1989

Interactive Control of Prototyping Process

Luqi; Lee, Yuh-Jeng

Naval Postgraduate School

Luqi and Y. Lee, "Interactive Control of Prototyping Process", Technical Report NPS 52-89-025, Computer Science Department, Naval Postgraduate School, 1989.
https://hdl.handle.net/10945/65232

142

NPS52-89-025

NAVAL POSTGRADUATE SCHOOL

Monterey, California

INTERACTIVE CONTROL OF PROTOTYPING PROCESS

LUQI

YUH-JENG LEE

APRIL 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.

This report was prepared by:

[redacted]

LUQI
Assistant Professor
of Computer Science

Reviewed by:

[redacted]

ROBERT B. MCGHEE
Chairman
Department of Computer Science

Released by:

[redacted]

KNEALE T. MARSHALL
Dean of Information
and Policy Science

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) NPS52-89- 025	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 O&MN, Direct Funding NSF CCR-8710737
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)

INTERACTIVE CONTROL OF PROTOTYPING PROCESS (U)

12. PERSONAL AUTHOR(S)
LUQI, LEE, Yuh-jeng

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Progress	FROM Sept 88 TO Mar 89	1989 March	32

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Computer-Aided Prototyping, CAPS, User Interface.

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

We present the use of CAPS (Computer-aided Prototyping) for the interactive construction, execution, debugging, modification, and controlling of software prototypes. We discuss the current version of CAPS, explicate its user interface for monitoring and coordinating the prototype development process, and depict the functioning of the integrated software tools.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS	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.

☆ U.S. Government Printing Office: 1986—606-24.

1. INTRODUCTION

The goal of rapid prototyping is to expedite the model-building process for the intended system and to evaluate if the resulting prototype meets given requirements. Rapid prototyping is particularly suitable for software development of iterative and costly nature [Pressman-88]. When the requirements cannot be completely determined, or there are problems or uncertainties about the proposed systems, prototyping allows a model to be constructed and tested in an early stage of the development work. After testing the prototype, modifications can be made to amend the original design and a new, improved set of specifications are then assembled to be used in the coding phase. In other words, the testing and refinement of requirements are performed before the actual engineering phase of a product.

The CAPS system (Computer-Aided Prototyping System) [Luqi-Ketabchi-88] is designed and constructed to increase the degree of automation in prototype development. It uses the executable Prototype System Description Language (PSDL) [Luqi-Berzins-Yeh-88] and consists of an integrated set of software tools including *user interface*, *syntax-directed editor*, *graphic editor*, *execution support system*, *design database*, *software base*, and *design management system*. In this paper, we discuss the use of CAPS for the interactive construction, execution, debugging, modification, and controlling of software prototypes.

1.1. The CAPS Process

The four major stages in the CAPS process, i.e., prototype design, construction, execution, and debugging/modification, support the iterative prototyping lifecycle [Luqi-

Ketabchi-88]. The initial prototype design starts with an analysis of the problem and a decision about which parts of the proposed system are to be prototyped. Requirements for the prototype are then generated, either in English or some formal notation. These requirements may be refined by asking the user to verify their completeness and correctness. After the preliminary requirements analysis and design is completed, the construction of the prototype may begin. When supplied with the design, CAPS will guide the user to produce a PSDL prototype representing the specification and design of the intended system using its user interface [Raum-88]. This prototype is then fed to the *execution support system* which translates the PSDL specification into Ada code and evaluates its behavior. Lastly, debugging and modification, which utilize all the tools, are performed over the entire CAPS. The purpose of CAPS is not to design a system, but rather to test and validate that design.

The user interface of CAPS is responsible for sequence control throughout the prototype development and for the insurance of continuity of the various levels of refinement during prototype construction. This interface possesses the knowledge of the functions of all components within CAPS and is able to interpret what the user is doing at any time and to generate queries to find out what the user wants whenever the system is unsure of the user's intention.

1.2. Language Issues

A PSDL software prototype consists of operators, data streams, and timing and control constraints. Operators are the basic building blocks in PSDL and can represent functions or state machines. They can be triggered by the arrival of sporadic or periodic inputs. Sporadic data arrive at random time intervals, while periodic data arrive at fixed time

intervals. When triggered an operator will be fired and produce output based on input values and the value of an internal state variable in the case of a state machine [Janson-88]. Operators are called atomic if they can be found in the *software base*, otherwise they are called composite and must be decomposed with a data flow diagram.

