Graphical tool for computer-aided prototyping

Luqi, P D Barnes and M Zyda

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 graphical editor under development for the Computer Aided Prototyping System (CAPS) proposes a dataflow-diagram-based model with multiple views and automatic program generation to manage the quantity of information necessary to prototype large, real-time systems.

software engineering, rapid prototyping, computer-aided design, computer graphics, graphical editor

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) under development at the Department of Computer Science, Naval Postgraduate School, replaces the traditional software life-cycle with a two-phase cycle that consists of rapid prototyping and automatic program generation[1]. 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[2,3].

Although prototyping generally involves dealing with problems at a high level of abstraction, crucial decisions that designers must make are still too many to be evaluated at a single level. As the designer delves beneath the surface into increasing detail, the amount of information that must be retained to make good design decisions becomes unmanageable. The designer quickly becomes inundated with an overload of information — thus the term 'information overload'. Prototyping large real-time systems requires tools for managing this information such that unnecessary detail may be hidden and essentials 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.

Department of Computer Science, Naval Postgraduate School, Monterey, CA 93943, USA.
Paper submitted: 10 May 1989.
Revised version received: 5 December 1989.

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[4]. 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, so 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 to be effective the application of graphical techniques must be coupled with a strategy for extracting a meaningful subset of available information. Yourdon[5] lists the following important features of automated tools for systems analysis and design:

- graphics support for multiple types of models
- error-checking features to ensure model accuracy
- cross-checking of different models
- additional software engineering support

Development of a graphical editor, then, requires addressing the following issues. First, it must provide multiple system views — reducing the amount and detail of information that must be assimilated at any one time. Second, a provision has to be made for maintaining consistency between the Prototype System Description Language (PSDL) and various graphic representations, as well as syntactic and semantic correctness of the design. Finally, under the subject of additional software engineering support, a graphic representation for prototyping must be automatically programmable. In the case of CAPS, it must map directly to equivalent PSDL representations with which CAPS can construct a prototype[3,6]. The following sections describe these issues, beginning with the goal of the editor, the generation of PSDL code through automatic programming.

Automatic programming

Automatic programming is a promising means of improving programmer productivity[7]. That is, the automatic generation of code from software specifications rather than manual generation through several layers of language translation. The PSDL prototyping language[8] used by CAPS applies this concept. PSDL specifications may be used directly 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, and 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 datum 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 behaviour by specifying conditions regarding its firing (execution) or input/output processing. While control constraints specify when an operator executes, timing constraints determine its execution time, response time, and period.

The best representation or graphic model to use to both represent the prototype and map to PSDL is a problem with no clear solution. Pressman[7] provides a taxonomy of graphical models for analysis and design based on various paradigms. However, no models specifically for prototyping are discussed. This is probably because most prototyping approaches are just starting to address graphical representation issues. An exception is ENVISAGER[6], which provides an animated simulation modelling capability where each database object has a unique iconic representation. Users combine components from the icon-base with communication paths to form a prototype. An example of an almost purely graphic-based prototyping system is the Oregon Speedcode Universe (OSU)[9]. The OSU is based on the notion of 'showing rather than telling' the prototype what to do. It takes advantage of existing user-interface objects in a common standardized environment to generate prototype user interfaces that require from one-half to one-tenth the normal amount of code.

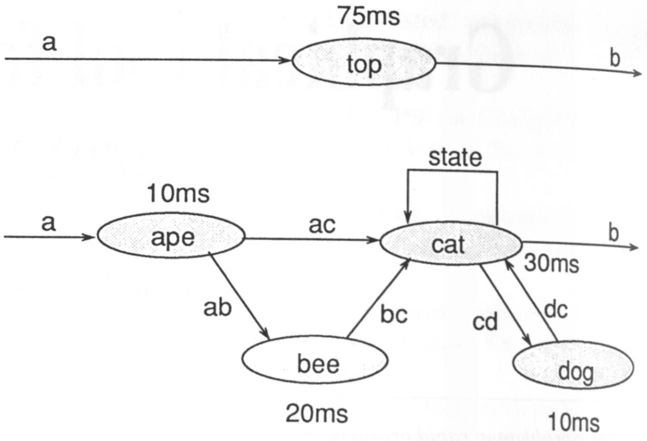


Figure 1. Operator decomposition[8]

