An Application of Template Methodology:
Rapid Prototyping of User Interface Management Systems

Deborah A. Frincke [frincke@cs.ucdavis.edu]
Gene L. Fisher [gfisher@polyslo.CalPoly.edu]
Myla Archer [archer@cs.ucdavis.edu]
Karl Levitt [levitt@cs.ucdavis.edu]

Division of Computer Science
University of California, Davis

1 Introduction

Probably the most important characteristic of a user interface is its 'look and feel.' Software consumers often select the programs they purchase based upon the program's ease of use (i.e., quality of the user interface), rather than just the relative number of features each provides. Clearly, it is important to determine the appropriateness of a program's interface early in the design phase. Unfortunately, textual descriptions do not fully capture 'look and feel', since this quality is a function of the way that users interact with the interface. We can model the 'look and feel' of user interfaces via rapid prototyping. However, the behavior of a particular user interface is greatly influenced by the particular user interface management system (UIMS) upon which it is built, so it is necessary to prototype an interface with regard to a particular user interface management system. Moving an application and its user interface to a different UIMS may result in a user interface that behaves in a substantially different manner. In this paper, we describe a methodology that may be used to develop a UIMS prototype using a template. This template will assist developers not only in developing new UIMSs, but in evaluating the behavior of user interfaces when the underlying UIMS changes.

In [AFL90], we developed a general tool for the rapid prototyping of operating systems, based on an executable template operating system specification (Secure Resource Manager, or SRM). In that work, we found that a prototype of a specific operating system could be generated by extending a template operating system specification defining such features as a scheduler, processes, objects, and requests. In this paper, we have taken advantage of the similarities of operating systems and UIMSs to produce an abstract model of event-driven (secure) graphical user interface management systems, called the SGUIM (Secure Graphical User Interface Manager) template.

Originally, programmers developed user interfaces by writing code that directly manipulated computer screens and input devices. However, the current trend in graphical computer systems is to provide an underlying UIMS to handle common tasks. A UIMS may be thought of as the manager of the graphical system components that comprise a collection of user interfaces. In many ways, a UIMS is similar to an operating system: it must keep track of graphical devices, schedule responses to user interactions, and maintain a collection of graphical objects on a screen. Similarly, the user interface of an application corresponds to a user process within an operating system; like processes, they manipulate existing (graphical) objects according to an underlying program or dynamic user inputs. Different UIMS systems vary in the type of services and graphical objects that they provide to a user. For example, some UIMS systems provide functions that permit graphical objects belonging to one application to overlap those belonging to another; other UIMSs provide 'beautification' services that rearrange graphical objects according to

0-8186-3040-X/92 $3.00 © 1992 IEEE

predetermined constraints. Our model provides a template that may be used to easily create rapid prototypes of these different types of UIMS.

Although there is a proliferation of tools permitting prototyping of interfaces for individual programs (for example, Prototyper[Cos90]), this is not the case for UIMS design. Further, prototyping tools for user interfaces generally emit code; thus, they are more appropriate for use during the coding phase rather than the specification phase of an application. We have provided a collection of independent 'building blocks' that may be used to design individual user interface components. These building blocks permit the user to experiment with different interfaces without resorting to low level graphical display code. Using the SGUIM template and the 'building blocks,' we can simulate both user interfaces and their management systems.

As in the case of our previous approach, the user interface management prototype permits us to answer several important questions about its design:

- Do the graphical displays exhibit the behavior and appearance expected by the user?
- Does the UIMS handle user interactions as intended?
- Are the services provided by the UIMS sufficient to permit the designer of individual user interfaces to develop the desired applications?
- Can a particular UIMS be in any sense secure?
- What difficulties will result if a particular user interface is transported from one UIMS to another? In other words, when an application having a particular user interface is transported, what changes will have to be made, and will the interface behave (with respect to both visual operations and security) as it did previously under the new UIMS? Will the UIMS behave the same when a new user interface is added?

To illustrate our approach, we have instantiated the template to produce prototypes of the high level portions of two user interface management systems: the X Window System[SG86], and the Blend UIMS[FF91].

At present, our experience indicates that the SGUIM prototype provides several significant advantages:

- Use of the SGUIM makes it is possible to separate specific concerns about a UIMS fairly easily. For example, one can easily isolate the portion of a UIMS that deals with the order in which user interactions are handled, and modify just that part without significantly changing the entire system.
- Use of the SGUIM permits a system designer to determine how portable a given interface will be. The individual user interfaces are all written in terms of Primitive Graphics (our basic graphical operations), and their appearance is determined partially by their individual definitions, and partially how those definitions are handled by the SGUIM prototype. Thus, a system designer can design an interface to work with one SGUIM prototype, and then substitute a second SGUIM prototype and see what changes will need to be made to the interface. This feature is particularly important for application designers, since many graphical applications must be designed to run under several different UIMS systems.
- Use of the SGUIM permits a system designer to perform a quick preliminary check of the suitability of a particular UIMS for a particular set of applications. By observing the functions provided by a particular SGUIM prototype, the system designer can decide whether or not that UIMS provides the functionality required to conveniently implement the applications.
- Use of the SGUIM model will help guide the system designer in harmoniously adding new features. For example, if a designer wants to add the ability to manage the arrangement of graphical views on a display, or change the order in which user interactions are handled, the SGUIM model will help localize the area that should be

changed in addition to permitting the designer to quickly observe the results of that change. The presence of the prototype will also permit the designer to check old programs to see whether they are adversely affected by the proposed changes without waiting until they are implemented.

