Calhoun: The NPS Institutional Archive

DSpace Repository

Faculty and Researchers   Faculty and Researchers' Publications

1989

Graphical Support for Reducing Information Overload in Rapid Prototyping

Luqi; Barnes, Patrick D.

Naval Postgraduate School

744

NPS52-89-028

NAVAL POSTGRADUATE SCHOOL
Monterey, California

GRAPHICAL SUPPORT FOR REDUCING
INFORMATION OVERLOAD
IN
RAPID PROTOTYPING

LUQI

PATRICK D. BARNES

MARCH 1989

Prepared for:

Naval Postgraduate School
Monterey, CA 93943

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral R. C. Austin
Superintendent

H. Shull
Provost

The work reported herein was supported by the National Science Foundation, the Office of Naval Research and the Naval Postgraduate School Research Council.

Reproduction of all or part of this report is authorized.

This report was prepared by:

█████████

LUQI
Assistant Professor
of Computer Science

Reviewed by:

█████████

ROBERT B. MCGHEE
Chairman
Department of Computer Science

Released by:

█████████

KNEALE T. MARSHALL
Dean of Information
and Policy Science

Graphical Support for Reducing Information Overload in Rapid Prototyping

Luqi

Patrick D. Barnes

Computer Science Department

Naval Postgraduate School

Monterey, CA 93943

Abstract

The basic problem in rapid prototyping of software is information overload. Graphic interfaces can help by providing multiple views, where each view is limited to providing information relevant to a particular task or problem. The graphic editor under development for the Computer Aided Prototyping System (CAPS) proposes a data flow diagram based model with multiple views and automatic program generation to manage the quantity of information necessary to prototype large, real-time systems.

1. Introduction

The need to improve software operational reliability and development productivity has resulted in research aimed at tools for rapidly prototyping large real-time software systems. The Computer Aided Prototyping System (CAPS) replaces the traditional software life cycle with a two phase cycle consisting of rapid prototyping and automatic program generation [4]. The rapid prototyping technique provides the designer with a means of writing specifications and using matching reusable software components to build a prototype of the intended system. The prototype can then be used to evaluate both user's needs and system feasibility [2,5].

Although prototyping generally involves dealing with problems at a high level of abstraction, crucial decisions designers must make are still too many to be evaluated at a single level. Thus the designer can quickly be inundated with an overload of information—much the same as a decision maker in business or management. Prototyping large real-time systems requires tools for managing this information such that unnecessary detail may be hidden, and essentials may be easily assimilated at a glance. This paper discusses the importance of using graphical representations to reduce information overload and describes a graphical editor in development for CAPS.

2. Reducing Information Overload

An essential function of CAPS is to help the prototype designer focus on subsets of the decisions in a design needed to evaluate alternatives and further refine or modify the system [6]. CAPS initially provided for entering component specifications via a syntax directed editor accessed through its user interface. It could then try to match the specifications to reusable modules in the database. For complex systems, however, if the search was unsuccessful, the component had to be manually decomposed into increasingly detailed statements which resulted in the information management problem previously described. What was needed was a means of entering specifications graphically, such that decomposition would be cleaner and information hiding could be managed via a multi-layered representation.

Graphics alone cannot provide a magic solution to the problem of software complexity. In fact, they can sometimes complicate matters even further. Thus the application of graphical techniques must be coupled with a strategy for extracting a meaningful subset of available information to be effective. Development of a graphic editor, then, requires addressing the following issues. First, the graphic representation must be automatically programmable. In the case of CAPS, it must map directly to equivalent Prototype System Description Language (PSDL) representations with which CAPS can construct a prototype [2,5]. Second, the graphic representation must provide multiple system views—reducing the amount and detail of information which must be assimilated at any one time. Finally, a provision has to be made for maintaining consistency between the PSDL and graphic representations, as well as syntactic and semantic correctness of the design.

2.1 Automatic Programming

One of the most promising means of improving programmer productivity is automatic programming. That is, the automatic generation of code from software specifications rather than manual generation through several layers of language translation. The context of this paper is CAPS which is based on the PSDL prototyping language [3]. PSDL specifications may be directly used to produce an executable Ada program from reusable components. PSDL provides for specification of both control and data flow and is based on the following mathematical model:

$$G = (V, E, T(V), C(V))$$

where V is the set of vertices

E is the set of edges

T(V) is the maximum execution time (MET) associated with vertex V

C(V) is the set of control constraints associated with vertex V.

In PSDL, vertices represent operators and edges represent data streams. Operators represent system components and can map to either functions or state machines. Such components communicate with one another via data streams carrying values of a fixed abstract data type or the special PSDL type EXCEPTION. Operators are either data driven or periodic. That is, they execute either in response to the arrival of a data flow, or at a predetermined interval. Operators can also be characterized as being either composite or atomic. If an operator cannot be further decomposed into data and control flow networks, it is atomic. Edges or data streams can represent either the traditional flows, in which data is guaranteed to reach its destination, or sampled streams. Sampled streams represent continuous streams of information which may be updated and sampled at different rates.

Control constraints are used to limit an operator's behavior by specifying conditions regarding its firing (execution) or i/o processing. While control constraints specify when an operator executes, timing constraints determine its execution time, response time, and period.

