and they are linked together in the same hierarchy as the objects they act for. This has the effect of mirroring exactly the application structure within the interface, and providing logical connections between user-commands and objects on which they act.

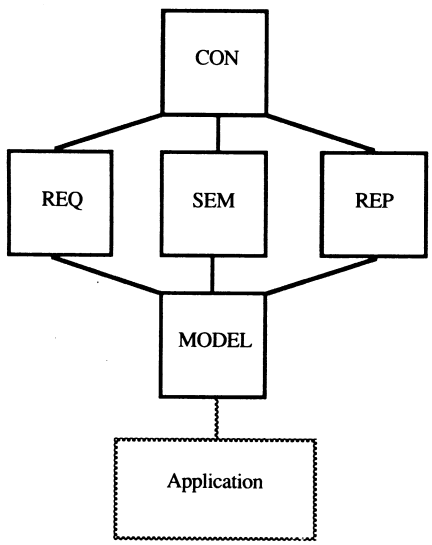

Figure 3. The Papillon Model.

In this model the communication between user and application is directed exclusively through the five modules of the interface. The collection of data for the application, and the display of the current condition and results of user action are all removed from the application. The application can be regarded as being 'pure', containing no direct calls for communication, and no graphics code at all. The separation of interface from application is beneficial in many ways: the modularity of the interaction code makes it easier to home in on errors, as the state of the interaction directs the developer to a specific routine in a specific module. In addition, development and modification of both application and interface can be performed independently, in the knowledge that the interdependence is minimal.

In the following sections each of the modules is described in detail, indicating the form that interaction takes, and how the interface communicates with the underlying system.

5.1. The CONtrol package

This package forms the first visible element of the model. For each data type implemented as an ADT, there is a collection of functions/procedures which may be invoked to modify or view an object instance. The CON package implements a directive based interface to those operations visible to the user (in current implementations this is done through a Graphical Kernel System *Choice* device). This allows the user to operate on the current object class by selecting from a menu those commands which apply to object instances. Standard commands for view manipulation and other application independent commands (panning, zooming, etc.) are also made available and allow the user to browse through the displayed data.

To initiate an action, the required menu option must be first selected. If this is a command which applies to the object in its entirety – scheduling of a plan or object deletion – the REQ, SEM and REP of the current object class must be invoked in turn to perform the action. The CON package, having ascertained the users choice, will call a function in each of the three packages which will collect any required data, call the ADT (application) through the Model package to perform the task and then update the representation of the object instance. Control will return to the CON package when these calls have completed.

Commands to edit the fields or sub-objects of the current level will involve the user moving through the hierarchy of the interface (Fig. 4). Modification of a field of an object implies editing the sub-object which makes up that field. For example, if the object were an object of type A with a sub-object of type B, then modification of the field B would require the user to select the 'EDIT B' command and indicate in which object the instance of B to be edited lay. This would pass control from the A's

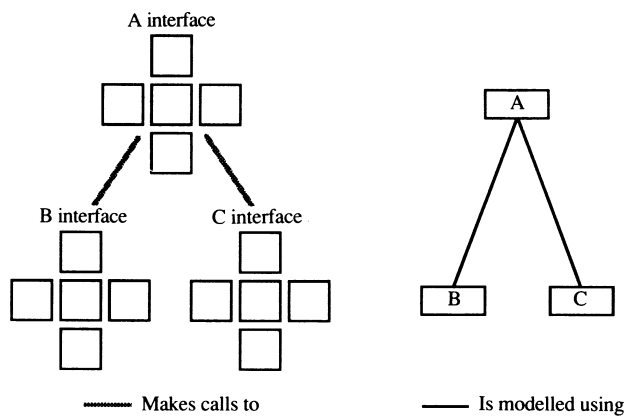

Figure 4. Interface and object hierarchies.

CON package to the B's CON, where the editing functions for B type objects would become available.

