Automatic Generation of Interactive Applications

by

Emanuel G. Noik

Technical Report 90-7

February, 1990

Computer Science Department
University of British Columbia
Vancouver, B.C. V6T 1W5
Canada

Abstract

As user interfaces become more powerful and easier to use, they are often harder to design and implement. This has created a great demand for tools which help programmers create interactive applications. While existing interface tools simplify interface creation, they typically focus only on the interface, do not provide facilities for simplifying application generation, and are too low-level. We have developed a tool which automatically generates complete interactive applications from a high-level description of the application's semantics. We argue that our system provides a very simple yet powerful environment for application development. Key advantages include: ease of use, separation of interface and application, interface and machine independence, more comprehensive programming aids, and greater potential for software reusability. While we tend to focus on the practical motivations for using such a tool, we conclude that this approach should form the basis of an important category of interface tools and deserves further study.

1 Introduction

During the last ten years we have witnessed a growing trend in the use of interactive applications which employ sophisticated graphical user interfaces. This increased demand for easy-to-use software has created a great need for programming tools which help to design and implement user interfaces. A number of different approaches have emerged [HH89, Mye89].

At the University of British Columbia, we set out to tackle this problem with the following goals in mind.

- The tool should be easy to use. In particular, interfaces should be defined at a very high level. The programmer should not be burdened with details.
- The tool should provide an integrated programming environment – it should not merely create interfaces.
- The tool should let the programmer create complete applications by combining existing components with his own.
- Applications should be portable across machines and user interfaces.
- The tool should support the notion of a class hierarchy of objects with inheritance. Operations which manipulate these objects are to be defined.
- The tool should be able to generate interfaces which support a number of interactive styles.

These objectives were met as follows.

- Describe the semantics of the application rather than the syntax of the user interface (by specifying **what** an application should do rather than **how**).
- Generate the entire application, not just the interface. In this way it is possible to embed more information about the environment in the tool. This can be used to further simplify the programming task as a number of difficult issues such as communication and control can be eliminated.
- Isolate environment and interface specific information in the code generation portion of the tool. This makes it possible to create applications for any platform, user interface, and interactive style. Existing user interface generation toolkits or UIMS's may be employed in realizing the desired interface. Thus the tool provides a layer on top of current technology – the specification is at a much higher level and in contrast to most current systems, provides a means for creating portable interface and style independent applications.

The resulting tool, NAAG (Not Another Application Generator), embodies these ideas. Before describing it, we review some motivations for using interface tools in general and application generators in particular.

2 Motivation

We clarify the need for programming tools to aid the design and implementation of user interfaces by reviewing some of the key reasons for using such tools.

Although this paper is primarily concerned with the restricted case of generating interactive applications, we review some key advantages of interface tools at several levels. At the most general level, we look at user interface tools and the advantages they offer. Next, we look at a class of tools which automatically generate user interfaces. Lastly, we elaborate on the advantages of generating interactive applications.

2.1 User interface tools

Interface tools have been created primarily for the purpose of making interfaces cheaper and easier to design and implement. Myers [Mye89] lists the following advantages derived from the use of interface tools:

1. Resulting interfaces are better:
   - Designs can be rapidly prototyped and implemented.
   - It is easier to incorporate changes discovered through user testing because the interface is easier to modify.
   - One application can have many interfaces.
   - More effort can be expended on the user interface tools than may be practical on any single interface because the tools will be used repeatedly.
   - Different applications will have more consistent interfaces because they have been created with the same tools.
   - It is easier to investigate different styles for an interface.
   - It is easier for many specialists to be involved in designing the interface.
2. The interface code will be easier to create and more economical to maintain:
   - The code will be better structured and more modular because it has been separated from the application.
   - The code will be more reusable because the tools incorporate common parts.
   - The reliability of the interface is higher.
   - Interface specifications can be represented, validated, and evaluated more easily.
   - Device dependencies are isolated in the user interface tool, so it is easier to port an application to different environments.

2.2 Automatic generation of user interfaces

Tools which automatically generate interfaces based on the application's semantics, have the following advantages over and above the ones listed in the preceding section.