The PSDL model can be mapped directly to the augmented data flow diagram [3]. In addition to data flows and transformations (operators), the PSDL model requires representa-
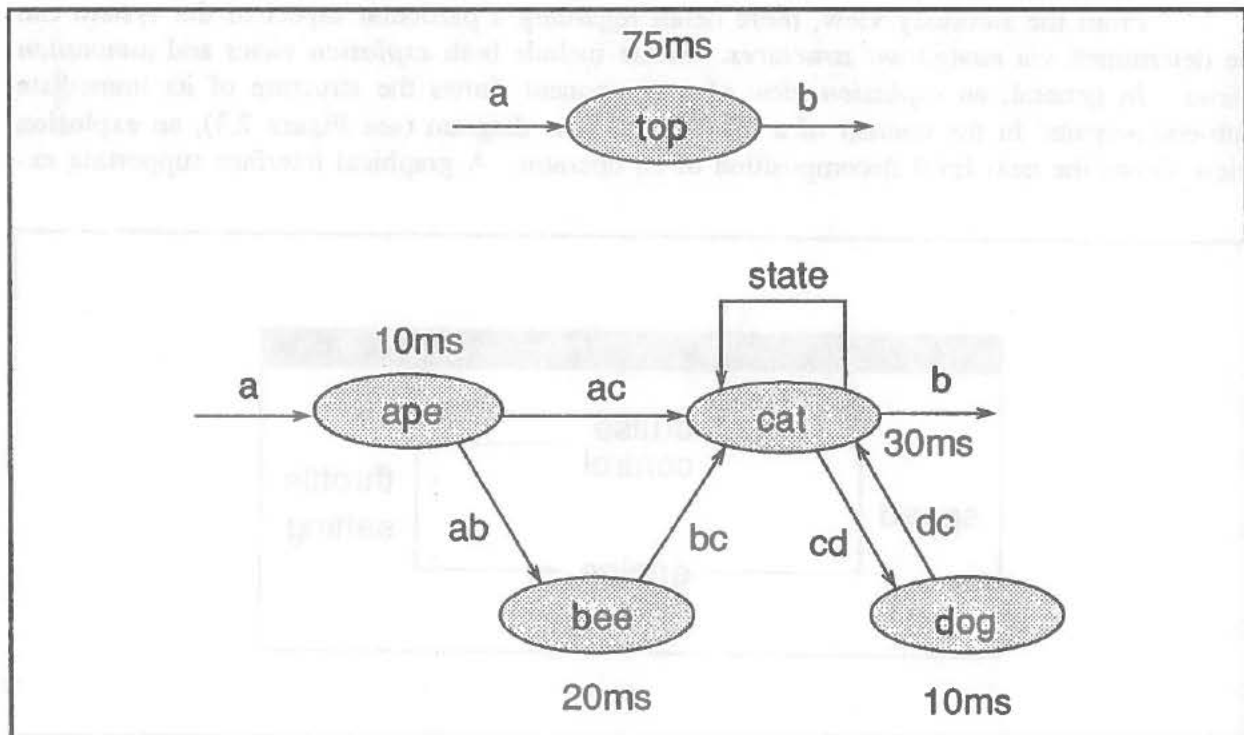

Figure 2.1 Operator Decomposition [7]

tion of execution time and constraints associated with each operator. Figure 2.1 shows how these objects are incorporated into the data flow diagram and decomposed. Note that consistency requires not only that inputs and outputs must match between levels, but timing constraints of a decomposition must not allow a path which has a total execution time in excess of the parent operator's MET.

2.2 Multiple Views

The CAPS database is able to maintain graphical representations such that they may be retrieved in a manner similar to hypertext[4]. That is, each operator exists as a node of a multi-way tree with its associated attributes and links to its parent and child nodes. Additional links are possible representing other types of relationships between nodes. This capability provides for the following proposed set of graphical system representations: *summary views, navigation structures,* and *focused slices*.

A *summary* view serves as an introduction to some aspect of the system under development. This lets someone unfamiliar with the system or component to get "the big picture" such as is necessary in a prototype demonstration or design review. A summary view therefore serves to establish a context for further explanation or detailed examination.

An example useful summary view for a PSDL composite operator is the a data flow diagram of Figure 2.2 showing only the operator's components and their interconnections. Such a view is valuable precisely because of the details it leaves out (such as data types and timing and control constraints). Without such additional detail, the relationship between major components of the operator can be readily understood by the observer.