The choice of graphical representations in the aforementioned systems is highly dependent on the class of prototypes the systems are targeted toward. CAPS is designed to meet the strict requirements of hard real-time systems. Thus a more general approach to graphic representation is taken. Although constrained to the underlying PSDL model, CAPS is not limited to a single view of the prototype. Rather, the approach taken is one of a basic dataflow diagram (DFD)[5] augmented with additional views as required to specify clearly the prototype. This multi-view approach is discussed further in the next section and is based on the concepts introduced by Smedstad and Anderson[10] and explored by Barnes and Hartrum[11]. This allows the prototyper to design without being constrained to a fixed set of components in the design database.

The PSDL model can easily be mapped directly to the augmented DFD[8]. Execution time and constraints associated with each operator are incorporated into the DFD and decomposed (see Figure 1). 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 that has a total execution time in excess of the parent operator's Maximum Execution Time (MET).

Multiple views

The CAPS database is able to maintain graphical representations so that they may be retrieved in a manner similar to hypertext[1]. 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 get 'the big picture', as is necessary in a prototype demon-

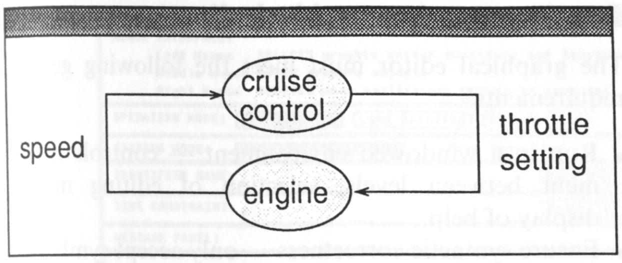


Figure 2. Summary view

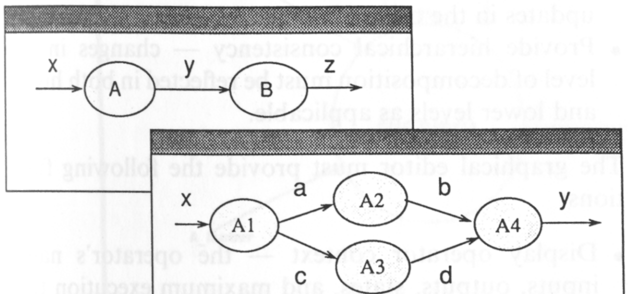


Figure 3. Exploring view of operator A

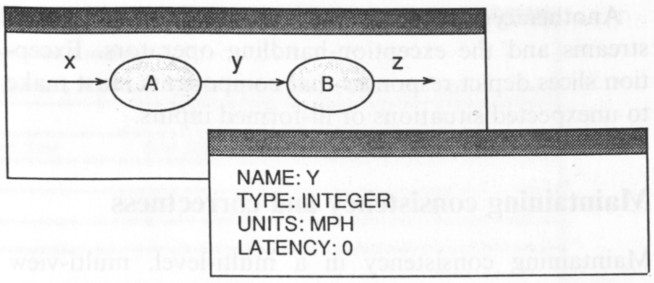


Figure 4. Annotation view for stream y

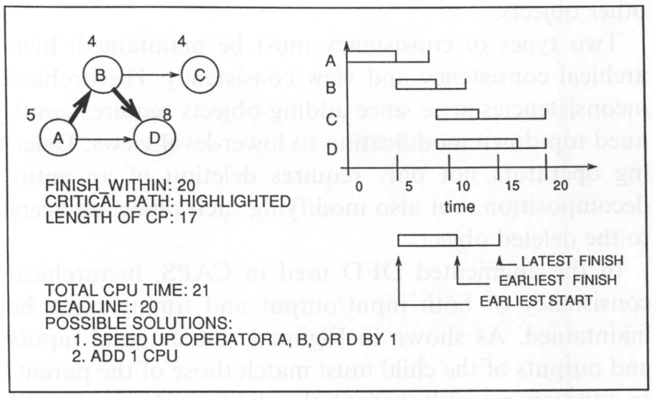


Figure 5. Focused slices, showing timing and critical path

stration 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 DFD of Figure 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 about a particular aspect of the system can be determined via navigation structures. These include both exploding views and annotation views. In general, an exploding view of a component shows the structure of its immediate subcomponents. In the context of the CAPS DFD of Figure 3, the exploding view shows the next level decomposition of operator A. Note that consistency among the data streams in the first view is maintained in the exploding view (specifically x and y). A graphical interface supporting exploding views makes it 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 about the selected component. The example in Figure 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 presented 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 that 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 that 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 5 indicates the variety of means that 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 that show just the time-critical operators, periodic operators, or sporadic operators are useful for analysing timing problems. The two graphics in Figure 5 show a highlighted 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 that 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 that components must make to unexpected situations or ill-formed inputs.

Maintaining consistency and correctness

Maintaining consistency in a multi-level, multi-view system provides a considerable challenge. The Coral[12] 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 modifying each object adjacent to the deleted object.

In the augmented DFD used in CAPS, hierarchical consistency of both input/output and timing must be maintained. As shown in Figure 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 regeneration 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.

DESIGN OF GRAPHICAL EDITOR

The design and development of a prototype graphical editor for the CAPS project was undertaken at the Naval Postgraduate School. The results of that effort are now described[13]. First, general editor requirements are stated. Next, considerations are presented about the user interface and input/output. Finally, implementation of the main processing algorithm for the prototype editor is described in some detail. In addition, some comments are provided about current and future development.

Requirements for graphical editor

The graphical editor must meet the following general requirements[13]:

- Run in a windowed environment — controls 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 must 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.