- SGUIM permits certain 'look and feel' attributes of a user interface to be evaluated early in the development cycle. For example, one can use the prototype to see how difficult it is to perform certain functions; with a sufficiently detailed specification, one can determine whether or not a graphical view conveys the information desired.
- As with the SRM, SGUIM promotes incremental development of UIMS by permitting both detailed and high-level specifications to be mixed. The user need only elaborate template specifications for areas that are of interest at a particular time.

Our current experience with SGUIM leads us to believe that the high-level specification of substantial portions of existing UIMS systems is surprisingly easy, taking only a few weeks. The SGUIM template itself took only a short time to derive from the original SRM system. Further, SGUIM prototype specifications are far shorter than the completed UIMS to which they correspond, since the SGUIM prototype contains only the abstract requirements of the system, rather than all of the implementation details.

2 Methodology

As indicated in the introduction, we propose to do early testing of user interface management systems by rapid prototyping from a template. This decision was made for three reasons. First, there is significant commonality among different user interface management systems. As shown in Figure 2, many UIMS may be considered resource managers, where user interactions and program operations generate requests to use graphical resources. Such resource types as processes, graphical pictures and widgets[1], events[2], and event handlers are common to a great many user interface management systems. Second, it has been our experience throughout the development of sample user interface management systems that it is easy for the designer to get caught up in the details of implementation rather than focusing on the design issues. By introducing a formal definition of the UIMS and abstracting out the actual screen and device implementation details, the developer can concentrate on the interesting design issues rather than clever programming techniques. Third, the template design methodology gives designers an easy way to directly compare UIMS features.

A template in general serves several purposes. First, it organizes one's thinking about the workings of an (almost) arbitrary system by factoring the system into its components that play conceptually different roles, and relieving one of having to specify certain high-level operations. Second, when completed (and in many cases, only partially completed) for a specific application system, it can be used as a rapid prototype of that system (or part of it). Third, it supports incremental development of systems, by permitting the developer to elaborate areas of interest in exactly the amount of detail desired, rather than fully developing all areas at once. Finally, it can be used to organize proofs of properties of any application system, by helping to isolate for each property just those aspects of the system that are relevant, and thus potentially simplifying actual proof obligations.

As mentioned earlier, this paper describes a template for the rapid prototyping of user interface management systems known as SGUIM. SGUIM is composed of the components shown in Figure 2. This model has been specialized to correspond to event-based user interface management systems. SGUIMs are made up of the following components:

[1] Within this paper, widgets may be considered to be "executing pictures", just as processes within operating systems may be considered "executing programs"