From the summary view, more detail regarding a particular aspect of the system can be determined via *navigation structures*. These include both *explosion views* and *annotation views*. In general, an *explosion view* of a component shows the structure of its immediate sub-components. In the context of a CAPS data flow diagram (see Figure 2.3), an explosion view shows the next level decomposition of an operator. A graphical interface supporting ex-
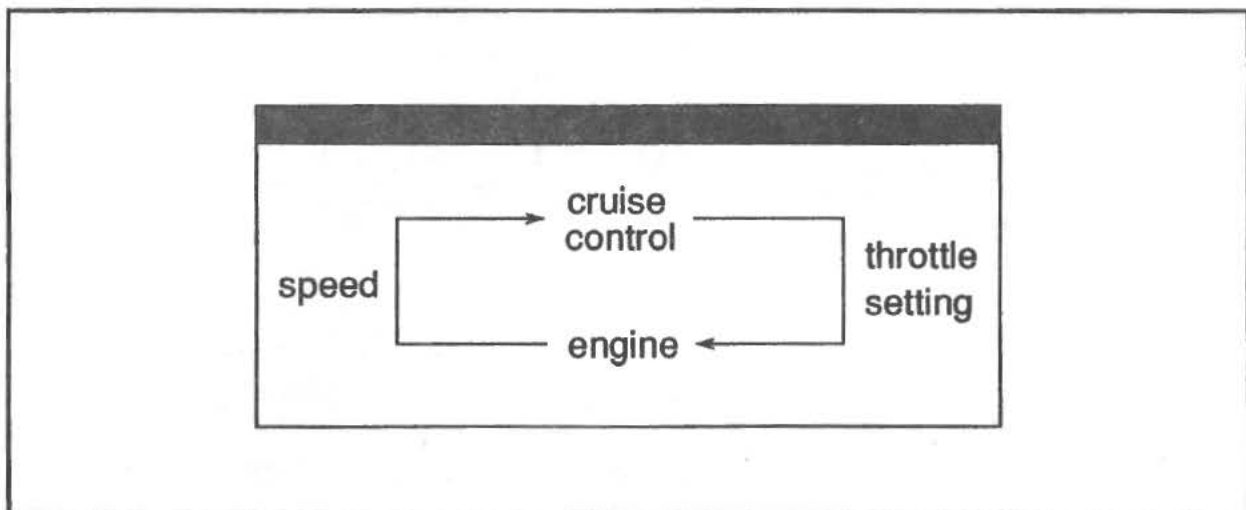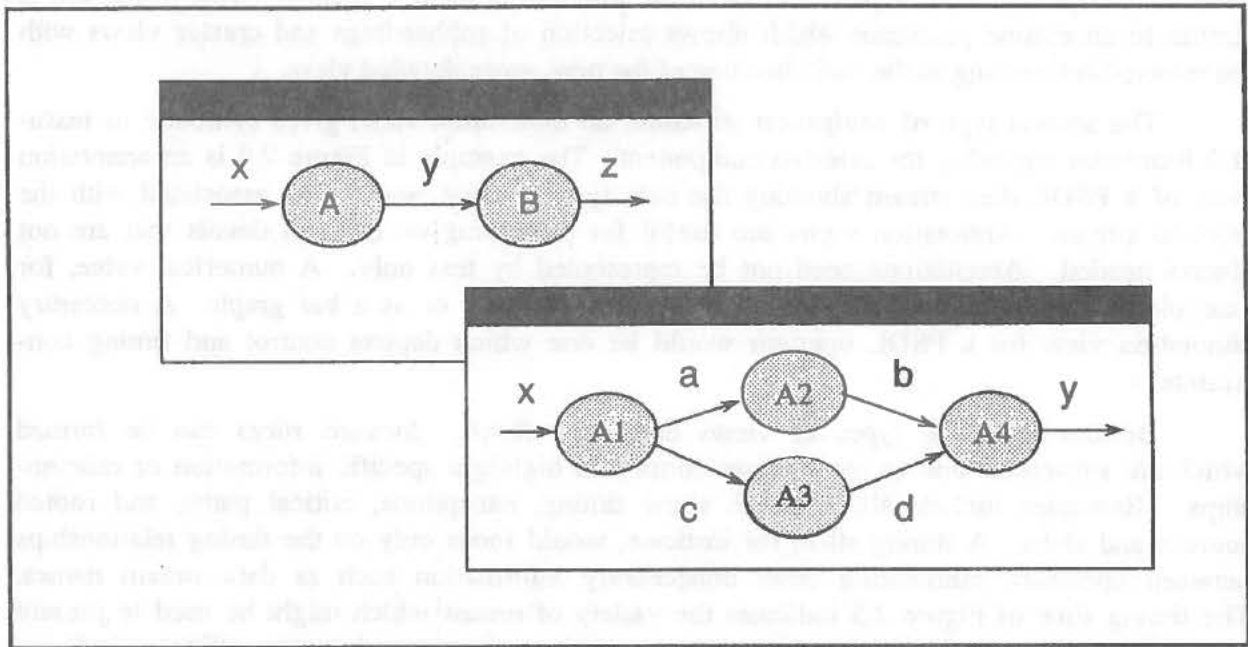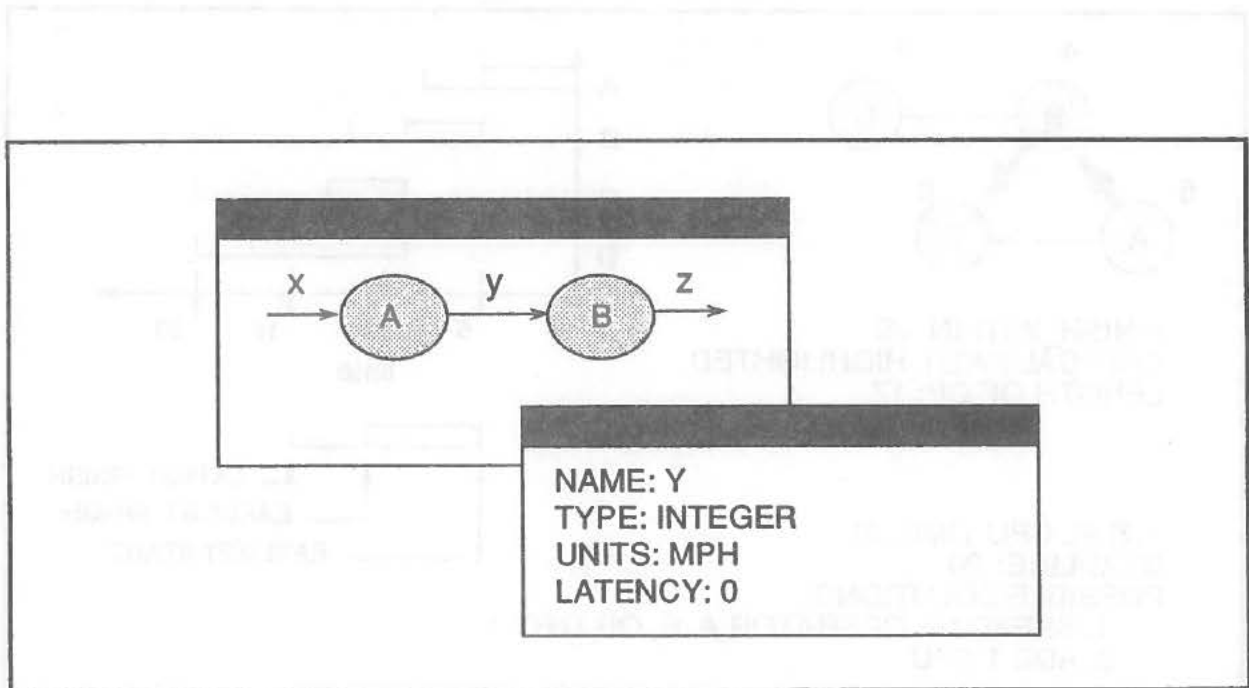
Figure 2.2 Summary View

Figure 2.3 Explosion View of Operator A

Figure 2.4 Annotation View for Stream Y

plosion views makes it very easy for a designer to repeatedly pick and display subcomponents of an operator until a part relative to the problem at hand is located. This procedure is similar to an outline processor which allows selection of subheadings and creates views with the selected subheading as the main heading of the new, more detailed view.

The second type of navigation structure, an *annotation view*, gives symbolic or textual information regarding the selected component. The example in Figure 2.4 is an annotation view of a PSDL data stream showing the data type, latency, and units associated with the selected stream. Annotation views are useful for presenting on demand details that are not always needed. Annotations need not be represented by text only. A numerical value, for example, could be presented digitally, as an analog gauge, or as a bar graph. A necessary annotation view for a PSDL operator would be one which depicts control and timing constraints.

Besides the three types of views described above, *focused slices* can be formed which are subsets of one or more views formed to highlight specific information or relationships. Examples include slices which show timing, exceptions, critical paths, and rooted sources and sinks. A timing slice, for instance, would focus only on the timing relationships between operators, eliminating other unnecessary information such as data stream names. The timing slice of Figure 2.5 indicates the variety of means which might be used to present such information. Similar views for maximum response time or minimum calling periods are also useful for summarizing the timing properties of a system.