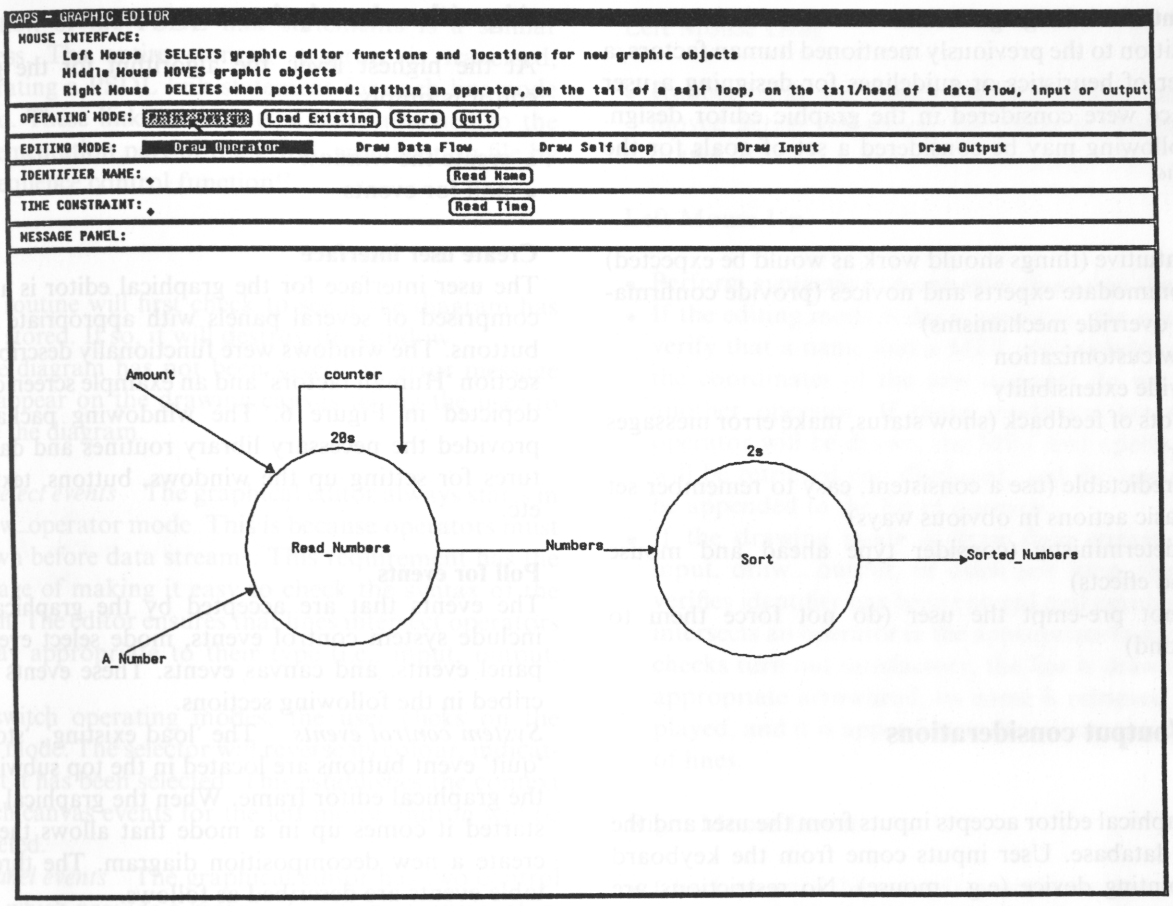
Interface design

A screen image of the graphic editor user interface is shown in Figure 6. Four factors influenced the design of the user interface[13]:

- the choice of machines on which to implement CAPS
- the choice of interface software support
- human-factors issues
- user-interface design guidelines

Sun Workstation
The selection of a graphic workstation for implementing a CAPS graphic editor was based both on the requirements previously stated and the availability of hardware at the test site. The Sun Workstation met the requirements, was readily available, and was consequently selected for the development and implementation of CAPS. The availability of a dedicated central processing unit in a multitasking environment greatly enhanced the design team's ability to code, test, and debug their software. The Sun Workstation also provided a powerful integrated programming environment based on the Unix operating system.

Sun View[14]
The Sun View user environment available on the Sun Workstation was chosen to provide windowing support



CAPS - GRAPHIC EDITOR

MOUSE INTERFACE:
Left Mouse SELECTS graphic editor functions and locations for new graphic objects
Middle Mouse MOVES graphic objects
Right Mouse DELETES when positioned: within an operator, on the tail of a self loop, on the tail/head of a data flow, input or output

OPERATING MODE: Print Design | Load Existing | Store | Quit

EDITING MODE: Draw Operator | Draw Data Flow | Draw Self Loop | Draw Input | Draw Output

IDENTIFIER NAME: | Read Name

TIME CONSTRAINT: | Read Time

MESSAGE PANEL:

Figure 6. User-interface screen image

for the graphical editor. It supports interactive graphics-based applications with multiple overlapping windows, and each window can run a task independent of the other windows. Sun View also provides a general toolkit for building window-based applications.

Human factors