[2] ex: mouse clicks, keyboard operations

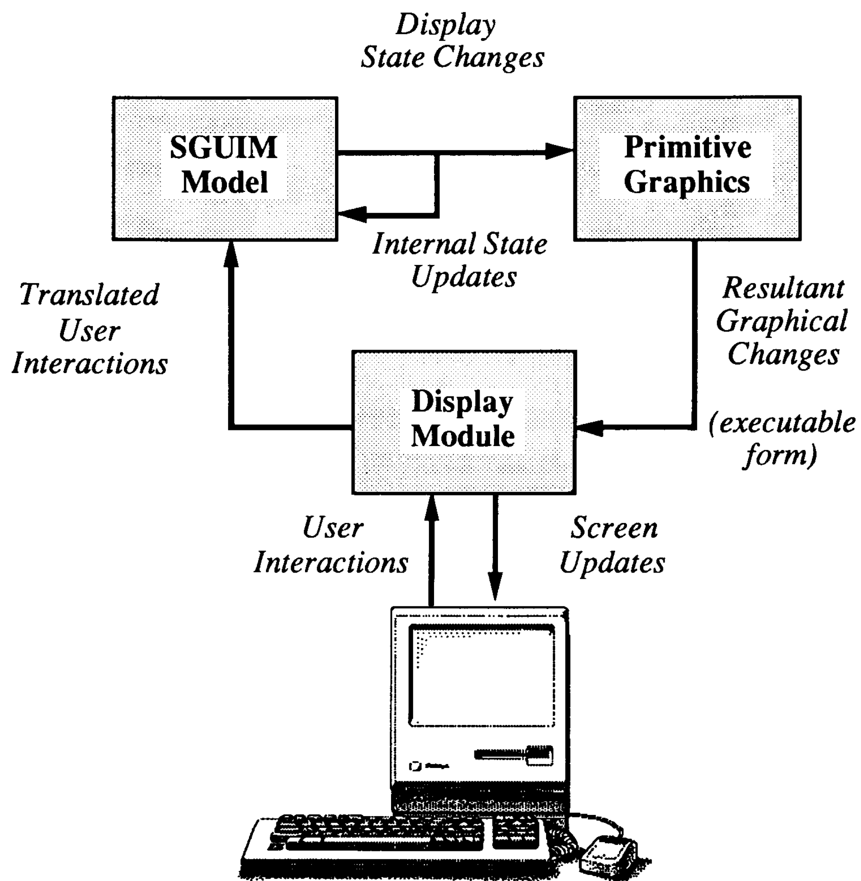


Figure 1: Prototyping system for user interface managers

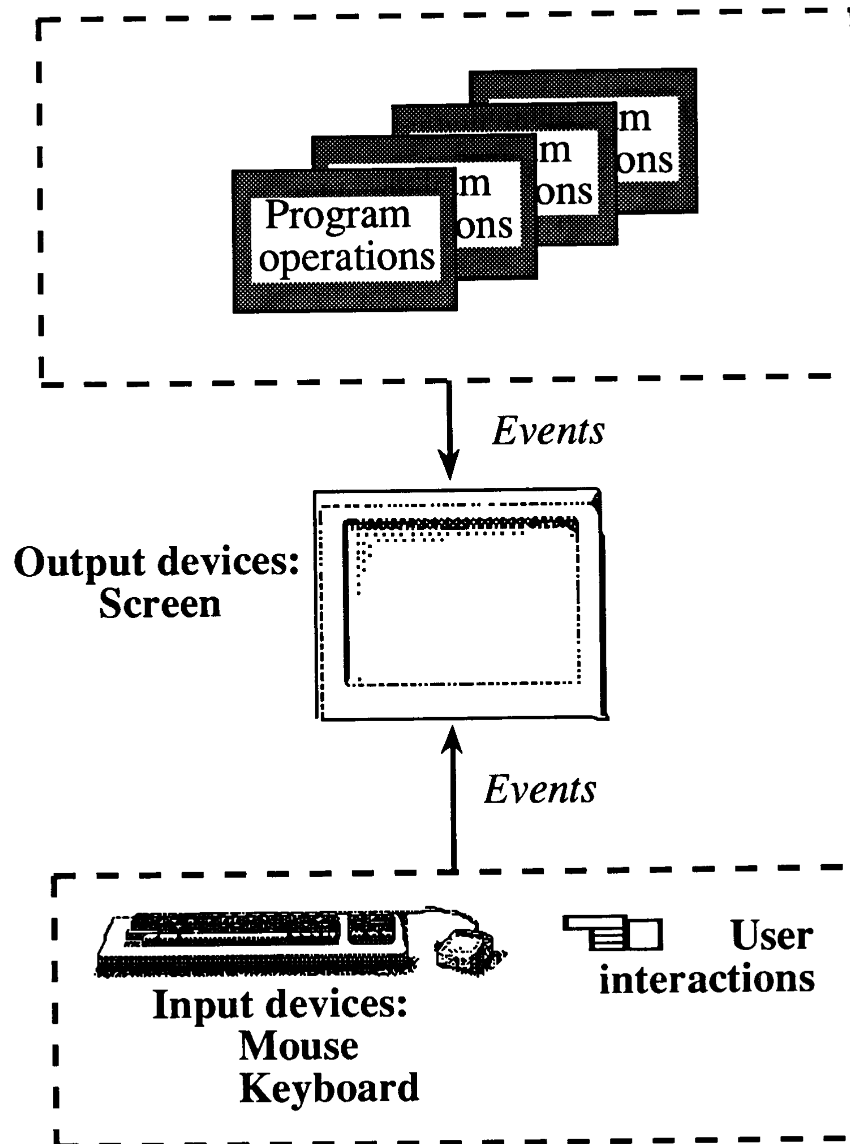


Figure 2: What is a UIMS?

- The *State* of a SGUIM is composed of five parts:
  - *ApplicationSet*, which contains all of the applications that are communicating with the SGUIM *EventHandler*. An application consists of the underlying user program along with its interface.
  - *ObjectSet*, which contains all of the objects that are potentially viewable within an application. Note that some of these objects may be created or destroyed dynamically as a result of operations performed by the user program or interface.
  - *ViewSet*, which contains all of the views that exist within the SGUIM. In the simplest case, each view will correspond to an object within an application. Views may be visible, invisible, or iconified; they may contain information beyond that found within a corresponding object. It is possible for a view to become disconnected from an object, or to correspond to more than one object. Among other components, Views inform the EventHandler of the operations which they can handle. Views are themselves hierarchical, since they may be made up of many view components.
  - *EventList*, which contains all of the events that occur within a system. Events may be generated by a user (e.g., mouse click) or by an application. Events are added to the list based on the EventHandler's polling of devices within the DeviceSet.
  - *History*, which contains a list of all the events that have been generated within the system.
  - *DeviceSet*, all of the physical devices within the state. For example, the mouse, the keyboard, the screen, etc.
- *SGUIMopSet*, operations that are performed within an SGUIM. SGUIM operations, unlike the SRM operations, can be added or modified dynamically as views are added to the state view state. SGUIM operations are associated with Views.
- *EventHandler*, a mechanism whereby events are translated into actions. The EventHandler corresponds to the SRM scheduler and is responsible for selecting the next event to be handled and ensuring that the appropriate SGUIMop/View handles it.
- *Interp*, which may be used to modify events on the event handler list. Among other uses, Interp determines when an event is 'stolen'—applications/views can 'grab' events even when they appear to be within the area that 'belongs' to another application/view.
- *SecPol* defines the security policy of the system; for example, which applications/users can manipulate particular views or devices.

2.1 Details of the SGUIM template

2.1.1 OBJ3

The prototype system developed to investigate the ideas presented in this work was implemented using Goguen's equational programming environment, OBJ3 [GM82]. OBJ3 is based upon order sorted equational logic, and provides executability via rewrite rules. As in other algebraic specification systems, OBJ3 objects are defined by providing a type signature for their operations and a collection of equations describing their behavior. These equations serve as rewrite rules, thus providing executability[3]. Objects are generic, in the sense that they may be parameterized.

Besides parameterized objects, OBJ3 includes the concept of ***theories***. OBJ3 theories define properties of objects. For example, a theory may require that an object provide a particular operator. Theories are most often used to

[3]Rewrite rules are equations where the expression on the left hand side may be replaced by the expression on the right hand side. Execution proceeds until such replacements are impossible.