Subsets showing just the time-critical operators, periodic operators, or sporadic operators are useful for analyzing timing problems. The two graphics in Figure 2.5 show a high-
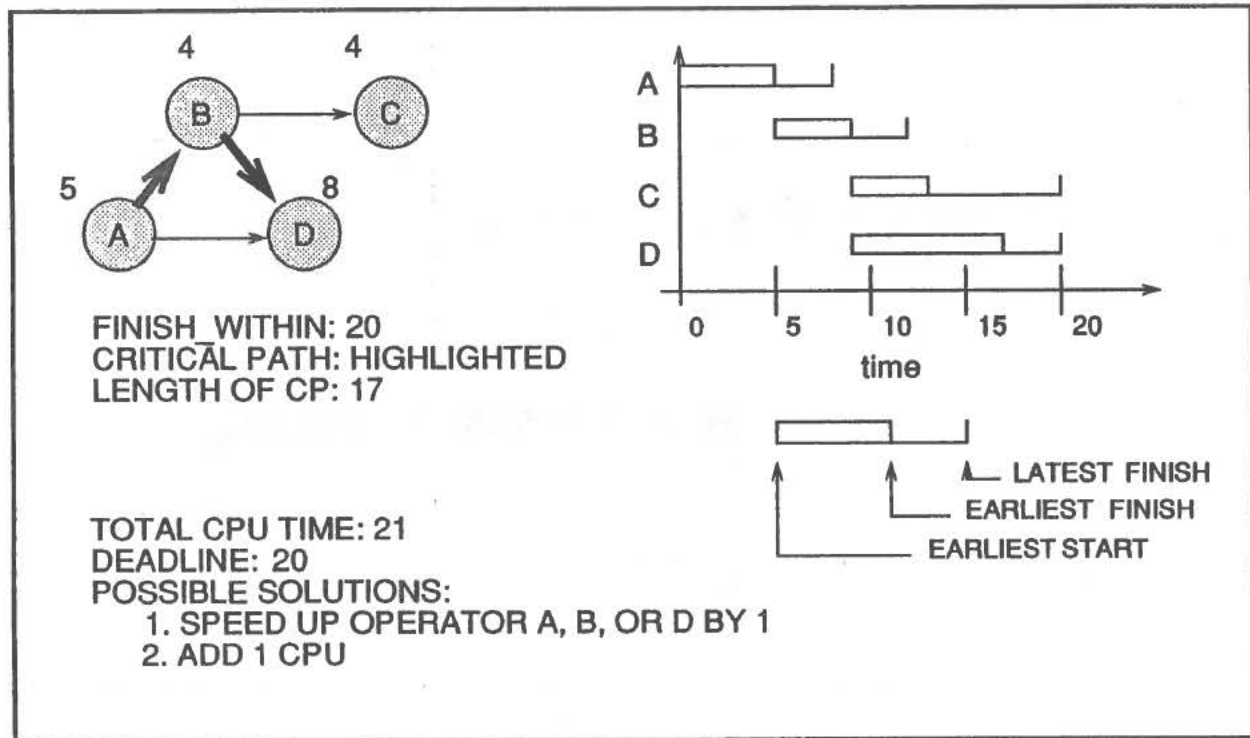

Figure 2.5 Focused Slices Showing Timing and Critical Path

lighted critical path slice and a schedule congestion graph. The latter illustrates the intervals between the earliest and latest time an operator can start executing. Such a display can be useful for the interactive design of a static schedule for a critical component with particularly tight constraints. It can also help identify critical nodes which might benefit most from efficiency improvements. Clearly automating the representation of such information frees the designer to make difficult design decisions rather than become immersed in detail.

Another type of focused slice shows only exception streams and the exception handling operators. Exception slices depict responses components must make to unexpected situations or ill-formed inputs.

2.3 Maintaining Consistency and Correctness

Maintaining consistency in a multi-level, multi-view system provides a considerable challenge. The Coral [10] developers dealt with this problem by associating constraints with objects. Thus manipulating an object takes into account its constraints and context in relation to other objects.

Two types of consistency must be maintained: hierarchical consistency and view consistency. Hierarchical inconsistencies arise since adding objects requires continued top down modification to lower level views. Deleting operators not only requires deletion of an entire decomposition, but also requires modifying each object adjacent to the deleted object.

In the augmented DFD used in CAPS, hierarchical consistency of both i/o and timing must be maintained. As shown in Figure 2.1, the external inputs and outputs of the child must match those of the parent. In addition, no path through the child graph may exceed the MET of the parent.

The graphical editor must also take into account view consistency. Any modifications to the graphic representation will require re-generation of the PSDL link statements along with other associated slice information to maintain the integrity of all views.

Constraints on graphic objects reveal the relationships necessary to affect appropriate changes to the attributes of related objects when the graphic representation is modified. Constraints involve both the application specified values such as timing, as well as the syntactic attributes built into the CAPS. These "built in" constraints not only help to ensure consistency, but also improve correctness.

Constraints on graphic objects can be applied in design of the editor to preclude drawing diagrams with syntactically incorrect internal representations. This prevents the designer from having to check to be sure the correct PSDL statements are being generated. Ideally, the prototype designer should not even need to understand the underlying PSDL syntax.

3. Design of the Graphical Editor

The design and development of a prototype graphic editor for the CAPS was undertaken by a thesis student at the Naval Postgraduate School and the results of that effort are described in this section [11]. First the general requirements are stated. Next considerations

are presented regarding the user interface and i/o. Finally, the main data structures and processing algorithm are described.

3.1 Requirements for the Graphical Editor

The graphical editor shall meet the following general requirements [11]:

• Run in a windowed environment—control movement between levels, selection of editing modes, display of help.

• Ensure syntactic correctness—only accept symbols in the graphic language.

• Support semantic checking—a symbol's context must reflect intended meaning.

• Provide view consistency—changes to the specification and graphical editor must result in comparable updates in the other view.

• Provide hierarchical consistency—changes in one level of decomposition must be reflected in both higher and lower levels as applicable.

The graphical editor shall provide the following functions:

• Display operator context—the operator's name, inputs, outputs, states, and maximum execution time taken from the PSDL specification.

• Draw objects—consisting of operators (bubbles), data streams, inputs, outputs , and self loops (arrows).

• Retrieve and edit—modify existing graphic decompositions.

• Generate PSDL link statements—automatically from the augmented DFD, of the form: *data_stream .source*[*:met*] --> *destination*.

3.2 Interface Design

Four factors influenced the design of the graphical editor's user interface [11]:

1. the choice of machines on which to implement CAPS

2. the choice of interface software support

3. human factors issues

4. user interface design guidelines

3.2.1 Sun Workstation

The Sun Workstation has many features which make it the machine of choice for the development and implementation of CAPS. The availability of a dedicated CPU in a multitasking environment greatly enhances the design team's ability to code, test and debug their software. The Sun Workstation also provides a powerful integrated programming environment based on the unix operating system.

3.2.2 Sun View

The Sun View window based user environment supports interactive graphics-based applications with multiple overlapping windows, each of which can run a task independent of the other windows, as well as a general toolkit for building window-based applications.