Human factors may be the most significant determinant of a successful user-interface design. Of the many human-factors performance issues in the literature, the following were specifically addressed in designing the graphical interface.

Functional principle Controls that are grouped according to their functionality are easier to learn and result in fewer errors[15]. The graphical editor's controls fall into three functional groups:

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

Sequence of use principle Controls that are organized in the same sequence that they are used eliminate the need to jump around and therefore minimize the amount of information the user must remember[15]. The controls of the graphical editor are organized to be used in a series of top-to-bottom sequences. The top panel provides context-sensitive help information about use of the tool. The second panel is used to control the basic system functions of printing, 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 selection panel. It is used to switch from drawing one type of object to another. After the selection is made, the user enters appropriate information. The input panels for these entries are therefore located immediately below the drawing selection panel. Below this is a message panel used to display error messages. The drawing canvas is a large work-area panel located immediately below the message panel. This ordering of controls always allows, but does not force, the user to operate in a top-to-bottom circular fashion as follows:

- select operator (optionally read HELP information)
- enter and read name
- enter and read time constraint
- ensure no errors have occurred
- draw object

Human memory capacity principle Studies have shown that humans have the capacity to remember 7+/−2 things at once[15]. Since this principle is revised downward, the graphic editor has been configured to allow decomposition into a maximum of seven operators. Also, the user's view of the system is decomposed into only five steps, simplifying the design method.