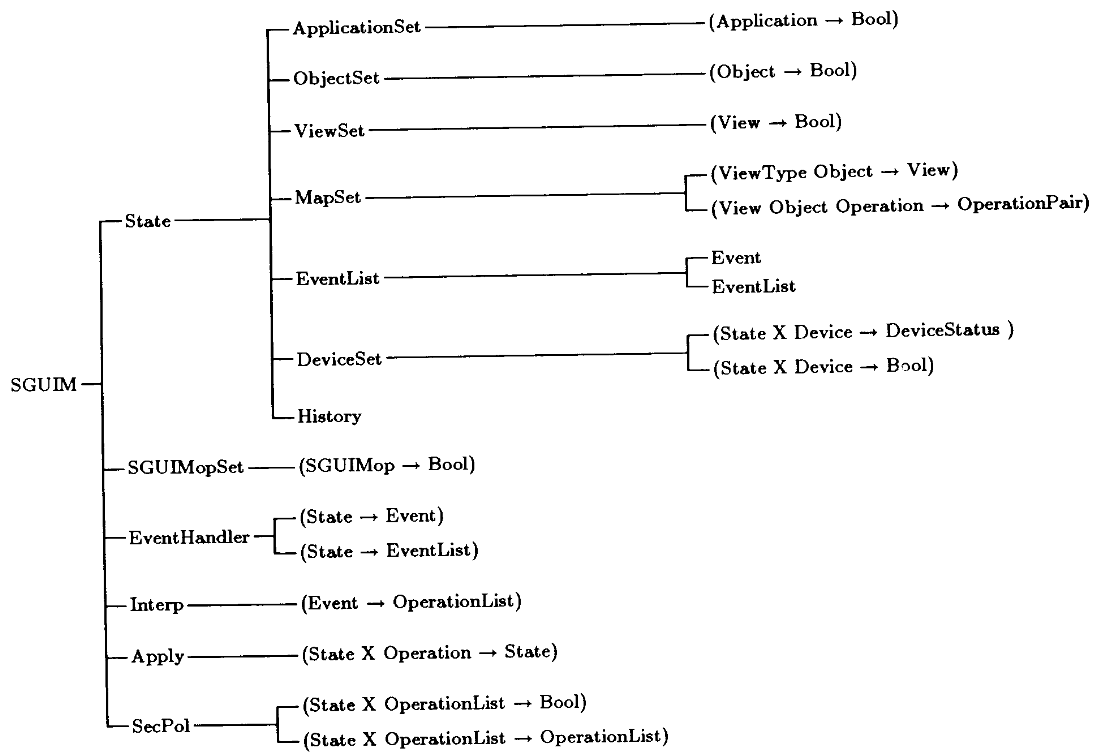


Figure 3: Overview of the SGUIM template specification

```
OBJ> red drawItemList('a 'b 'c nilElementList) .

reduce in DISPLAY-LIST-ELT :
              drawItemList('a ('b ('c nilElementList)))

rewrites: 135

result GraphicsObjectList:
(drawBox(Point(50,30),Point(55,20))
 (drawText(Point(52,20),
   cellDisplay('c)) ))

((drawBox( Point(50,40 ),Point(55,30))
 (drawText(Point(52,35 ),
   cellDisplay('b)) ))

((drawBox(Point(50,50),Point(55, 40))
 (drawText(Point(52,45 ),
   cellDisplay('a)) ))
   ))

50                55
<---------------->

 ---------------- 50
|       a        |
 ---------------- 40
|       b        |
 ---------------- 30
|       c        |
 ---------------- 20
```

Figure 4: Primitive graphics commands used for displaying an item list.

place requirements on an object's parameters. Views are used to bind objects to object parameters. This is done via the ***view***[4], which specifies the correspondence between the parameter theory and the actual object's components. This ensures that instantiated parameters satisfy assertions about the generic parameters. For example, consider a generic ordered list object that is parameterized by the type of object to be contained in the list. In order to maintain "orderedness", it is necessary to ensure that the items in the list have some sort of ordering operator, such as <=. This is done by declaring that the `item` parameter of the generic ordered list object has a <= operation. To create a particular type of ordered list, it is necessary to include an ordering operation within the definition of the object to be kept in the list. A view is then used to map that ordering operation to the <= operation required by the parameter theory.

As in the SRM, we develop the template by searching for the most natural decomposition of a UIMS. Rather than Requests, Files/Objects, and Processes, we observe Events, Views, and Applications. Each of these may be described as a data type, or OBJ3 object (Appendix A contains several sample specifications).

As for SRM, there are four kinds of data types within SGUIM. These are Kind 1 (invariant across all applications), Kind 2 (special tailored member), Kind 3 (system-specific representation) and Kind 4 (only for specific applications). These ***kinds*** serve as a guide for the designer when extending the SGUIM template to create a prototype of a specific system. For example, a SGUIM state is required to contain a set of devices. These devices are Kind 3; though they are required to contain a unique identifier and operations to initialize and reset themselves, the actual configuration of the device and the form of the operations is left to the designer. The DeviceSet itself is Kind 2; the designer may add members and operations, but the majority of the specification is invariant.

2.2 Primitive Graphics

Graphical views of application objects are represented by primitive graphics functions, and can build up small prototypes without further implementation. Primitive graphical objects are the basic graphical building blocks provided by the system. They include object (box, text, circle, ellipse, line, scroc[5]) and operations (move, rotate, flip, scale).