1. Interactive applications are easier to create and maintain:
    - The interface can be generated from the information implicit in the description of the application's semantics. For example, consider an operation which requires as one of its inputs an integer value selected from a given range of integers. In most systems the programmer would have to describe **how** such a value might be obtained – using a *slider* gadget for example. A programmer using a tool which relies on semantic information, however, would describe this input simply as an integer in a range – the tool would decide how to best obtain such a value using available resources.
    - The ability to automatically generate interfaces will be more important as interfaces become more sophisticated. An interface tool can be seen as a combination of compiler and expert system. It must use large amounts of information to solve a translation problem – creating a low-level description from a high-level one. As the gap between the source and target languages widens, the tool must employ more information and smarter translating schemes to carry out the transformation.
2. It is possible to achieve a greater separation of interface and application:
    - Interface or dialogue independence implies that the dialogue (interface) and the computational (application) components are independent – changes in one should not affect the other [HH89]. Since the tool generates the interface, the programmer can treat the interface as a black box whose behaviour can be described by a set of rules. This set of rules defines an internal dialogue – a protocol which the dialogue and computational components must adhere to. However, as different tools may employ different internal dialogues, portability may be sacrificed.
    - It is easier to generate interfaces which are tailored to a specific user community and employ a unique style of interaction. Where alternatives exist, the tool can be designed to pick the most appropriate method of communicating with the user. For example, if the interface must obtain an integer from a given range, the tool may choose to use a *slider* gadget if the range is large, and a *numeric text* gadget with increment/decrement buttons if the range is small. At a more abstract level, a user-modelling facility can be used to guide the human-computer conversation. Tools which automatically generate interfaces can be used in conjunction with such facilities to dynamically create the most appropriate interface based on an on-going evaluation of the interaction.
    - Since the tool uses semantic information only, it can generate interfaces which are more uniform and reliable (at the cost of reduced flexibility as the programmer is given less control over what is generated).

While most interface tools emphasize the task of describing the interface, this class of tools regards the interface as a side-effect – something which can be generated automatically from the deeper semantic description.

2.3 Automatic generation of interactive applications

Application generators take the preceding approach one step further by creating the entire application rather than just the interface. This has a number of inherent advantages.

1. It represents the simplest model for application development:
   - Minimum programming effort is required to create interactive applications. The programmer does not have to create the dialogue component, nor write internal dialogues to communicate with an automatically generated dialogue component. As such, he can focus his efforts on creating the computational component.
   - Issues such as control and communication are eliminated. Typically these are the greatest impediments to interface development – even once the programmer has mastered this aspect, there is no guarantee that the programs he writes will be portable, as most toolkits and UIMS's handle these issues differently.
   - Many tasks can be eliminated or simplified. For example, most window systems require the programmer to supply routines for resizing and repainting windows, event processing, and resource management. Most of these tasks can be performed by the tool, while others such as mouse event processing can be greatly simplified.
2. More information can be embedded in the tool:
   - Since the tool has intimate knowledge of the target environment, it can generate code which is optimized. For example, there are typically many ways of performing graphics operations in a given system. The tool can be designed to choose the most efficient means to carry out computationally-intensive tasks.
   - A more complete programming environment can be defined. Since the tool generates the entire application, it can utilize other tools to reduce the programming effort. For example, a make utility can be invoked to automatically compile code generated by the tool and combine it with external object modules to obtain an executable version of the application.
   - The tool can be used to carry out many of the tasks that are usually performed by the programmer – this makes it possible to produce applications which are more modular, contain fewer bugs, and use resources more efficiently than hand-coded programs.
3. Programming model is more consistent:
   - Applications are easier to port, as the programmer is not required to learn a new environment. It is possible to port existing applications without change simply by recompiling them using the modified version of the tool.

- Although the ability to *escape* from the tool can be important in certain circumstances, restricting functionality has the advantage that the applications are more portable – the programmer does not have to rely on mechanisms outside of the tool and cannot introduce compatibility problems.

4. The tool is more portable:

   - The very high-level specification makes the fewest assumptions about the nature of the interface or the underlying application. This allows the tool to choose the most appropriate mechanisms to realize the interface. For example, while one system may use programmed function keys to obtain user inputs from menu panels, another may employ a mouse to select items from drop-down menus accessed from a menu bar. A third may use pop-up menus with pull-rights and dialog boxes. This idea extends beyond being able to use different interactive styles or devices as the only information being captured is **what** must be accomplished – not **how**.
   - Can use existing tools to realize interfaces. Each target environment typically has several interface tools which are capable of generating interfaces consistent with that of most applications created for that target. For example, the Apple Macintosh computer has a very standardized interface, as does the NeXT, yet a functionally equivalent application would possess a different *look and feel* depending on which platform it was developed for.
   - Code generation module can be modified to create source code for a variety of operating systems, languages, and user interfaces. As more information is embedded in the tool, the tool can be adapted to more environments.

2.4 Summary

In this section we studied the advantages of interface tools at three levels of specialization. As the tools become more specialized they are typically less complex and easier to use. This gain comes at a price – reduced generality. This, however, does not have to be a crippling deficiency. One of the key considerations in any task is the choice of the right tool for the job. It is undesirable to use a low-level tool where a higher level tool will suffice. Similarly, if the problem cannot be expressed in a high-level fashion, a more general tool should be sought. Although application generators are limited with respect to the kinds of applications they are capable of producing, they nevertheless have a number of important advantages and are of significant practical value.