User-interface design guidelines
In addition to the previously mentioned human factors, a number of heuristics or guidelines for designing a user interface were considered in the graphic editor design. The following may be considered a set of goals for the design[16]:

- be intuitive (things should work as would be expected)
- 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)
- do not pre-empt the user (do not force them to respond)

Input/output considerations

Inputs
The graphical editor accepts inputs from the user and the design database. User inputs come from the keyboard and pointing device (e.g., mouse). 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. User inputs are treated as events that are context sensitive and fall into the following categories: system control events, mode select events, text panel events, and canvas events. These events will be described in more detail later.

When the graphical editor is used to edit an existing diagram, the tool interface retrieves the necessary reconstruction information from the design database. The graphical editor uses this information to reconstruct the diagram.

Outputs
The graphical editor employs both graphical and textual outputs to both the workstation display and to a file. 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 be displayed in the message panel. 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 subwindow border, is provided.

After a decomposition has been completed and the user wishes to store the design, the editor will generate the PSDL link statements along with additional information needed to reconstruct the display. The CAPS interface will store this information in the design database[17]. Additionally, the graphical image may be dumped to a bitmapped file for output to a printer.

Algorithm description

At the highest level, the algorithm for the graphical editor is simply:

- create the window
- poll for events

Create user interface
The user interface for the graphical editor is a window comprised of several panels with appropriate text and buttons. The windows were functionally described in the section 'Human factors' and an example screen display is depicted in Figure 6. The windowing package used provided the necessary library routines and data structures for setting up the windows, buttons, text panels, etc.

Poll for events
The events that are accepted by the graphical editor include system control events, mode select events, text panel events, and canvas events. These events are described in the following sections.
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 that allows the user to create a new decomposition diagram. The three selectable events are described as follows.

Load existing

- The routine 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[17].
- If the object is an operator, an operator storage element is created, filled in with its information, and 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 that serve as the source operator for input lines. The only fields of an EXTERNAL that get useful values are those pointing at the line list.
- Any object encountered during the load process that 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 that was read in.
- After all of the objects in the file being loaded have been read in, stored, and linked, the diagram is then drawn.

Store

- Information for diagram reconstruction is stored by doing a traversal of the storage structure and writing out the contents of each operator node immediately followed by the contents of each of its associated line nodes.

- Creation of the PSDL link statements is a similar process. The entire storage structure is traversed, generating a PSDL link statement for each line node found. These link statements will be attached to the implementation part of the PSDL specification file by the sequence control function[17].

Quit

- This routine 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.

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 colour, indicating that it has been selected. This establishes the context in which canvas events for the left mouse button will be interpreted.

Text panel events The graphical editor has two control panels, which provide a means of entering textual information.

Name panel

- The name panel allows the user to enter a name for an operator or a line.
- Selecting the read_name button causes the name panel to be read and the syntax checked.
- 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.

MET panel

- The MET panel works essentially the same as the name panel. The difference is that the routine checks to ensure 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.

Canvas events Below the control panels is a large workarea called the canvas. The following mouse events are handled by the editor when the mouse is in the canvas.

Left Mouse Down

- Capture the x and y coordinates of the position where the event occurred (the starting position of the object being drawn).

Left Mouse Drag

- Rubber-band the object. As the mouse pointer is moved across the screen, the object is erased and redrawn.

Left Mouse Up

- Perform syntactic and semantic checks on the object.
- If the editing mode is draw_operator, the routine will verify that a name and a MET are available and that the coordinates of the new operator do not overlap another operator. If these conditions are met, the operator will be drawn, the MET and operator name will be retrieved and displayed, and the 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, the routine verifies identifier has been entered and ensures the line intersects an operator in the appropriate fashion. If the checks turn out satisfactory, the line is drawn with an appropriate arrowhead, its name is retrieved and displayed, and it is appended to the source operator's list of lines.

Right Mouse Down

- Check 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 and the screen redrawn.

Current efforts

The CAPS interface and graphic editor are currently undergoing a major modification. The prototype mentioned in this paper is being ported to the X-windows environment and is being implemented using InterViews[18]. This version will implement the exploding and annotation views as well as focused slices discussed in the section 'Multiple views'. The need for consistency checking discussed in the section 'Maintaining consistency and correctness' will be partially accomplished by integrating the syntax-directed editor and the graphic editor, providing a single source for valid PSDL code. Many of the human-factors issues raised in the section 'User-interface design guidelines', not addressed in the prototype, will be implemented in the new version. A modification of the step-by-step design process will be introduced to greater facilitate expert use and modification of an existing design.