3.2.3 Human Factors

The following human factors principles were considered in designing the graphical interface:

1. **Functional Principle.** Controls which are grouped according to their functionality are easier to learn and result in fewer errors [8]. The graphical editor's controls fall into three functional groups:

- session control group
- drawing mode group
- text input group

2. **Sequence of Use Principle.** Controls which are organized in the same sequence that they are used eliminates the need to jump around and therefore minimizing the amount of information the user must remember [8]. The controls of the graphical editor are organized to be used in a series of top to bottom sequences. The top panel is used to control the basic system functions of loading and storing decompositions and quitting the tool. As such, it is used at the beginning and end of an editing session. The next panel down is the drawing mode panel. It is used to switch from drawing one type of object to another. After the drawing mode is changed, the user must enter a name and in the case of an operator, a time constraint. The input panels for these are therefore located immediately below the drawing mode panel. The drawing canvas is located immediately below the input panels. This ordering of controls always allows, but does not force, the user to operate in a top to bottom circular fashion as follows:

- select mode (optional)
- enter and read name
- enter and read time constraint (if in operator mode)
- draw object
- repeat until done

3. **Human Memory Capacity Principle.** Studies have shown that humans have the capacity to remember 7 $\pm$2 things at once [8]. The graphical editor was designed to meet this criteria. It consists of five basic parts and its longest menu has only five choices.

3.2.4 User Interface Design Guidelines

Guidelines which should be applied when designing a user interface include the following [1]:

- Be intuitive (things should work as you would expect)
- Accommodate experts and novices (provide confirmation override mechanisms)
- Allow customization
- Provide extensibility
- Use lots of feedback (show status; make error messages clear)
- Be predictable (use a consistent, easy to remember set of basic actions in obvious ways)
- Be deterministic (consider type ahead and mouse ahead effects)
- Avoid modes (if states that persist are necessary, make the feedback and exit path obvious)
- Do not preempt the user (don't force them to respond)

3.3 Input/Output Considerations

3.3.1 Inputs

The graphical editor accepts inputs from both the mouse and the keyboard. No restrictions are placed on the order that any of these inputs must occur except that each type of object should be drawn with a given name.

An **operating mode select** event occurs when one of the operating mode buttons is selected via the mouse. Selectable operating modes include (1) load an existing diagram, (2) store the current diagram or (3) quit. The default mode is for the editor to be ready to create a new diagram.

An **drawing mode select** event is also a mouse input. This input establishes the context in which canvas events will be interpreted.

To **draw an object**, the mouse is used in a typical rubber-banding point and click style with the release of the mouse button completing the drawing.

The point and click method is also used to **delete an object**.

Textual inputs are typed in via the keyboard. First the mouse pointer must be positioned in a text panel and after typing the text, the corresponding read button is selected to initiate text processing.

When the graphical editor is used to edit an existing diagram, the system retrieves the necessary reconstruction information from the design database and. The graphical editor then reads this information and reconstructs the diagram.

3.3.2 Outputs

The graphical editor has two kinds of outputs: visual and textual. If the user draws an object, it is displayed on the canvas so that he can see it. If a user generated error occurs, an error message will immediately be displayed at the top of the canvas. Once the error condition has been corrected, the error message disappears.

When the mouse is in a particular subwindow of the display, visual feedback in the form of a bold sub-window border, is provided.

After a decomposition has been completed and the user has selected store, the editor will generate two kinds of textual output. The PSDL link statements along with additional information needed to reconstruct the display will be written to a file. The CAPS user interface will store this information in the design database [7].

3.4 Data Structures

The primary storage structure for the graphical editor is a linked list of operators [11]. A single structure named *Operator_list* has pointers to the head and tail elements of this list. Each element in the operator list is a structure of type Operator. The Operator structure has ten fields.

The **name** field is a variable length string representing the name of the operator. The **optype** field is used to signify whether the structure represents an actual operator or whether it represents a NULL operator which is used as the source for external inputs. The **xstart** and **ystart** fields specify the starting (x,y) coordinates of the operator. The **xstop** and **ystop** fields contain the operator's stopping (x,y) coordinates. The **met** field contains a variable length string representing the operators maximum execution time. The last three fields are pointers. The **head** and **tail** fields point to a linked list of lines which leave this operator. The **next** field points to the next operator in the linked list of operators.

As mentioned in the previous paragraph, each operator has an associated *Line-list*. This *Line-list* is a linked list of structures of type *Line*. Lines originate at the operator to which they are attached.

The **name, xstart, ystart, xstop, ystop** and **next** fields of the *Line-list* structure serve the same purpose as in the *Operator-list* structure. The **lntype** field is identifies whether the line is an input, output, data stream or a state. The **dest** field holds the name of the operator on which the line terminates. For output lines, this field will always point to a name structure containing the name EXTERNAL.

3.5 Algorithm Description

At a very high level, the algorithm for the graphical editor is simply:

- create the window

- poll for events

Sun View has built-in routines which allow the interface designer to construct an interactive window application. One need only provide the routines for handling the details of the application. The following is a description of the main routines of the graphical editor [11].

3.5.1 Create the User Interface

The user interface for the graphical editor is a window comprised of five sub-windows, each of which is one of the Sun View application building blocks. The Window is created as follows:

- **Create a frame.** This is done with the Sun Window routine *window_create*. Parameters for this routine allow the specification of various attributes for the frame object. These attributes include its name, icon, size, location, window type, location on the screen and numerous others.

- **Create each of the sub-windows.** Again using the routine *window_create*, the graphical editor frame is tiled with four panel sub-windows and a drawing canvas sub-window.

3.5.2 Poll for Events

This function is greatly simplified by the Sun View notification-based system. Rather than maintain a main event polling loop within the application program, Sun View has the polling loop in a notifier. The notifier reads events and then notifies the appropriate application procedure that input has occurred. This scheme requires that each procedure must be registered with the notifier so that it knows who to call for a particular event. So procedures which are to be called as a result of a button being pushed are registered with the notifier by the *panel_create_item* routine [9]. The events which are accepted by the graphical editor are described as follows:

3.5.2.1 System Control Events

The **load existing, store,** and **quit** event buttons are located in the top subwindow of the graphical editor frame. When the graphical editor is started it comes up in a mode which allows the user to create a new decomposition diagram. The three selectable events are described as follows:

1. **Load Existing.**

   - The routine *load_proc* reads the reconstruction data from a file and checks its type. This data must have been previously retrieved from the design database by the CAPS user interface [7].