3 NAAG

3.1 Introduction

NAAG is a tool which generates interactive applications from a high-level description of the application's semantics. Essentially, it enables a programmer to create an interactive application by specifying **what** type of tasks the application is to carry out rather than **how**. As a direct consequence, NAAG applications are portable not only across machines, but across user interfaces.

Existing user interface generation systems can be extremely time-consuming to learn and difficult to use. NAAG insulates the programmer from many complexities at some cost in terms of reduced functionality. For example, NAAG provides for interface independent graphics. The programmer simply calls predefined routines to output graphical elements in his own preferred coordinate system. NAAG maintains its own display list and performs such tasks as the repainting and resizing of graphics windows while maintaining correct aspect ratios.

Note, however, that NAAG is much more than an interface generator–it is a programming environment consisting of a very-high-level language, macro preprocessor, compiler, and automatic make utility. Existing UNIX tools are used to realize a number of these components. Furthermore, NAAG allows the programmer to define classes, and operations which can be applied to objects belonging to these classes. NAAG generates graphical, object-oriented user interfaces, suitable for interacting with the objects which the application creates.

3.2 Application programmer's model

This section provides an overview of the NAAG programmer's model. NAAG is described in greater detail elsewhere [Noi90].

From the NAAG perspective, an application consists of an organization of operations which manipulate objects belonging to a number of classes. NAAG provides the means for realizing these abstractions as C functions and data types. Since operations are the main building blocks of NAAG applications, we start out by looking at some examples.

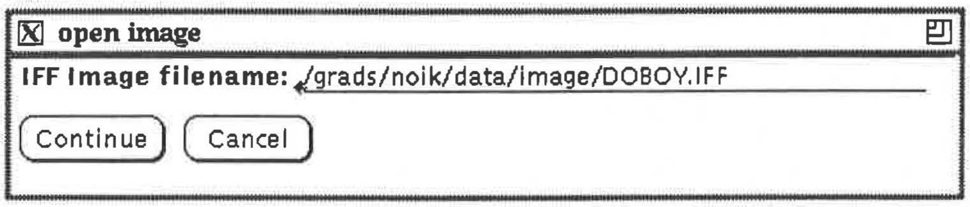
First, consider one common type of operation: reading an object description from a file. Typically this operation requires a single input: the filename. Figure 1 contains a NAAG program segment which describes an operation for reading image data from a file, while Figure 2 shows the resulting dialog box generated by NAAG (for a specific interface – see implementation notes). The **argument** construct is used to describe what type of value the application must obtain from the user in order to carry out the operation. Table 1 shows the argument types which are currently implemented in NAAG. As a further example, Figure 3 shows a more involved operation description, while Figure 4 contains the resulting dialog box.

Given a number of operations, the next step would be to organize the operations into a hierarchy of menus. While this information is used to establish a semantic grouping of operations based on some criteria, there is no implication of how these groupings will be realized. Figure 5 contains a NAAG description of one menu in a vision application, while Figure 6 shows a part of the menu tree which was generated by NAAG. Menus contain menu items which may be other menus, operations, or stubs – operations which have not been implemented.

By using include files it is possible to easily assemble existing menus and operations into complete menu hierarchies. As the programmer does not have to be familiar with the contents of the included files, this method of composing new applications from existing components is not only

```
/*--- file:  LoadImage.op ---------------------------------------------------*/

.operation LoadImage {
   .class .none                     /* operation does not belong to a class */
   .libraries { "-limgio" }         /* object library which contains function */
   .label "open image"
   .help "This routine opens images stored in IFF format"

   .argument {
      .text .max_length 40 .initial "/grads/noik/data/image/DOBOY.IFF"
      .label "IFF Image filename:"
      .help "Enter the filename of the image to be opened"
   }

   .result {                        /* result is an Image object */
      .class Image
   }
}

/*--- end of file:  LoadImage.op -------------------------------------------*/
```

Figure 1: A NAAG description of an operation which reads image data from a file.