CONCLUSIONS

This paper presents the need for 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, promises a significant improvement in user productivity.

Research on the graphical editor, as it relates to PSDL, indicates that a prototype design can be developed with much greater ease using the graphical editor than with only the syntax-directed editor. Graphical editor capabilities will also greatly enhance prototype modification, presentation, and documentation for further development.

Computer-aided software engineering (CASE) tools using graphical representation of software are years behind similar tools used in business and industrial applications. As Yourdon writes "Just as the cobbler's children are the last to get shoes, programmers and system analysts have traditionally been the last to get the benefits of automation for their own work."[5]. In recognition that software development is requiring an ever greater portion of the available corporate budget, tools supporting graphical representations early in requirements and design must be used to improve the software process. A form of decision support for software design is needed[11]. That is, the management and presentation of information needed by software engineers to make qualitative design decisions. The CAPS interface and graphic editor are a step in that direction.

REFERENCES

1 **Luqi and Ketabchi, M** 'A computer aided prototyping system' *IEEE Software* Vol 5 No 2 (March 1988) pp 66–72

2 **Luqi, Berzins, V and Yeh, R** 'A prototyping language for real-time software' *IEEE Trans. Soft. Eng.* Vol 14 No 10 (October 1988) pp 1409–1423

3 **Luqi** 'Software evolution via rapid prototyping' *Computer* Vol 22 No 5 (May 1989) pp 13–25

4 **Luqi and Lee, Y** 'Interactive control of prototyping process' *Technical report NPS 52-89-014* Naval Postgraduate School, Monterey, CA, USA (1989)

5 **Yourdon, E** *Modern structured analysis* Yourdon Press, Englewood Cliffs, NJ, USA (1989)

6 **Diaz-Gonzalez, J and Urban, J** 'Prototyping conceptual models of real-time systems: a visual perspective' in *Proc. Twenty-Second Annual Hawaii Int. Conf. System Sciences* IEEE Computer Society Press, Los Alamitos, CA, USA (January 1989)

7 **Pressman, R** *Software engineering: a practitioner's approach* (second edition) McGraw-Hill, New York, NY, USA (1987)

8 **Luqi and Berzins, V** 'Rapid prototyping real-time systems' *IEEE Software* Vol 5 No 5 (September 1988) pp 25–36

9 **Lewis, T G, Handloser, F, Bose, S and Yang, S** 'Prototypes from standard user interface management systems' *Computer* Vol 22 No 5 (May 1989) pp 51–60

10 **Smedstad, T and Anderson, O** 'Introduction of the concept 'integrated projection illustrating'' *Soft. Eng. Notes* Vol 3 No 1 (January 1988)

11 **Barnes, P and Hartrum, T** 'A decision-based methodology for object-oriented design' in *Proc. IEEE 1989 Nat. Aerospace and Electronics Conf.* (May 1989)

12 **Szekely, P and Meyers, B** 'A user interface toolkit based on graphical objects and constraints' in *Proc. ACM OOPSLA Conf.* (1988)

13 **Thorstenson, R** 'A graphical editor for the computer aided prototyping system' *MS Thesis* Naval Postgraduate School, Monterey, CA, USA (December 1988)

14 **Sun Microsystems** *Sunview programmer's guide revision: A* Sun Microsystems, Inc., Mountain View, CA, USA (October 1986)

15 **Sanders, M and McCormick, E** *Human factors in engineering and design* (sixth edition) McGraw-Hill, New York, NY, USA (1987)

16 **Hopgood, F *et al.*** *Methodology of window management* Springer-Verlag, Berlin, FRG (1986)

17 **Raum, H** 'Design and implementation of an expert user interface for the computer aided prototyping system' *MS Thesis* Naval Postgraduate School, Monterey, CA, USA (December 1988)

18 **Linton, M, Vlissides, J and Calder, P** 'Composing user interfaces with InterViews' *Computer* Vol 22 No 2 (February 1989) pp 8–22