5.2. The REQuirements package

This package corresponds (in concert with the CON package) to one part of the lexical level of the interface, and to the input portion of the presentation component of the Seeheim model. The package is used to collect any other items required for the operation, in most cases simply collecting the parameters for the object modification or creation. This process may be interrupted to drop down a further layer in the hierarchy of objects in order to create an instance of a sub-object to be used in the operation being performed on its parent. Initially the REQ package was simply concerned with collecting lexical information, ensuring that the syntax of the command was correct. This delays feedback, in the case of erroneous input, until the SEM package is called, and the error detected by the Model. However, by making the Model package available to the REQ, some limited semantic checking can also be performed here, producing more rapid feedback. Calls must be present in the Model package to allow for the type of checking that the REQ wishes to do. If the user is directing the application to set the value of a field, the REQ should be able to call the Model package to ensure that the value given is a reasonable one for that type, any more complete checking – e.g. that the value is reasonable in context – can be left until later.

5.3. The SEM package

As the name implies, this package implements the semantic level of the interface. Once the call to the REQ package has completed successfully, an identical routine in the SEM package is called by the CON which makes a call to the underlying application, providing the parameters collected by the PRE section. All commands which modify the application data structure are encapsulated at this level, insulating the application from the interface. Single commands invoked by the user may split into a number of calls required to perform the task, depending on the structure of the underlying model – if it too is object oriented then the calls will match in structure, but prototypes of the interface structure have been built on top of non-object oriented applications.

The SEM package is also responsible for the trapping of error conditions which may be raised within the application if the semantic content of the call is invalid. Error trapping must provide a response to notify the user of the failure of the call, and also ensure that any information necessary is passed back to the CON package to allow the interface to cope gracefully with the error.

5.4. The MODEL package

The model package implements the equivalent of the Application Interface Module of the Seeheim model. It contains the definitions of all routines which are externally visible, and the declarations of application specific data types which must be used to invoke these routines. Ideally this level is made up of the package specification of an ADT, since this encapsulates all the information that is required, and it is in the most suitable form, matching the form of the interface above. This package is called by the SEM package, and as stated previously, there is no requirement that the visible functions match those presented to the user, since the mapping from the users model to the application structure is managed by the SEM package. Other modules in the interface may call the Model level, but are restricted to making inquiry functions, as it is important in maintenance and debugging to have the application and interface as separate as possible. As all modifying calls must pass through the SEM to the Model, tracking down errors and remodelling the application have become localised tasks.

5.5. The REPresentation package

This package is perhaps the most interesting in the model. The function of the package is to provide facilities for representing the application data model to the user, and so it is basically the output portion of the Presentation Component of the Seeheim model. This representation can be stored in a number of ways, through user/designer editable graphical data structures, or simply hardcoded into the package. Whatever the form, the separation of the representation into a single package does allow for rapid redesign of the appearance of the system.

Since the REQ package is invoked in a similar manner to the others in the model, by being called from the CON, it will receive as parameters only those that are passed to the SEM package. For this reason it is important to ensure that the routines in the Model package called by the SEM are well designed, and do not have side-effects. If only those objects received as parameters are modified, then the REP has simply to update those objects which appear in the call.

The structure and the method of operation of the REP module depends on the organisation of the data model. The question of whether the interface and the application should share the same data model is a major issue to be resolved early in the design of the user interface. Shared data models are highly desirable for efficient implementation of the emerging direct manipulation interfaces; however, this can only be done at the expense of separability which is an important characteristic of a good user interface. It should be noted that the role of the user interface and the functionality of the application undergo many changes during the life time of an application, and therefore it is important to preserve the independence of the application and the user interface in order to facilitate the modification of the application without affecting the interface and vice versa. The question of allowing shared or separate data models in the design of user interfaces is examined further in Section 6.