   - If the object is an operator, an operator storage element is created, filled in with its information, and is attached to the list of operators.

   - If the object is of type EXTERNAL, an operator element is also created and is linked to the operator list. EXTERNALs are NULL operator nodes which serve as the source operator for input lines. The only fields of an EXTERNAL which get useful values are those pointing at the line list.

   - Any object encountered during the load process which is not an operator or external is some type of line. Therefore, a line storage element is created, its values are filled in and it is linked to the line list of the last operator which was read in.

- After all of the objects in the file being loaded have been read in, stored and linked, the diagram is ready to be drawn. *Load_proc*'s final action is to call the routine *redraw_diagram* which traverses the entire linked storage structure and draws each object.

2. **Store**.

- Information for diagram reconstruction is stored by the routine *store_diagram*. This routine does a traversal of the storage structure, writing out the contents of each operator node immediately followed by the contents of each of its associated line nodes.

- Creating the PSDL link statements is a similar process. The *create_PSDL* routine traverses the entire storage structure, generating a PSDL link statement for each line node it finds. The link statement is a string of characters which result from the concatenation of the following six sub-strings:

  - the line name (stored in the line node)
  - the character "."
  - the source operator's name (the name of the operator whose lines are being processed)
  - an optional ":" and MET (if the source operator has an MET)
  - the character string "-->"
  - the destination operator's name (stored in one of the line nodes fields)

- These link statements will be attached to the implementation part of the PSDL specification file by the sequence control function [7].

- After the diagram has been stored, the *store_diagram* routine tells the system that it is safe to exit.

3. **Quit**.

- The routine *quit_proc* will first check to see if the diagram has been stored. If so, it will destroy the window.

- If the diagram has not been saved, an error message will appear on the drawing canvas telling the user to store the diagram.

- The user can terminate the session without saving by quitting via the normal Sun View windowing menu. Selecting "Quit" from this menu will circumvent the storage check and kill the editor.

3.5.2.2 Mode Select Events

The graphical editor always starts in the *draw_operator* mode. This is because operators must be drawn before data streams. This requirement has the advantage of making it easy to check the syntax of the diagram. The editor ensures that lines intersect operators in a way appropriate to their type (i.e. input, output, etc.).

To switch operating modes, the user clicks on the desired mode. The selector will reverse its color indicating that it has been selected. The selection causes the notifier to call the *mode_select* routine which sets the global *edit_mode* variable to the appropriate value. This establishes the context in which canvas events for the left mouse button will be interpreted.

3.5.2.3 Text Panel Events

The graphical editor has two panels which provide a means of entering textual information.

1. **Name Panel.**

• The name panel allows the user to enter a name for an operator or a line.

• The *read_name* button must then be selected causing the notifier to inform the routine *input_name* to read the panel and the routine *is_valid_ada_id* to check the syntax of the name.

• If the name is not a valid Ada identifier (PSDL identifier syntax matches Ada), an error message is displayed and the name must be edited before drawing events on the canvas.

2. **MET Panel.**

• The MET panel works essentially the same as the name panel. The difference is that the routine *is_valid_MET* is used to check that the value is an integer and has the appropriate units.

• Invalid values result in an error message and a lockout of operator events from the canvas since only operators have a MET.

3.5.2.4 Canvas Events.

The graphical editor screens the event handler for left mouse down events, left mouse drag events, left mouse up events and right mouse down events.

1. **Left Mouse Down.**

• Capture the x and y coordinates of the position where the event occurred. These values are stored and become the starting position of the object being drawn. Which object is drawn depends upon the current drawing mode.

2. **Left Mouse Drag.**

• Rubber-band the object. As the mouse pointer is moved across the screen *process_canvas_events* repeatedly captures the position of the pointer. For each new position, the routine rubber-band is called to blank out the previous version of the object and then redraw it using the most recent starting and stopping coordinates. The result is that the line is erased and redrawn as fast as the user moves the pointer across the screen.

3. **Left Mouse Up.**

• Call rubber-band a last time to delete the last rubber-banded version then capture the final stopping coordinate. The routine *process_object* then performs syntactic and semantic checks on the object.

• If the editing mode is *draw_operator*, the routine *process_object* will verify that a name and a MET are available and that the coordinates of the new operator do not overlap another operator.

• When all of these conditions are satisfactory it calls the routine *process_operator*. This routine will cause the following seven steps to take place:

> • the object will be drawn
>
> • the MET will be retrieved
>
> • the name will be retrieved
>
> • the name will be displayed, centered in the operator
>
> • the MET will be displayed, centered over the operator
>
> • the operator will be allocated storage and stored
>
> • the stored operator will be appended to the list of operators

• If the drawing mode is *draw_data_stream*, *draw_input*, *draw_output*, or *draw_self_loop*, routine *process_object* will verify availability of a legal Ada identifier and will ensure the line intersects an operator in the appropriate fashion.

• If the checks turn out satisfactory, the routine *process_line* is called. This routine will cause the following actions:

> • draw the appropriate line
>
> • draw the arrowhead on the end of the line
>
> • if the line is an input line, create a NULL operator to act as its source and link the NULL operator to the operator list
>
> • retrieve the line's name
>
> • display the name on the line
>
> • create the storage for the line and fills in the values
>
> • append the line to the source operator's list of lines

4. Right Mouse Down.

When a right mouse down event occurs, the routine *process_canvas_events* checks to see if the mouse's coordinates are within the pick criteria of either a line or an operator. If so, the object is deleted from the storage structure by routine *delete_line* or *delete_op* as appropriate. After the deletion is complete, routine *redraw_diagram* draws the remaining objects.

4. Conclusions and Recommendations

This paper discussed the capability of graphical representations to ease the prototyping process and reduce the problem of information overload. The application of information hiding and multiple views, coupled with ensuring consistency and automatic programming promise a significant improvement in user productivity. Development of a graphical editor for performing hierarchical decomposition of composite PSDL operators for CAPS was also discussed. Research on the graphical editor, as it relates to PSDL, indicates that a prototype design can be developed with much greater ease than with only the syntax-directed editor. Graphic editor capabilities will also greatly enhance prototype modification, presentation, and documentation for further development.

Work is still needed in integrating the graphic editor, syntax-directed editor, and database with the user interface. Only the most basic DFD view has been implemented in the graphic editor and a more sophisticated means of automatically performing consistency updates should be pursued. An expert mode is desperately to increase productivity, and a number of enhancements could be made to improve user-friendliness in the graphical interface.