An example of the primitive graphics commands produced by the prototype to display an object is shown in Figure 4. In this Figure, an item list is displayed as a downgrowing list with origin(50, 55). The stream of commands is passed on to the Display Model, which generates the actual graphics calls used to update the system.

2.3 Display Model

The Display Model is used to manage the physical computer system. Without the Display Model, it would not be possible to actually view the results of user interactions; one would be forced to examine textual specifications instead. The Display Model is implemented in using the C++ and InterViews [LVC89]. Essentially, the Display Model waits for the user to interact with the physical system in some way (for example, by moving the mouse). These interactions are then transformed into their corresponding event descriptions and passed on to the SGUIM prototype. The SGUIM prototype determines how the display should change, and updates its internal state. These changes are then passed on to the primitive graphics interpreter, which translates them into low level graphical commands. These graphical commands are received by the Display Model, which updates the physical screen to correspond.

[4] The OBJ3 use of the term *view* is different from the term as used in referring to graphical views.

[5] A scroc is a simple, convex, rectifiable, open curve

3 Comparing UIMS and Adding New Features

When comparing UIMS, there are many questions that a developer may ask. It is difficult to answer these questions when the user is comparing actual system, since the design is obscured by the details. For example, one might want to determine whether a UIMS will always handle a particular set of user interactions sequentially. This is extremely difficult to answer for most UIMS, since there may be literally thousands of lines of code interspersed within the full UIMS system that can affect the order in which user interactions are interpreted. The question is easier within SGUIM, since one can first prove that only certain segments of the specification can potentially affect observation and handling of user interactions, and then examine these sections (possibly using a verification system) to see how the interactions are handled.

Use of the SGUIM template makes it easy to add or modify UIMS features, both by making the portion of SGUIM that is modified easy to determine, and by permitting the developer to see the result of the changes without writing a great deal of low-level code. Suppose that it was desirable to add layout rules to a UIMS, where the layout rules defined ways in which graphical objects could appear on the screen. For example, one might want to add a grid, and force windows and other objects to be aligned upon the grid. Figure 5 shows two ways in which the SGUIM template could be modified to add such a feature.

- Modify SGUIM so that operations are modified to comply with the layout rules when they are schedule For example, a user might request that a particular window be moved to a new location. When the SGUIM scheduler selects that request, it is modified so that the new location complies with the layout rules before it is executed.
- Modify SGUIM so that the display is modified to comply with layout rules after each change. For example, the window movement operation is performed as specified, and then SGUIM checks the display *as a whole* and modifies it as necessary.

Clearly, these two ways of handling layout control will result in interfaces that respond to user interactions in different ways.

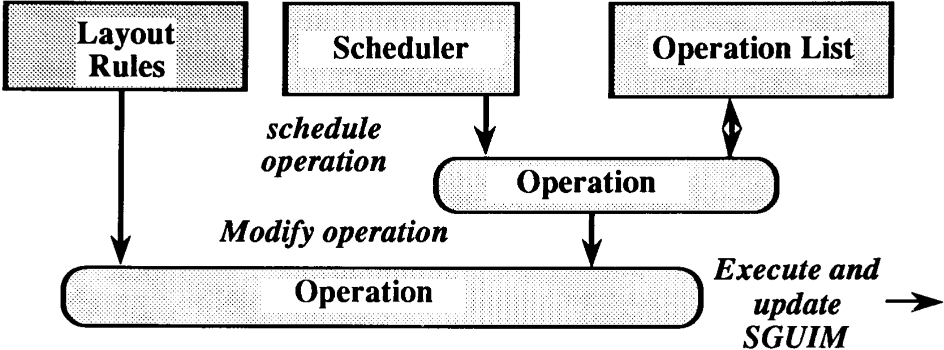
4 Results and Future Directions

In developing the Blend and X prototypes within SGUIM, we had several encouraging results. As mentioned in the introduction, development of new UIMS systems was relatively quick, and modifications were easy to make. The template style does appear to be flexible enough to handle a large class of UIMS systems. However, development at present is limited by the host machine's memory size and CPU speed.

Sparc I, 16 Meg		
	X11	Blend
length	60 pgs	92 pgs
memory	18M (8-11 resident)	20M (8-12 resident)
load time (cpu)	28 min	55 min
1st op (cpu)		43sec
2nd op		19sec

There are several projects currently underway that are intended to improve SGUIM's effectiveness. First, writing specifications of graphical objects is nearly as difficult as writing code that implements these objects (though more portable). To reduce the difficulty of this task, we are presently investigating the possibility of using a graphical programming tool (DEMO [WF91]) to generate the bulk of these graphical specifications. Second, SGUIM's execution

Change operations that modify view appearance or location so that they obey the layout rules:



Perform the operation as specified, then rearrange entire set of views to conform to layout rules