A data stream represents the flow of data between two operators. This communication can be in the form of either a data flow stream or a sampled stream. A data flow stream can be thought of as a FIFO queue. The data in a data flow stream is never lost and is always acted on in the order of arrival. A sampled stream can be thought of as a single memory cell. This type of data can be used many times or written over before use, depending on the rate of its input and use. Data flow streams must be used when each piece of data represents a unique transaction.

Another major aspect of PSDL is timing and control constraints. The real-time nature of prototypes necessitates timing and non-procedural control constraints in PSDL. Each time critical operator contains a special element called *maximum execution time*, indicating the maximum time in which the operator can complete execution after it is fired. Control constraints specify conditional requirements for the firing of operators.

2. INTERACTIVE CONSTRUCTION OF EXECUTABLE PROTOTYPES

Given initial requirements, the *syntax-directed editor* and *graphic editor* guide the user through the production of a PSDL prototype, the *design database* stores the elements of the prototype being constructed, and the *software base* provides the capability to retrieve reusable Ada components [cf., Booch-87]. Below we describe functions of each of these

components in the CAPS process.

2.1. Accessing the Design Database

The *design database* is a hierarchical storage structure for the development of a PSDL prototype [Douglas-89]. This structure is initially implemented as a multiway tree with each node (i.e., a component of the PSDL specification) containing:

- PSDL Specification part
- PSDL Implementation part (graph or Ada code)
- Graphic Record (if implementation is graphic)
- PSDL Control Constraints part (if implementation is graphic).

The Specification part can be further divided to obtain the various elements of the specification. In particular, it can represent an operator name, inputs, outputs, states, or maximum execution time. The Implementation part consists of the link statements produced in the *graphic editor*, or written or retrieved Ada code. The Graphic Record is the data used only by the *graphic editor* that is used to redraw a the data flow diagram.

Each level of a tree is produced by the decomposition of the parent operator. The database is able to recognize the relation of parent and child. This enables queries of the type *find child* and *find parent* to be performed, as well as a search by operator name. Finally the *design database* is able to traverse the entire tree in breadth-first order to produce a PSDL software prototype.

Inputs to the Design Database are:

- Graphic Record (from *graphic editor*)

- PSDL Implementation (graph or Ada code)
- PSDL Control Constraints
- PSDL Specification

The *design database* outputs are the same as the inputs, except that a complete PSDL prototype is now added. The following operations are designed to enable the *design database* to aid in the construction and modification of a prototype.

- *Create Root Node*: allows for the creation of a tree of operators in the database.
- *Create Child Node*: creates a new node for information storage and sets the parent-child relationship between this new node and its parent.
- *Store Property*: stores a PSDL part (Specification, Implementation or Control Constraints), subpart (Operator Name, Input List, Output List, State List or Maximum Execution Time), or Graphic record in the named node.
- *Get Property*: retrieves the above mentioned properties from the *design database*.
- *Get Children*: returns the names of all the children of the named operator.
- *Delete Node*: removes the named operator from the *design database*. Because of the hierarchical nature of the *design database*, this operation will effectively remove the entire subtree that is rooted at the named operator.
- *Traverse Tree*: performs a breadth-first traversal of the *design database* that collects the PSDL components into a single program.

2.2. Drawing Graphic Diagrams

The *graphic editor* is a graphics tool for drawing enhanced data flow diagrams in the PSDL computational model [Luqi-Berzins-Yeh-86]. It is the part of CAPS where most of the input of a prototype description is performed. The decomposition of a PSDL Operator into lower level operators defines the actual creation of new nodes in a tree structure. The names of all operators and data streams are entered at this point. The editor insures a valid decomposition by checking the consistency of inputs, outputs, states and maximum execution times. The *graphic editor* can also show the data flow diagram of the parent operator to aid the user in retaining the place of a single operator in the prototype [Thorstenson-88].

Inputs to the *graphic editor* include the operator name, input, output, and state lists and the maximum execution time. Outputs from the *graphic editor* are the PSDL link statements and the Graphic Record. The operations performed by the *graphic editor* include drawing operators data streams, inputs, and outputs showing a parent data flow diagram, and loading and storing the Graphic Record.