IFF Image filename: /grads/noik/data/image/DOBOY.IFF

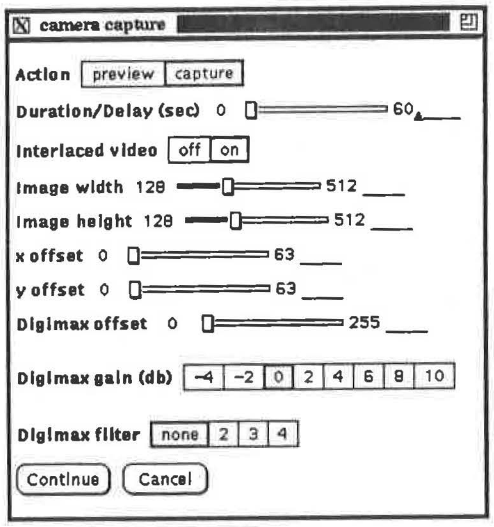
Figure 2: *LoadImage* dialog box generated by NAAG.

```
/*--- file:  camera.op ----------------------------------------------------*/

.operation camera {
   .class .none
   .libraries { "-limglib" }
   .label "camera capture"
   .help "Capture an image with a camera"

   .argument {
      .set { "preview" "capture" } .initial 1
      .label "Action" }
   .argument {
      .range .minimum 0 .maximum 60 .initial 0
      .label "Duration/Delay (sec)" }
   .argument {
      .flag .initial 1
      .label "Interlaced video" }
   .argument {
      .range .minimum 128 .maximum 512 .initial 256
      .label "Image width" }
   .argument {
      .range .minimum 128 .maximum 512 .initial 256
      .label "Image height" }
   .argument {
      .range .minimum 0 .maximum 63 .initial 0
      .label "x offset" }
   .argument {
      .range .minimum 0 .maximum 63 .initial 0
      .label "y offset" }
   .argument {
      .range .minimum 0 .maximum 255 .initial 0
      .label "Digimax offset" }
   .argument {
      .set { "-4" "-2" "0" "2" "4" "6" "8" "10" } .initial 2
      .label "Digimax gain (db)" }
   .argument {
      .set { "none" "2" "3" "4" } .initial 0
      .label "Digimax filter" }
   .result {
      .class Image }
}

/*--- end of file:  camera.op ---------------------------------------------*/
```

Figure 3: An image capture operation.

Argument	Description
range	an integer from a given range
set	an item from a given set
flag	a boolean value
text	a text item
point	a point in the window
region	a region in the window
object	a handle to the object

Table 1: NAAG operation argument types.


camera capture

Action: preview | capture
Duration/Delay (sec) 0 – 60
Interlaced video: off | on
Image width 128 – 512
Image height 128 – 512
x offset 0 – 63
y offset 0 – 63
Digimax offset 0 – 255
Digimax gain (db): -4 | -2 | 0 | 2 | 4 | 6 | 8 | 10
Digimax filter: none | 2 | 3 | 4

Figure 4: *camera* dialog box generated by NAAG.

```
/*--- file:  image.mn  ------------------------------------------------------*/

.menu {
   .class Image
   .label "Image operations"
   .help "This menu contains a number of image processing operations"

   .menu {
      .class Image
      .label "Transformations"
      .help "This menu contains operations for transforming image data"

#     include "flip.mn"
#     include "rotate.mn"
      .menu {
         .class Image
         .label "Rescale"
         .help "Operations for the rescaling of images"

#        include <image/zoom.op>
#        include <image/munch.op>
#        include <image/resample.op>
         .stub "iscale"
      }
   }

#  include "statistics.mn"
#  include "window.mn"
#  include "lowlevel.mn"
}

/*--- end of file:  image.mn  -----------------------------------------------*/
```

Figure 5: A NAAG menu which contains a number of image processing operations.