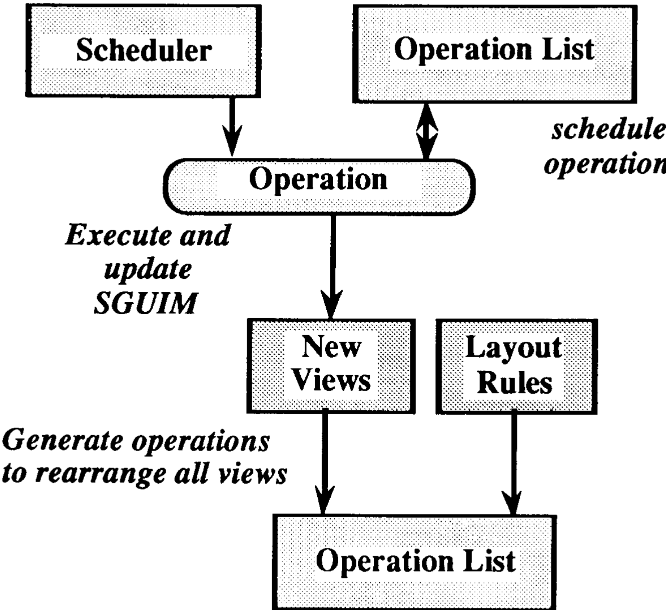


Figure 5: Adding a beautifier by modifying the request

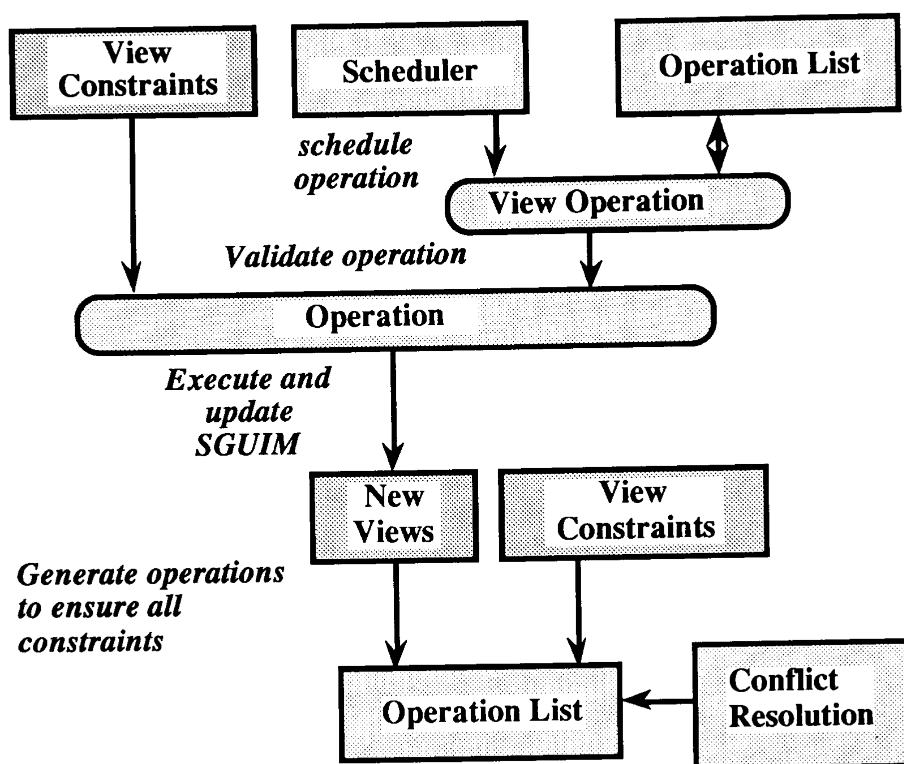


Figure 6: Adding Constraints

speed can be greatly improved by re-implementing OBJ structures directly in Lisp for efficiency. Third, describing true distributed systems via SGUIM is difficult, due to the "left to right" nature of OBJ's execution. The possibility of using a version of OBJ that has been developed with the idea of handling concurrent systems (OBJC [Buf90]) is being examined. Finally, SGUIM has only been used to describe event-based UIMS systems. To further evaluate SGUIM's generality, we intend to develop prototypes for other types of UIMS systems, particular constraint-based systems. Figure 6 shows a high-level design of such a system.

References

[AFL90] M. Archer, D. A. Frincke, and K. Levitt. A template for rapid prototyping of operating systems. *International Workshop on Rapid System Prototyping*, June 4-7 1990.

[Buf90] J. Buffenbarger. *Equational Specification and Verification of Concurrent Systems*. PhD thesis, University of California, Davis, December 1990.

[Cos90] G. R. Cossey. *Prototyper V3.0*. SmethersBarnes, Inc., 1990.

[FF91] G. Fisher and D. Frincke. Formal specification and verification of graphical user interfaces. *24th Hawaii International Conference on system Sciences*, January 1991.

[GM82] J. A. Goguen and J. Meseguer. Rapid prototyping in the OBJ executable specification language. *ACM SIGSOFT Software Engineering Notes*, 7(5):75–84, December 1982.

[LVC89] M. A. Linton, J. M. Vlissides, and P. R. Calder. Composing user interfaces with InterViews. *IEEE Computer*, 22(2):8–22, 1989.

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

[WF91] D. Wolber and G. Fisher. Developing user interfaces by stimulus response demonstration. *COMPSAC*, September 1991.

A Portions of the SGUIM specification as instantiated for Blend

This section of the appendix contains a representative sample of the SGUIM specification as instantiated for Blend. Figure 7 defines the graphical views of the system. Figure 8 describes one of the physical devices attached to the system. Figure 9 describes the top level of the Blend instantiation.

A.1 Views

Within SGUIM, the VIEW specification defines the graphical objects displayed by the system. As shown in Figure 7, views have among other attributes unique identifier, a location, a corresponding application object, a type, and a list of handles. Handles define the "hot spot" of the view, permitting user actions to be handled differently depending upon the location of devices (such as mouse location) when the action takes place.

There are many operations that may be performed upon Blend views. As one might expect, views may be resized and moved (**scaleView, moveView**). In addition, actions that take place within the physical area corresponding the a view can be matched against the view's list of appropriate reponses.