5.6. Assembling the Interface

The structure described here is intended to be constructed for each object in the application hierarchy and the set of tools provided supports the assembly of the interface without additional programming. For each object, the CON package provides access to all of the functions which may be called to modify an instance of the class, and the user may climb up and down through the

hierarchy as the creation or selection of sub-objects is performed. However, this alone cannot be said to provide a complete user interface. Other tools are necessary to provide the user with the ability to browse, and to examine the data to various degrees of detail, as well as methods to structure the interface and make it more transparent. The design of this tree of interface elements must be connected in some way, and this structure reflected to the user. The method chosen is to have a window (or GKS workstation in implementation) in which each level of the hierarchy is viewed. As the user moves through the tree, downward movements open a window in which the item of the subclass will be viewed and edited (Fig. 5), and upward movements close the window of the level being exited and return control to the window from which the sub-object access was made. For each window there is a fixed set of menus and other devices which may be used, and the objects are always confined to a single window. This makes the interaction very ordered, and simplifies the task of managing the interface. As far as each level is concerned, they exist autonomously, drawing in their own window (or page of output), and collecting input through fixed devices. This paradigm mirrors the object oriented nature of the application model, and so will closely reflect the users model of the system if the conversion from real-world to object structure is a natural one.

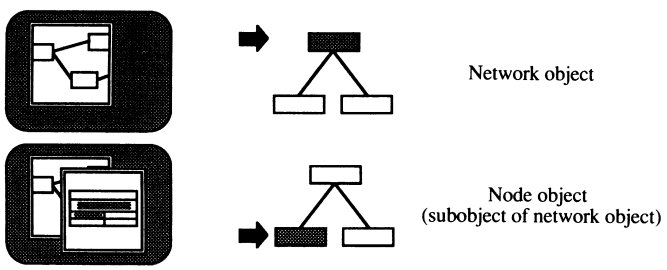

Figure 5. Window/object correspondence.

With this structure in place it has been easy to design a collection of application independent tools which provide run-time support for the interface. Tools are provided for panning and zooming, and for manipulating views of different classes of output pages. Similarly, general tools which perform other tasks, such as menu and window layout editors have been developed, which allow the designer or user to reconfigure elements of the interface appearance and content to suit their own particular preferences.

5.7. Evaluation of the Interface Manager

We have presented a methodology which can assist designers and implementors to achieve separable and easily reconfigurable graphic based user interfaces. Insulation of each element in the system from the other by using a well defined communication path is essential in reducing the overheads in the development of graphics systems. It also allows commonly used application independent tools to be constructed, and used across a range of systems developed using a common methodology. The techniques described here have been used in a number of graphics based interfaces, and particularly in the design and implementation of a graphical editor which has been used in further development of a User Interface Manager (UIM).

A number of applications belonging to different fields were developed using the Papillon Prototype which incorporates the methodology presented in this paper and the results are encouraging. An application consists of several objects and at run time, the application data is entered either by the user or directly by the 'driver' software. Application objects are displayed in their own windows and the operations on the objects, such as Add, Delete, Replace, Edit, View, etc are performed by selecting the appropriate menu item. A typical snapshot of the runtime system showing the state of a machine in a machine shop is given in Fig. 6. The user can switch between windows or open new windows containing different representations. The structure of the system is such that the representations can be easily edited at run time and the interface appearance modified without recourse to recompilation. Whenever an application changes the value of an object, that change is automatically reflected in the screen display.

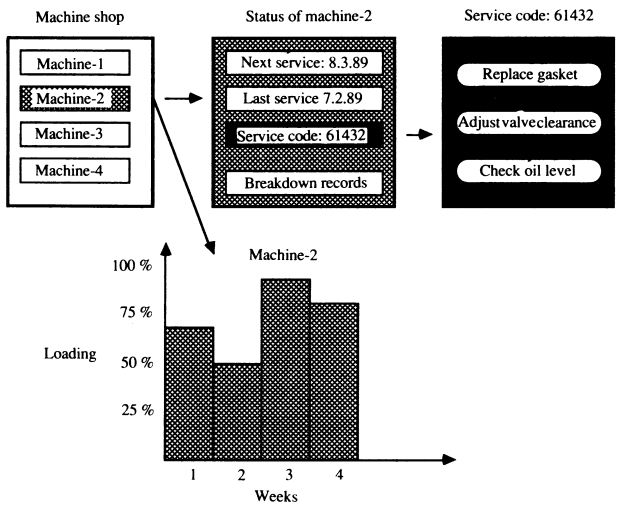

Figure 6. An Example of Graphics Interface.