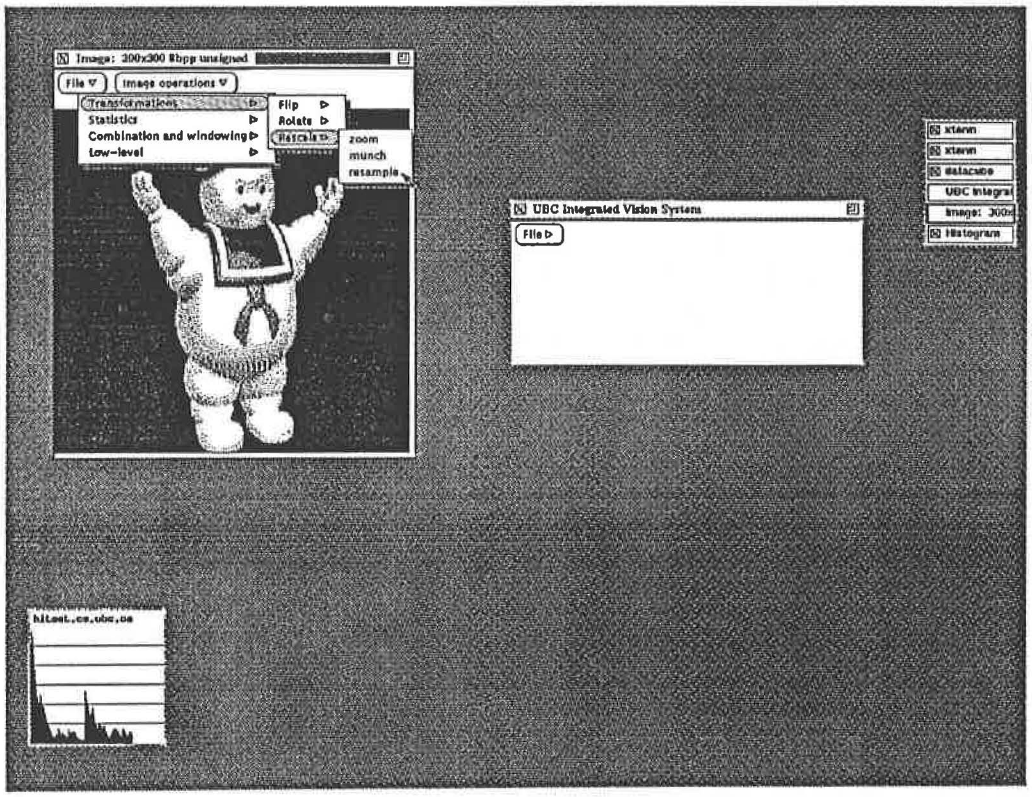


Figure 6: A partial menu tree attached to an image object.

```
/*--- file:  vision.n  --------------------------------------------------*/

.application vision

   .label "UBC Integrated Vision System"
   .help "This is a sample image processing application for computer vision"

   .classes
#     include <hist/hist.cl>
#     include <image/image.cl>

   .menus
#     include "file.mn"
#     include "image.mn"

/*--- end of file:  vision.n  -------------------------------------------*/
```

Figure 7: A NAAG main program.

```
/*--- file:  image.cl  --------------------------------------------------*/

.define_class Image {
   .class .none                      /* not a subclass of an existing class */
   .includes { "image/image.h" }     /* C typedef for Image data structure */
   .libraries { "-limg" }            /* object library which contains routines */
   .display_routine DisplayImage     /* NAAG display routine */
   .destroy_routine DestroyImage     /* NAAG destroy routine */
}

/*--- end of file:  image.cl  -------------------------------------------*/
```

Figure 8: A NAAG class definition.

simple, but very powerful.

Now that we have menu trees, we can look at the basic NAAG application. It consists of a section for defining classes and a section for defining menus. Figure 7 contains a sample NAAG application main program. What is left to describe is the way one defines a class using NAAG. Figure 8 shows the class definition for the *Image* class used in the preceding examples. The **display** routine is a user-supplied function which is invoked by NAAG to display an object. It simply calls some sequence of NAAG drawing primitives to render the given object. The NAAG primitives are listed in Table 2. Since these primitives are device and interface independent, a single display routine will suffice for all target environments. The **destroy** routine is a user-supplied function which is invoked by NAAG to release any resources held by the object (typically, it frees memory allocated to store the object's data structure).

A few quick words about the primitives. The drawing primitives function in the coordinate system defined by the user when creating a new window. This eliminates scaling of coordinate

```
int n_colour(int red, int green, int blue);
int n_draw_line(int x1, int y1, int x2, int y2);
int n_draw_point(int x, int y);
int n_draw_rectangle(int x1, int y1, int x2, int y2);
int n_draw_text(int x, int y, char *s);
int n_draw_image(int x, int y, int width, int height, int depth, char *data);
int n_line_width(int width);
int n_locator(void (*event_proc)(int event, int x, int y));
int n_message_error(char *s);
int n_message_plain(char *s);
int n_window_clear();
int n_window_create(int x1, int y1, int x2, int y2, char *title);
int n_window_message(char *s);
int n_window_restore();
int n_window_save();
int n_window_title(char *title);
```

Table 2: NAAG primitives

values as NAAG maintains the correct aspect ratio when the drawing windows are resized, and is an example of one more way that the programmer's task can be simplified. The **n_window_save** primitive can be used to record the state of the window while **n_window_restore** restores the window to the previously saved state. NAAG maintains an internal display list of items drawn and treats image data in the same manner as graphical items such as lines. Images are automatically resampled when a window is resized, and gray-level images are automatically dithered if they are being displayed on a bi-level display. This makes it possible to easily combine image and graphics elements since they are treated uniformly and share a common coordinate system. The **n_locator** routine lets the programmer register a routine for processing mouse events. Mouse events have been reduced to just four types: button up, button down, move, and drag. The locator primitive in conjunction with the screen save/restore primitives can be used to implement rubber-banding, dragging, and graphical undo operations with just a few lines of high-level code. For example, Figure 9 shows a C function which implements rubber-banding of rectangular regions.

Now that we have a complete program, we can create the entire application with a single command. NAAG automatically generates the complete application and any related files. Thus the programmer may alter the menu tree by moving, adding, or removing components and obtain the modified application with a single recompilation.