A.2 Mouse

The MOUSE specification is included to illustrate the portion of the SGUIM template that deals with devices. As shown in Figure 8, defining a device requires that (1) the basic type specification be created (here, obj MOUSE), and (2) the operations that return the device's status, reset/initialize the devices, etc. must be defined.

A.3 SGUIM

The SGUIM specification is used to define the way that SGUIM handles Blend's state components. For example, the **stepSguim** operation is defined so that the following sequence of operations occurs:

- Select an event from the Blend state's event list **getNextEvent**
- Determine the event that will handle the event, and the operation it uses in so doing **handleEvent**
- Check to see that this operation does not violate the system security policy
- Expand the operation to form a list of suboperations, if necessary
- Perform the event and update the State
- Add the operation to the SGUIM history set.

```
obj VIEW is
protecting HANDLELIST .

using RECORD-9PART *(
        sort Record to View,
             ...
        op _ @FIELD1 to _ @ViewId,
        op _ @FIELD2 to _ @ObjectId,
        op _ @FIELD3 to _ @UpperLeft,
        op _ @FIELD4 to _ @LowerRight,
        op _ @FIELD5 to _ @HandleList,
        op _ @FIELD6 to _ @ViewType,
        op _ @FIELD7 to _ @Data,
        op _ @FIELD8 to _ @OpMaps,
        op _ @FIELD9 to _ @ApplicationId,
             ...
) .

op makeView : Int Id Id Arg OpMapSet Id -> View .
op pointIsWithinView : View Point -> Bool .

op matchSingleEvent    : HandleList Event Point -> Int .
op matchMultipleEvent : HandleList EventList Point -> Int .
op domatchSingleEvent : HandleList Event Int Point -> Int .

*** Modification of existing view
op scaleView : View Int Int -> View .
op moveView : View Int Int -> View .
op plusIntView : View Int -> View .

***
*** Make a template given the view's id,
*** the corresponding object name, and the
*** view's type. Add in the standard handles.
***
eq makeView(Vi, Oi, Ti, Data, Omapset, Ai) =
View(Vi, Oi,  initPoint, initPoint, standardHandleList,
     Ti, Data nilList, Omapset, Ai) .


***
*** Note that (0,0) is upper left corner of screen, and
*** (MaxX, MaxY) is lower right corner.
***
eq pointIsWithinView(V, P) =
( V @UpperLeft @X <= P @X ) and
( V @UpperLeft @Y <= P @Y ) and
( V @LowerRight @X >= P @X ) and
( V @LowerRight @Y >= P @Y ) .

*** P indicates the true loc  of the View-object.
*** Two events are equiv if the loc of one event
*** occurs 'within' the specified hotbox of the other.

eq domatchSingleEvent(Hl, E, N, P) =
if Hl == nilList then -1
else
  if equivEvent(car(car(Hl) @EventList), E,
    (car(Hl) @HotBoxUleft) offset by P,
    (car(Hl) @HotBoxLright) offset by P)
  then N
  else domatchSingleEvent( cdr(Hl), E, N + 1, P)
  fi
fi .

eq matchSingleEvent(Hl, E, P ) =
       domatchSingleEvent(Hl, E, 0, P) .

***
*** View manipulation operations
***    Note that moveView may actually be used
***    to remove one of the views altogether from the visil
***    screen.
***
eq moveView(V, X, Y) =
   (V @UpperLeft equals
        (V @UpperLeft offset by Point(X, Y)))
     @LowerRight equals
        (V @LowerRight offset by Point(X, Y)) .

eq scaleView(V, X, Y) =
    if ((V @UpperLeft @X < X) and (V @UpperLeft @Y < Y))
    then  V @LowerRight equals Point(X, Y)
    else V
    fi .
 ...
endo

view VIEWV from ELEMENT to NEWOBJ[VIEW] is
  sort Element to View .
  op   undefined to undefined .
endv

make VIEWS is LIST[VIEWV]*(
    sort List to ViewSet,sort Element to View ) endm

obj VIEWSET is
protecting VIEWS .

op closestView : ViewSet Point -> View .
op selectView : ViewSet Int -> View .
op doupdateView : ViewSet View -> ViewSet .

vars Vset  : ViewSet .
vars V     : View .
vars P     : Point .
vars Name  : Int .

eq closestView(Vset, P) =
  if Vset == nilList then nullView
  else if pointIsWithinView(car(Vset), P)
       then car(Vset)
       else closestView(cdr(Vset), P)
       fi
  fi .

 ...

endo
```