However, experiments with the Papillon Prototype have highlighted the need for a more flexible interface structure presented to the end user, particularly when we are dealing with inexperienced users. Large applications may have many levels within the tree, and traversing this can be time consuming and possibly confusing in certain cases. These difficulties are overcome by collapsing the boundary wherever the logical difference between levels is weak and by making commands of different levels available concurrently. When concurrent menus are available, the user will perceive the options as equally valid, and the interface can deal with the mapping of commands to the internal structure. This flatter interface is used to homogenise objects which are similar in the users mind, and remove some of the more artificial boundaries imposed by OOD.

In Conversational Metaphor, the interface is seen as a conversation or dialogue between a user and the system; here, an object is represented by a name and manipulations are specified in some programming language. The Direct Manipulation Metaphor[16, 21] is aimed at providing the user the illusion of directly acting upon the objects of interaction without the intermediary

of the system. Currently we are in the process of extending our interface design technique to the area of Direct Manipulation (DM). This has become one of the most promising techniques in user interfaces, both transparent and flexible, and suitable for a broad spectrum of users with different levels of experience. The next stage in this research is to develop a direct manipulation manager, which will sit above the current level. The modularity of the interface structure so far developed provides a reliable and structured method for producing interfaces. Combination of this with an application independent DM manager would result in a user interface based on sound methodology, and exhibiting a direct and easily mastered external view.

6. SHOULD THE INTERFACE SHARE OR KEEP A SEPARATE DATA MODEL?

We have already mentioned in Section 5.5 some of the difficulties involved in sharing the same data model by both the application and the interface. Ideally the data model from which the representation is derived should be shared with the application. However, this requires that the application designer incorporate graphics fields in all objects, allowing these to be accessed by the REP package, or that the application model is modified to include them. This is not always possible, as the interface may be added to a previously written application which cannot be modified to suit the interface requirements. An alternative is to allow the REP to hold a corresponding data structure which holds only that information which corresponds to the purely graphical data (e.g. positioning, colour etc.), and have it query the application model for the other information it requires (field values etc.). If this second option is chosen, then the effect of functions with side effects becomes crucial. The REP module will inquire about only those objects received as parameters to the call, assuming that only they can be modified, in order to update the display. If other objects have been modified they will not be checked, and so consistency between appearance and actual state will break down. The former option of a shared interface/application model is more desirable, since it allows the possibility of direct manipulation, with the REQ package being able to access the shared model in response to user actions. The REQ package can then use positional information to determine objects the user selects by pointing and clicking. However, the nature of shared models runs contrary to the aim of separability, since the application should not be concerned with the maintenance of graphical data, such as the positional information etc. required by the REP module.

At the present time we are investigating possible approaches for resolving the problem of combining the interface and application data models by developing tools for use in the construction of the application which incorporate graphics functionality. By including graphics functionality within the predefined objects used to model the application data, it is possible to remove much of the burden of graphics coding, and of update management from the developer. This provides an elegant solution to the problem of data sharing which has previously resulted in the close coupling of application and display routines, and more recently the problems encountered in UIMS work.

7. CONCLUDING REMARKS

The Papillon prototype consists of three major subsystems: *Application Building Blocks*, *Graphics Editors* and the *User Interface Generator*. These subsystems were developed at different sites on different machines operating under different operating systems before integration into a single system. The configurability, separability and portability features of the prototype were tested by first building pure (no graphics) application software and then adding graphical user interface. In particular, we were successful in demonstrating that only a few days' work is needed to tailor a graphical user interface to a very large CIM application (about 50000 lines of code in Ada) developed at a different site on a different machine.

The research under Papillon project was carried out over a period of three years. The development of various tools, techniques, methods and the prototype itself were periodically reviewed by a group of independent experts, specially appointed by the CEC. The objective feedback from the reviewers were always a source of inspiration and helped the project achieve most of its aims. The Papillon system is still a prototype which needs further refinements before full integration into real CIM environments. A commercial software product based on the Papillon prototype has recently been announced by one of the partners in the Papillon consortium. A number of enhancements to the prototype including compatibility with non-GKS graphics packages and support for non-Ada systems are currently underway or under investigation. It is expected that the refined version will have applications not only in the integration of CIM components but also in the development of more general graphical software.