Much work needs to be done. Application of graphical representation of information needed by software engineers for making qualitative design decisions is years behind the application of similar technology in business and industry. It's time *decision support* came home to the software community in recognition that software development is requiring an ever greater portion of the available corporate budget.

5. References

1. F. Hopgood and others, *Methodology of Window Management*, Springer-Verlag, 1986.

2. Luqi, V. Berzins and R. Yeh, "A Prototyping Language for Real-Time Software", *IEEE Trans. on Software Eng.*, October, 1988, 1409-1423.

3. Luqi and V. Berzins, "Rapidly Prototyping Real-Time Systems", *IEEE Software*, Sep. 1988, 25-36.

4. Luqi and M. Ketabchi, "A Computer Aided Prototyping System", *IEEE Software 5*, 2 (March 1988), 66-72.

5. Luqi, "Software Evolution via Rapid Prototyping", *IEEE Computer*, May 1989.

6. Luqi and Y. Lee, "Interactive Control of Prototyping Process", Technical Report NPS 52-89-014, Computer Science Department, Naval Postgraduate School, 1989.

7. H. Raum, "Design and Implementation of an Expert User Interface for the Computer Aided Prototyping System", M. S. Thesis, Computer Science, Naval Postgraduate School, Monterey, CA, Dec. 1988.

8. M. Sanders and E. McCormick, *Human Factors in Engineering and Design*, 6th Edition, McGraw-Hill, 1987.

9. *Sunview Programmer's Guide, Revision: A*, Sun Microsystems Inc., Oct. 1986.

10. P. Szekely and B. Myers, "A User Interface Toolkit Based on Graphical Objects and Constraints", in *Proceedings of ACM OOPSLA Conference*, 1988.

11. R. Thorstenson, "A Graphical Editor for the Computer Aided Prototyping System", M. S. Thesis, Computer Science, Naval Postgraduate School, Monterey, CA, Dec. 1988.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center   2
    Cameron Station
    Alexandria, Virginia 22304-6145

2. Library, Code 0142   2
    Naval Postgraduate School
    Monterey, California 93943-5002

3. Office of Naval Research
    Office of the Chief of Naval Research   1
    Attn. CDR Michael Gehl, Code 1224
    800 N. Quincy Street
    Arlington, Virginia 22217-5000

4. Space and Naval Warfare Systems Command
    Attn. Dr. Knudsen, Code PD 50   1
    Washington, D.C. 20363-5100

5. Ada Joint Program Office   1
    OUSDRE(R&AT)
    Pentagon
    Washington, D.C. 20301

6. Naval Sea Systems Command   1
    Attn. CAPT Joel Crandall
    National Center #2, Suite 7N06
    Washington, D.C. 22202

7. Office of the Secretary of Defense   1
    Attn. CDR Barber
    STARS Program Office
    Washington, D.C. 20301

8. Office of the Secretary of Defense   1
    Attn. Mr. Joel Trimble
    STARS Program Office
    Washington, D.C. 20301

9. Commanding Officer   1
    Naval Research Laboratory
    Code 5150
    Attn. Dr. Elizabeth Wald
    Washington, D.C. 20375-5000

10. Navy Ocean System Center   1
    Attn. Linwood Sutton, Code 423
    San Diego, California 92152-500

11. National Science Foundation   1
    Attn. Dr. William Wulf
    Washington, D.C. 20550

12. National Science Foundation   1
    Division of Computer and Computation Research
    Attn. Dr. Peter Freeman
    Washington, D.C. 20550

13. National Science Foundation   1
    Director, PYI Program
    Attn. Dr. C. Tan
    Washington, D.C. 20550

14. Office of Naval Research   1
    Computer Science Division, Code 1133
    Attn. Dr. Van Tilborg
    800 N. Quincy Street
    Arlington, Virginia 22217-5000

15. Office of Naval Research   1
    Applied Mathematics and Computer Science, Code 1211
    Attn: Dr. James Smith
    800 N. Quincy Street
    Arlington, Virginia 22217-5000

16. New Jersey Institute of Technology   1
    Computer Science Department
    Attn. Dr. Peter Ng
    Newark, New Jersey 07102

17. Southern Methodist University   1
    Computer Science Department
    Attn. Dr. Murat Tanik
    Dallas, Texas 75275

18. Editor-in-Chief, IEEE Software   1
    Attn. Dr. Ted Lewis
    Oregon State University
    Computer Science Department
    Corvallis, Oregon 97331

19. University of Texas at Austin   1
    Computer Science Department
    Attn. Dr. Al Mok
    Austin, Texas 78712

20. University of Maryland
    College of Business Management
    Tydings Hall, Room 0137
    Attn. Dr. Alan Hevner
    College Park, Maryland 20742   1

21. University of California at Berkeley
    Department of Electrical Engineering and Computer Science
    Computer Science Division
    Attn. Dr. C.V. Ramamoorthy
    Berkeley, California 94720   1

22. University of California at Los Angeles
    School of Engineering and Applied Science
    Computer Science Department
    Attn. Dr. Daniel Berry
    Los Angeles, California 90024   1

23. University of Maryland
    Computer Science Department
    Attn. Dr. Y. H. Chu
    College Park, Maryland 20742   1

24. University of Maryland
    Computer Science Department
    Attn. Dr. N. Roussapoulos
    College Park, Maryland 20742   1

25. Kestrel Institute
    Attn. Dr. C. Green
    1801 Page Mill Road
    Palo Alto, California 94304   1

26. Massachusetts Institute of Technology
    Department of Electrical Engineering and Computer Science
    545 Tech Square
    Attn. Dr. B. Liskov
    Cambridge, Massachusetts 02139   1

27. Massachusetts Institute of Technology
    Department of Electrical Engineering and Computer Science
    545 Tech Square
    Attn. Dr. J. Guttag
    Cambridge, Massachusetts 02139   1

28. University of Minnesota
    Computer Science Department
    136 Lind Hall
    207 Church Street SE
    Attn. Dr. J. Ben Rosen
    Minneapolis, Minnesota 55455   1

29. International Software Systems Inc.   1
    12710 Research Boulevard, Suite 301
    Attn. Dr. R. T. Yeh
    Austin, Texas 78759

30. Software Group, MCC   1
    9430 Research Boulevard
    Attn. Dr. L. Belady
    Austin, Texas 78759

31. Carnegie Mellon University   1
    Software Engineering Institute
    Department of Computer Science
    Attn. Dr. Lui Sha
    Pittsburgh, Pennsylvania   15260