Figure 7: High level definition of a View

```
obj MOUSE is

protecting INT .
protecting QIDL .
protecting POINT .

using RECORD-5PART *(
        sort Record to Mouse,
        op Record to Mouse,
               ...
        op _ @FIELD1 to _ @MouseId,
        op _ @FIELD2 to _ @Loc,
        op _ @FIELD3 to _ @Button,
        op _ @FIELD4 to _ @Double,
        op _ @FIELD5 to _ @Changed,
               ...
) .

op initMouse : -> Mouse .
op resetMouse : Mouse -> Mouse .

vars Mce : Mouse .

eq initMouse =
      Mouse(1, initPoint, -1,
            false, false ) .
eq resetMouse(Mce) =
      Mouse( Mce @MouseId, Mce @Loc, -1,
             false, false ) .
endo

obj DEVICESTATUS is
sort DeviceStatus .

protecting QIDL .
protecting MOUSE .
protecting KEYBOARD .

op deviceStatus : Mouse -> DeviceStatus .
op deviceStatus : Keyboard -> DeviceStatus .
op deviceFromStatus : DeviceStatus -> Mouse .
op deviceFromStatus : DeviceStatus -> Keyboard .

...

endo

obj DEVICE is
protecting QIDL .

protecting DEVICESTATUS .

using RECORD-4PART *(
        sort Record to Device,
        op Record to Device,
               ...
        op _ @FIELD1 to _ @DeviceId,
        op _ @FIELD2 to _ @DeviceType,
        op _ @FIELD3 to _ @DeviceStatus,
        op _ @FIELD4 to _ @ActiveUser,
               ...
        ) .
```

```
op initDevice : -> Device .
op resetDevice : Device -> Device .

vars D : Device .

eq resetDevice(D) =
       if      D @DeviceType == 'keyboard then
                 (D @ActiveUser equals 'none
                    @DeviceStatus equals
                        deviceStatus(resetKeyboard(
                            deviceFromStatus(D @DeviceStatus))))
       else if D @DeviceType == 'mouse then
                 (D @ActiveUser equals 'none
                    @DeviceStatus equals
                        deviceStatus(resetMouse(
                            deviceFromStatus(D @DeviceStatus))))
       else initDevice fi fi .

endo

view DEVICEV from ELEMENT to NEWOBJ[DEVICE] is
  sort Element to Device .
  op   undefined to undefined .
endv

make DEVICES is LIST[DEVICEV]*( sort List to DeviceSet ) endm

obj DEVICESET is
  protecting DEVICES .

  op resetDeviceSet : DeviceSet -> DeviceSet .
  op doreset : DeviceSet DeviceSet -> DeviceSet .
  op doupdateDevice : DeviceSet Device -> DeviceSet .
  op initDeviceSet : -> DeviceSet .

  vars DS1 DS2 : DeviceSet .
  vars D       : Device .

  eq initDeviceSet =
        Device('a, 'mouse, deviceStatus(initMouse), 'none)
        Device('b, 'keyboard, deviceStatus(initKeyboard), 'none)
        nilList .

  eq resetDeviceSet(DS1) = doreset(DS1, nilList) .

  eq doreset(DS1, DS2) =
        if DS1 == nilList then DS2
        else
           doreset(cdr(DS1), prepend(DS2, resetDevice(car(DS1))))
        fi .

  eq doupdateDevice(DS1, D) =
        if DS1 == nilList then DS1
        else
           if car(DS1) @DeviceId == D @DeviceId
           then D cdr(DS1)
           else car(DS1) doupdateDevice(cdr(DS1), D)
           fi
        fi .

endo
```

Figure 8: High level definition of a Mouse

```
obj SGUIM is
protecting QIDL .

protecting EVENTHANDLER .
protecting APPLY .
protecting SECPOL .
protecting HISTORYSET .
protecting SCHEDULER .

using RECORD-7PART *(
                    ...
        op _ @FIELD1 to _ @SguimId,
        op _ @FIELD2 to _ @State,
        op _ @FIELD3 to _ @EventHandler,
        op _ @FIELD4 to _ @SguimOpSet,
        op _ @FIELD5 to _ @Interp,
        op _ @FIELD6 to _ @SecPol,
        op _ @FIELD7 to _ @History,
                    ...
        ) .
*** initialization
op initSguim : -> Sguim .
*** translate one event to op
op stepSguim : Sguim -> Sguim [memo] .
*** translate device changes
op stepSguimDevices : Sguim -> Sguim [memo] .
*** add one device change
op updateDevice : Sguim Device -> Sguim [memo] .
*** if an op, exec it
op execSguim : Sguim -> Sguim [memo] .
*** add user appl. prog
op addApplication : Sguim Application -> Sguim .

vars Sg : Sguim .
vars D : Device .
vars Ap : Application .
   ...
eq stepSguim(Sg) =
  (Sg @State equals
      stepEventList(
          Sg @State @OpListPair equals
           interpOp( stepEventList(Sg @State),
            checkOperationSecurity( Sg @State,
              handleEvent(stepEventList(Sg @State),
                         getNextEvent(Sg @State))),
 'view)))

   @History equals
     append( Sg @History
      History('h, handleEvent(stepEventList(Sg @State),
                              getNextEvent(Sg @State))        )) .
```

```
*** *
*** * Simulate a device state change, such as mouse click or ke
*** * (doesn't  update the event queue)
*** *
eq updateDevice(Sg, D) =
        Sg @State equals
         ((Sg @State) @DeviceSet equals
                doupdateDevice( Sg @State @DeviceSet, D)) .


*** *
*** * Translate all device state changes to events, and
*** * reset the devices to their null state.
*** *
eq stepSguimDevices(Sg) =
        Sg @State equals pollDevice(Sg @State @DeviceSet,
                                    nilList,
                                    Sg @State) .


*** *
*** * Add an application to the SGUIM, including the mappings t
*** * objects and their pictorial representation
*** *
eq addApplication(Sg, Ap) =
        Sg @State equals
                (addViewObjsToState( Sg @State @ApplicationSet
                                append(Sg @State @Application
                                Ap)) .


*** *
*** * execOp executes an operation, and updates the oplist.
*** * Note that execOp is now actually executing a pair of
*** * corresponding operations (both _view_ and object), and
*** * schedule returns an OpListPair.
*** *      (generally performed right after a stepSguim)
*** *                      (see STATE)
*** *
eq execSguim(Sg) = Sg @State equals
                          (execOp(Sg @State,
                                schedule(Sg @State, Sg @State (

endo
```

Figure 9: Top Level SGUIM