3.3 Implementation notes

The current version of NAAG produces X Windows applications for Sun 3 and Sun 4 architectures. One of the most important features of X is its device-independent architecture which allows programmers to develop portable graphical user interfaces. This was a key consideration in

```
void RubberBandRegion(int event, int x, int y)
{
   static int u, v;

   switch(event) {

      case N_LOCATOR_DOWN :
         n_window_save();
         u = x; v = y;
         break;

      case N_LOCATOR_DRAG :
         n_window_restore();
         n_draw_rectangle(u, v, x, y);
         break;

      case N_LOCATOR_UP :
         n_window_restore();
         /* (u,v) - (x,y) is the chosen region    */
         /* ... code to do something with it ...  */
         break;

      default :
         break;
   }
}
```

Figure 9: A rectangular region rubber-banding routine.

choosing a target environment since it enabled a single target to work on a number of platforms. See [SG86, Nye89, You89] for a description of the X Windows System, the Xlib programming interface, and the Xt Intrinsics layer of the X toolkit.

To simplify the generation of X-based applications we chose to use the XView Open Look Toolkit [Hel89]. XView enables the programmer to build interactive applications without having to know many of the details of the underlying window system–its object-oriented programmer's interface is simpler to learn and easier to use than Xlib.

A number of UNIX tools are used in constructing NAAG [KR]. The C preprocessor *cpp* [KR78] is used as a first pass filter and allows the programmer to take advantage of the macro preprocessor features such as the file include facility when preparing NAAG source code. *lex* [LS] and *yacc* [Joh] are used to implement the lexical analysis and parse phases of the compiler and to generate a parse tree. These phases of the compiler are independent of the target environment. Code generation is performed by modules written in C, which create the following outputs:

- C program which calls a number of functions supplied in the NAAG run-time support libraries to build an XView interface.
- A Makefile for compiling the generated C code and linking in all external object modules and libraries.
- Help information to be used at run-time.
- A *man* style manual page.

The *make* utility is invoked by NAAG to compile the generated source code and link in all associated object modules and libraries including the NAAG generic and target-specific run-time support modules. Figure 10 gives a graphical representation of the relationships between the various components.

3.4 Relationship to existing tools

Many object-oriented tools such as Interviews [LVC89] allow programmers to define user interfaces in a disciplined fashion and often at a higher level than the underlying windowing system which they employ. These tools are most appropriate for designing unique or custom interfaces rather than creating relatively standard interactive applications. Although they typically simplify a number of programming tasks, they still require a high degree of sophistication from the programmer. Furthermore, they typically do not provide any facilities to simplify other aspects of software development, but focus entirely on generating the interface. Direct manipulation tools such as Peridot [Mye87], which allow the designer to interactively describe the user interface by giving examples, are even less desirable in our context as we wish to automate the task of application generation. Similarly, tools which attempt to combine programming and direct manipulation such as [ABM89], are not suitable for automating application generation.