32. IBM T. J. Watson Research Center   1
    Attn. Dr. A. Stoyenko
    P.O. Box 704
    Yorktown Heights, New York 10598

33. The Ohio State University   1
    Department of Computer and Information Science
    Attn. Dr. Ming Liu
    2036 Neil Ave Mall
    Columbus, Ohio 43210-1277

34. University of Illinois   1
    Department of Computer Science
    Attn. Dr. Jane W. S. Liu
    Urbana Champaign, Illinois 61801

35. University of Massachusetts   1
    Department of Computer and Information Science
    Attn. Dr. John A. Stankovic
    Amherst, Massachusetts 01003

36. University of Pittsburgh   1
    Department of Computer Science
    Attn. Dr. Alfs Berztiss
    Pittsburgh, Pennsylvania 15260

37. Defense Advanced Research Projects Agency (DARPA)   1
    Integrated Strategic Technology Office (ISTO)
    Attn. Dr. Jacob Schwartz
    1400 Wilson Boulevard
    Arlington, Virginia 22209-2308

38. Defense Advanced Research Projects Agency (DARPA)   1
    Integrated Strategic Technology Office (ISTO)
    Attn. Dr. Squires
    1400 Wilson Boulevard
    Arlington, Virginia 22209-2308

39. Defense Advanced Research Projects Agency (DARPA)   1
    Integrated Strategic Technology Office (ISTO)
    Attn. MAJ Mark Pullen, USAF
    1400 Wilson Boulevard
    Arlington, Virginia 22209-2308

40. Defense Advanced Research Projects Agency (DARPA)   1
    Director, Naval Technology Office
    1400 Wilson Boulevard
    Arlington, Virginia 2209-2308

41. Defense Advanced Research Projects Agency (DARPA)   1
    Director, Strategic Technology Office
    1400 Wilson Boulevard
    Arlington, Virginia 2209-2308

42. Defense Advanced Research Projects Agency (DARPA)   1
    Director, Prototype Projects Office
    1400 Wilson Boulevard
    Arlington, Virginia 2209-2308

43. Defense Advanced Research Projects Agency (DARPA)   1
    Director, Tactical Technology Office
    1400 Wilson Boulevard
    Arlington, Virginia 2209-2308

44. MCC AI Laboratory   1
    Attn. Dr. Michael Gray
    3500 West Balcones Center Drive
    Austin, Texas 78759

45. COL C. Cox, USAF   1
    JCS (J-8)
    Nuclear Force Analysis Division
    Pentagon
    Washington, D.C. 20318-8000

46. LTCOL Kirk Lewis, USA   1
    JCS (J-8)
    Nuclear Force Analysis Division
    Pentagon
    Washington, D.C. 20318-8000

47. University of California at San Diego   1
   Department of Computer Science
   Attn. Dr. William Howden
   La Jolla, California 92093

48. University of California at Irvine   1
   Department of Computer and Information Science
   Attn. Dr. Nancy Levenson
   Irvine, California 92717

49. University of California at Irvine   1
   Department of Computer and Information Science
   Attn. Dr. L. Osterweil
   Irvine, California 92717

50. University of Colorado at Boulder   1
   Department of Computer Science
   Attn. Dr. Lloyd Fosdick
   Boulder, Colorado 80309-0430

51. Santa Clara University   1
   Department of Electrical Engineering and Computer Science
   Attn. Dr. M. Ketabchi
   Santa Clara, California 95053

52. Oregon Graduate Center   1
   Portland (Beaverton)
   Attn. Dr. R. Kieburtz
   Portland, Oregon 97005

53. Dr. Wolfgang Halang   1
   Bayer AG
   Ingenieurbereich Progessleittechnik
   D-4047
   Dormagen, West Germany

54. Dr. Bernd Kraemer   1
   GMD Postfach 1240
   Schloss Birlinghaven
   D-5205
   Sankt Augustin 1, West Germany

55. Dr. Aimram Yuhudai   1
   Tel Aviv University
   School of Mathematical Sciences
   Department of Computer Science
   Tel Aviv, Israel 69978

56. Dr. Robert M. Balzer   1
    USC-Information Sciences Institute
    4676 Admiralty Way
    Suite 1001
    Marina del Ray, California 90292-6695

57. U.S. Air Force Systems Command   1
    Rome Air Development Center
    RADC/COE
    Attn. Mr. Samuel A. DiNitto, Jr.
    Griffis Air Force Base, New York 13441-5700

58. U.S. Air Force Systems Command   1
    Rome Air Development Center
    RADC/COE
    Attn. Mr. William E. Rzepka
    Griffis Air Force Base, New York 13441-5700

59   LuQi   50
    Code 52Lq
    Computer Science Department
    Naval Postgraduate School
    Monterey, CA 93943-5100

60. Research Administration   1
    Code: 012
    Naval Postgraduate School
    Monterey, CA.   93943

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release;
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	distribution is unlimited.

4. PERFORMING ORGANIZATION REPORT NUMBER(S) NPS52-89-028	5. MONITORING ORGANIZATION REPORT NUMBER(S)

6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School	6b OFFICE SYMBOL (If applicable) 52	7a. NAME OF MONITORING ORGANIZATION National Science Foundation & ONR Sponsored Navy Direct Funding
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943		7b. ADDRESS (City, State, and ZIP Code) Washington, D.C. 20550
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Naval Postgraduate School	8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER NSF CCR-8710737 O&MN, DirectFunding

8c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

GRAPHICAL SUPPORT FOR REDUCING INFORMATION OVERLOAD IN RAPID PROTOTYPING (U)

12. PERSONAL AUTHOR(S)
LUQI, P. Barnes, M. Zyda

13a. TYPE OF REPORT Progress	13b. TIME COVERED FROM Sept 88 TO Mar 89	14. DATE OF REPORT (Year, Month, Day) 89/03	15 PAGE COUNT 27

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Software Engineering, Rapid Prototyping, Computer Aided
			Design, Computer Graphics.

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

The basic problem in rapid prototyping of software is information overload. Graphic interfaces can help by providing multiple views, where each view is limited to providing information relevant to a particular task or problem. The graphic editor under development for the Computer Aided Prototyping System (CAPS) proposes a data flow diagram based model with multiple views and automatic program generation to manage the quantity of information necessary to prototype large, real-time systems.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL LUQI	22b. TELEPHONE (Include Area Code) (408) 646-2735	22c. OFFICE SYMBOL 52Lq

DD FORM 1473, 84 MAR · · · · · 83 APR edition may be used until exhausted.
All other editions are obsolete