8. ACKNOWLEDGEMENTS

This work was done as part of the Papillon project, partly funded by the Commission of the European Communities under the ESPRIT programme. We are grateful to M. Mac an Airchinnigh, J. Drumgoole, P. Hickey and other members of the Papillon consortium for their contributions and in particular to Chris Chedgey (Genetics Software Ltd) who played a major role in the overall development of the Papillon prototype. Our thanks are also due to Dr R. Zimmermann, Project Officer, CEC, for his comments, criticisms, advice and help. Finally, many thanks go to the reviewers for making several suggestions for improving the quality and clarity of this paper.

9. REFERENCES

1. ACM SIGGRAPH, Status report of the Graphics Standards Planning Committee. *Computer Graphics* **13** (3), entire issue (1979).
2. D. Bjørner and C. B. Jones (eds), *The Vienna Development Method: The Meta-Language*. Lecture Notes in Computer Science 61. Springer-Verlag, Heidelberg (1978).
3. D. Bjørner and C. B. Jones (eds), *Formal Specification and Software Development*. Prentice Hall, New Jersey (1982).
4. G. Booch, *Software Engineering with ADA*. Benjamin/Cummings, Menlo Park, CA (1983).

5. W. Buxton *et al.*, Towards a comprehensive user interface management system. *Computer Graphics* **17** (3), 35–42 (1983).
6. CAD Centre, *GINO-F User's Guide*. Cambridge, England (1985).
7. H. W. Dzida and W. D. Itzfeldt, *Factors of User Perceived Quality of Interactive Systems*. Report Nr. 40 of the Institut Für Software Technologie, GMD (1978).
8. G. Enderle, K. Kansay, and G. Paff, *Computer Graphics Programming*. Springer-Verlag, Heidelberg (1984).
9. G. Fischer, Human–computer interaction software: lessons learned, challenges ahead. *IEEE Software* **6** (1), 44–52 (1989).
10. J. Foley and A. Van Dam, *Fundamentals of Interactive Computer Graphics*. Addison Wesley, New York (1982).
11. J. Foley, Models and tools for the designers of user-computer interfaces. In *Theoretical Foundations of Computer Graphics and CAD*, edited R. Earnshaw, Springer-Verlag, Heidelberg (1988).
12. M. Green, The University of Alberta UIMS. *Computer Graphics*, **19** (3), 205–213 (1985).
13. M. Green, A survey of three dialogue models. *ACM Transactions on Graphics*, **5** (3), 244–275 (1986).
14. H. R. Hartson, and D. Hix, Human computer interface development: concepts and systems. *ACM Computing Surveys* **21** (1), 5–92 (1989).
15. H. Lieberman, There's more to menu systems than meets the screen. *Computer Graphics* **19** (3), 181–189 (1985).
16. B. A. Myers, Creating interaction techniques by demonstration. *Computer Graphics and Applications* **7** (9), 51–60 (1987).
17. D. R. Olsen, SYNGRAPH: a graphical user interface generator. *Computer Graphics* **17** (3), 43–50 (1983).
18. D. R. Olsen, Pushdown automata for user interface management. *ACM Transactions on Graphics* **3** (3), 177–203 (1984).
19. D. R. Olsen, Input/output linkage in a user interface management system. *Computer Graphics*. **19** (3), 43–50 (1985).
20. G. Pfaff (ed.), *User Interface Management Systems*. Springer-Verlag, Heidelberg (1985).
21. B. Shneiderman, *Designing the User Interface*. Addison-Wesley, New York (1987).
22. J. A. Sutton and R. H. Sprague. *A Study of Display Generation and Management in Interactive Business Applications*. IBM San Remo Laboratory, Technical Report R.J. 2392 (31804) (1978).
23. J. J. Thomas, Graphical Input Interaction Technique (GIIT): Workshop Summary. *Computer Graphics* **17** (1), 5–30 (1983).