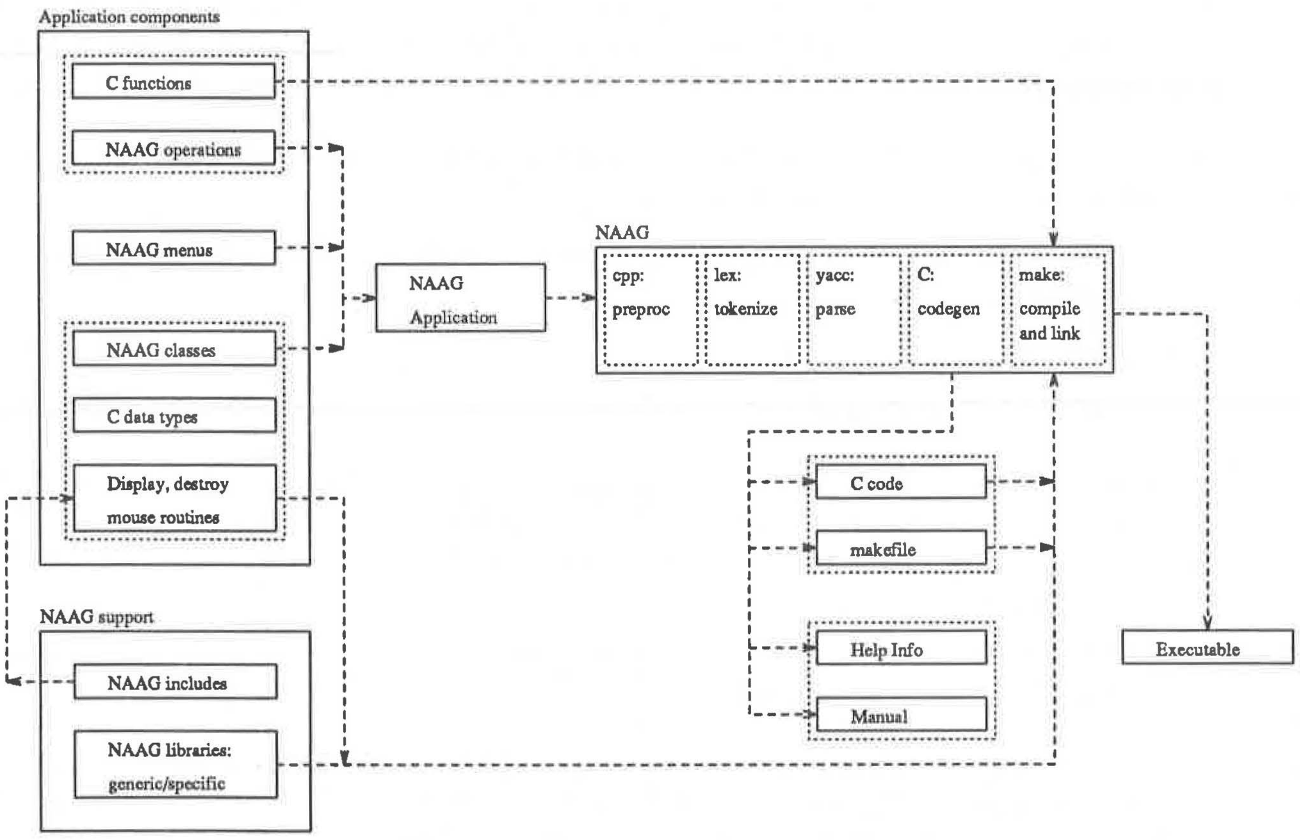


Figure 10: NAAG framework for application development.

One of the first attempts to describe the interface in terms of its semantics rather than its syntax was given in [Bla77]. Our approach is similar to that of Syngraph [Ols83, OD83], Mike [Ols86], and IDL [Fol87, FKKM89]. We have described some of our goals and motivations earlier. Many of these ideas came from earlier work in attempting to create a framework for developing portable interactive applications for computer vision [Noi89]. Our approach differs from that of Mike or IDL in the following respect.

1. NAAG is intended to be a tool for programmers:
   - It enables programmers to create interactive applications with minimal effort. NAAG employs a number of existing tools to reduce the programming effort and emphasizes creation of complete interactive applications rather than of just the interface.
   - A certain level of computer-literacy is assumed. Specifically, arguments that language-based interface tools are harder to learn and use than tools which allow the interface to be defined by direct-manipulation, are not relevant, as programmers are accustomed to using formal languages to specify computational tasks. Note, however, that NAAG can be used to create interactive graphical editors for manipulating NAAG programs in a manner similar to visual programming [Mye86].
2. NAAG generates applications which use standard interactive techniques:
   - It is often worthwhile to sacrifice flexibility for a substantial gain in ease of use. A large class of applications have relatively modest requirements in terms of the kinds of inputs they must acquire and the outputs they must display.
   - Both the applications it generates and NAAG itself are very portable. NAAG employs existing technology and can be modified to generate semantically equivalent interfaces for different targets. This allows a single specification to be compiled to obtain applications which run on several machines and employ a variety of interfaces and interactive styles.
3. The integrated environment allows applications to be defined and implemented very quickly:
   - NAAG is simple and fast enough to be used as a rapid-prototyping tool. For example, a simple drawing program employing standard graphics procedures and mouse interactions was defined and implemented in under three hours. This application, as every NAAG program, is fully portable, as it does not rely on any mechanisms outside of NAAG.
   - Tasks such as compiling and linking existing and generated modules to create the complete application are performed automatically and help to reduce the programming effort.
   - As in the TIGER system's *common development environment* at Boeing [Kas85], NAAG fosters a consistent, unified approach to application development. The programmer's view of the application is simple and remains consistent across environments. Furthermore, since software written in NAAG must adhere to a number of conventions (modular design, consistent treatment of the roles of inputs and outputs,

interface-independent operations, etc), it may be easily incorporated in other systems. As the underlying operations (semantics) have to be realized in every case, the total investment in NAAG consists of a number of very simple NAAG specifications. This amounts to a very small proportion of the total effort and allows the programmer to focus on the semantics of the application rather than developing code to interact with the user.

4. NAAG contains more knowledge of the target environment and is able to produce optimized code:

   - Operating system and programming tools (macro preprocessor, compilers, make utility, object modules and libraries, etc).
   - Application programming language.
   - Interface toolkit or UIMS used to realize the interface. Tasks such as communication, resource management, window management, and graphics are performed automatically and in the most efficient manner.
   - Information on how to perform various interactive tasks such as obtaining different kinds of information from the user, error handling, and providing help, is built into NAAG.

4 Conclusions

Interface tools simplify the design and implementation of user interfaces. Sophisticated general-purpose tools are difficult to use and may not be appropriate for many standard applications. Tools which generate interactive applications from a high-level description of the application's semantics require minimum effort to use and provide greater opportunity to employ existing technology.

NAAG, a tool we have developed at UBC, represents an advance in software engineering in several ways:

1. Enables the programmer to create sophisticated interactive applications with minimum effort.

2. Fosters a strategy which emphasizes the creation of modules which are clearly separated from the user interface and thus have greater potential for reusability.

3. Provides a simple yet powerful mechanism for defining object-oriented classes and operations on objects belonging to these classes.

4. Increases software portability. By modifying NAAG's code generation module, existing applications can be recompiled to run on different machines and to employ a variety of user interfaces and interactive styles.

5. Promotes a more consistent environment for the application developer and a more uniform interface for the user through the standardization of programming tools and interactive styles respectively.

6. Simplifies and speeds up application creation resulting in applications which are more efficient and contain fewer bugs than hand-coded systems.

As user interfaces become more sophisticated, more powerful interface tools will have to be created. This approach provides a very simple mechanism for creating interactive applications and should be of significant practical value in the future.

Acknowledgements

This work owes much to David Lowe who has been a constant source of support and guidance. The creation of NAAG has been an evolutionary process – the feedback that I have received from David and others was invaluable. In particular, I would like to thank Michael Horsch for his willingness to listen to my numerous ideas and for providing helpful insights. I extend my thanks also to Andrew Csinger for exercising his excellent proof reading skills.

References

[ABM89] G. Avrahami, K.P. Brooks, and Brown M.H. A two-view approach to constructing user interfaces. *Computer Graphics*, 23(3):137–146, 1989.

[Bla77] J.L. Black. A general purpose dialogue processor. In *Proceedings of the National Computer Conference*, pages 397–408, New York, NY, 1977. ACM.

[FKKM89] J. Foley, W.C. Kim, S. Kovacevic, and K. Murray. Defining interfaces at a high level of abstraction. *IEEE Software*, 6(1):25–32, January 1989.

[Fol87] J. Foley. Transformations on a formal specification of user-computer interfaces. *Computer Graphics*, 21(2):109–113, 1987.

[Hel89] D. Heller. *XView Programming Manual.* O'Reilly and Associates, Sebastopol, CA, 1989.

[HH89] H.R. Hartson and D. Hix. Human-computer interface development: Concepts and systems for its management. *ACM Computing Surveys*, 21(1):5–92, 1989.

[Joh] S.C. Johnson. *YACC–Yet Another Compiler-Compiler.* Bell Laboratories, Murray Hill, NJ.

[Kas85] D.J. Kasik. An architecture for graphics application development. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 365–371, New York, NY, 1985. IEEE.

[KR] B.W. Kernighan and D.M. Ritchie. *UNIX Programming.* Prentice-Hall, Englewood Cliffs, NJ.

[KR78] B.W. Kernighan and D.M. Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, NJ, 1978.

[LS] M.E. Lesk and E. Schmidt. *Lex - A Lexical Analyzer Generator.* Prentice-Hall, Englewood Cliffs, NJ.

[LVC89] M.A. Linton, J.M. Vlissides, and P.R. Calder. Composing user interfaces with interviews. *IEEE Computer*, pages 8–22, 1989.

[Mye86] B.A. Myers. Visual programming, programming by example, and program visualization; a taxonomy. In *Proceedings of the SIGCHI'86: Human Factors in Computing Systems*, Boston, MA, April 13–17, 1986. ACM.

[Mye87] B.A. Myers. Creating interaction techniques by demonstration. *IEEE Computer Graphics and Applications*, pages 51–60, September 1987.

[Mye89] B.A. Myers. User-interface tools: Introduction and survey. *IEEE Software*, 6(1):15–23, January 1989.

[Noi89] E.G. Noik. A user interface independent computer vision system. University of British Columbia, 1989.

[Noi90] E.G. Noik. Naag–not another application generator user's guide. University of British Columbia, 1990.

[Nye89] A. Nye. *Xlib Programming Manual.* O'Reilly and Associates, Sebastopol, CA, 1989.

[OD83] D.R. Jr. Olsen and E.P. Dempsey. Syngraph: A graphical user interface generator. *Computer Graphics*, 17(3):43–50, 1983.

[Ols83] D.R. Jr. Olsen. Automatic generation of interactive systems. *Computer Graphics*, 17(6):53–57, 1983.

[Ols86] D.R. Jr. Olsen. Mike: The menu interaction kontrol environment. *ACM Transactions on Graphics*, 5(4):318–344, 1986.

[SG86] R.W. Scheifler and J. Gettys. The x window system. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.

[You89] D.A. Young. *X Window Systems Programming and Applications with Xt.* Prentice Hall, Englewood Cliffs, NJ, 1989.