2.3. Generating Text String Part of Prototypes

The *syntax-directed editor* in CAPS produces a syntactically correct PSDL specification and performs syntax checks on existing PSDL files [Teitelbaum-Reps-81, Portor-88]. This specification consists of two parts, an enhanced data flow diagram and a non-procedural control constraint part. The *syntax-directed editor* reads in and completes partial PSDL specifications and produces PSDL control constraints.

The *syntax-directed editor* accepts as input the partial PSDL specifications that are produced in the *graphic editor* via user interface and outputs a syntactically correct PSDL specification, including control constraints.

2.4. Retrieving Templates from the Software Base

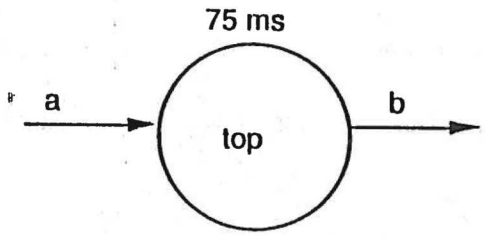
The *software base* is a database of reusable Ada components that are indexed and searched for based on PSDL specifications. It has two parts: a query module and a maintenance module. The query module receives the PSDL specification part and returns one or more Ada modules that meet those specifications, if search is successful. The maintenance module is involved with the creation and upkeep of the database. All records must be stored by PSDL specification so that they can later be searched by the same specification. The *software base* schema is constructed based on an object-oriented database and its management system. The feasibility of such an approach is illustrated in [Galik-88].

2.5. The User Interface for Interactive Control

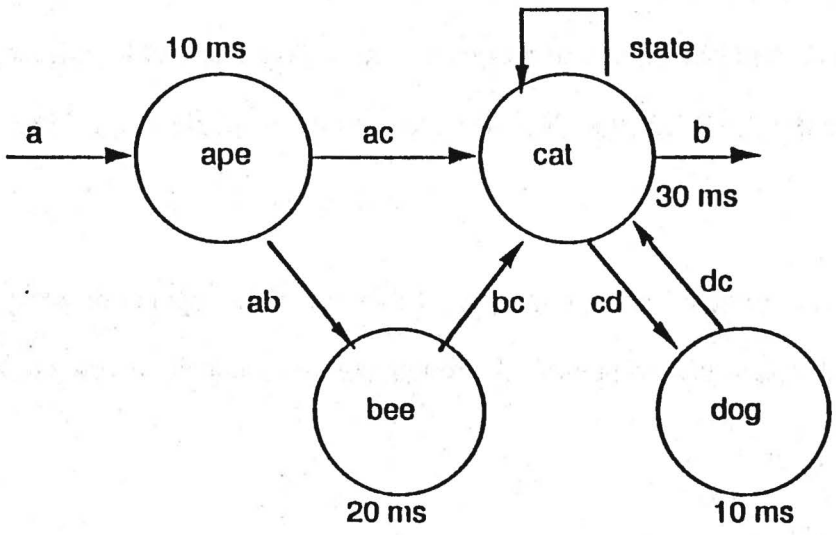
The user interface has two main functions during the construction of the prototype: sequence control of the construction effort, and the insurance of consistency of the level-to-level decomposition of the operators. The sequence control is performed by utilizing the if-then-else logic of the Bourne Shell [Sun-86]. The consistency of the decomposition is harder to achieve.

In the decomposition of an operator, a number of child operators are produced through the use of the data flow diagram. Although this decomposition can produce any

number of new operators with any number of data streams between them, the inputs and outputs of the system of child operators must be exactly the same as those of the parent. The *graphic editor* can insure this by reading in an input and output list. It will not allow any other inputs or outputs and if all these inputs and outputs are not utilized the user will be notified that the decomposition is not valid. Additionally the *graphic editor* can check the semantic consistency for PSDL specifications at different levels, e.g., ensuring the maximum execution time for any of the children of an operator does not exceed that of the parent and inheritance constraints for external stream types. Finally, the *graphic editor* helps to enforce the correctness of a PSDL specification, i.e., a state variable in a child must also exist in the data flow diagram decomposition as a self loop or a internal connection between two operators. For example, given the diagram with operator *top*:



the following is a valid decomposition (since it has the same inputs and outputs as the above diagram), with 4 lower level operators *ape*, *bee*, *cat*, and *dog*:



The *graphic editor* will also produce the link statements below, for inclusion in the PSDL implementation:

```
a.EXTERNAL-->ape
ab.ape:10ms-->bee
ac.ape:10ms-->cat
bc.bee:20ms-->cat
b.cat:30ms-->EXTERNAL
state.cat:30ms-->cat
cd.cat:30ms-->dog
dc.dog:10ms-->cat
```

As previously stated, the four operators *ape*, *bee*, *cat*, and *dog* represent the four child nodes of the node *top* in the *design database*. These nodes are created, but the name of these operators is not the only thing known about them. The link statements can be used to determine the inputs, outputs, names of any state variables, and maximum execution times of these operators. The user interface reads the link statements and determines all of the information required to produce a partial specification. The specification has the names but not the types of the data streams. Production of this specification helps to ensure error free PSDL prototypes by relieving the user of the need to remember what he has previously entered. It also requires data to be entered only once.

There is one additional place where the user interface creates part of the PSDL prototype. The Implementation part of PSDL consists of link statements followed by a data stream list. This list consists of the internal data streams that have been drawn in the enhanced data flow diagram. The user interface appends the data stream list to the end of the link statements. To complete the Implementation part of the PSDL operator, the type of each of these data streams is added automatically using the *syntax-directed editor*.

3. EXECUTION MANAGEMENT OF PROTOTYPE SYSTEMS

Prototype execution utilizes the *translator*, *static scheduler*, and *Dynamic Schedule* to produce an executable prototype in Ada, that can test the design and requirements of the actual system.

The *translator* in CAPS translates the PSDL prototype into Ada. This is done by taking the Ada implementation of the atomic operators and adding the control constraints of the composite operators to produce a group of loosely coupled Ada modules [Altizer-88]. The input to the *translator* is the PSDL prototype that was produced in the breadth first traversal of the *design database*. The output is the package of Ada modules.

The *static scheduler* produces a schedule of time critical operators, if can be done [Marlowe-88, O'hern-88]. If it is impossible to produce a valid schedule because of the timing constraints set in the construction of the prototype, the user will be notified by the *debugger*. We discuss this condition below in the section of debugging and modification. process will be described in the debugging and modification section.

The *dynamic scheduler* produces a dynamic schedule which integrates the static schedule with time critical operators, a collection of non-critical operators and an exception handler in the *debugger*. The *dynamic scheduler* adds the ability to run non-time critical operators in conjunction with the static schedule [Wood-88].

The produced dynamic schedule is a Ada program that consists of two major tasks and the exception handler. The higher priority task is the schedule of time critical operators. This task will execute until reaching a designated milestone in the schedule. If it is ahead of

the schedule, the secondary task (non-time critical operators) will be executed for the amount of excess time. In the event that the prototype falls behind its time schedule at any milestone, an exception will be triggered and control is passed on to the *debugger*.

To aid in debugging, a trace and a graphical representation of the prototype being executed are planned. The trace will list the name of the operator and the time when it is entered. This information is critical when evaluating the actual real-time performance of the prototype. The run-time status of the prototype will be displayed by presenting the user with a tree that represents the nodes of the *design database*. The nodes on the frontier of the tree that correspond to the operators currently executing will be highlighted. This allows the user to monitor the run-time behavior of the prototype.

4. PROTOTYPE DEBUGGING AND MODIFICATION

In the above sections, we discuss the construction and execution of prototypes. During the execution, most of the effort will be placed on debugging the the prototypes, with all the correctness checking by the tools.

4.1. Run-Time Debugging

The debugging of the prototype takes place during the execution of the *static scheduler*, while the static schedule is being produced, and during the execution of the dynamic schedule of the prototype. The *debugger* must be broken into two parts because exceptions caused by static problems arise before compilation, while many of the dynamic timing problems of a real-time system will not occur until the prototype has been compiled

and is executing. [Wood-88]

The *debugger* has two basic functions for correcting errors in the prototype. The first is through direct user interaction with the prototype and the second is through the *syntax-directed editor* and the *graphic editor* in the modification mode.

The *debugger* gives the user a chance to make small changes to the prototype in the *execution support system*. This allows rapid feedback as to the results of the change. The problem with this method of modification is that these changes are temporary, although they will be recorded in the *design database* and available for user review during modification. In other words, the only way to make a permanent change to the prototype is with the *syntax-directed editor* or *graphic editor* through the user interface. This is because the correctness check can be done only through these tools.

4.2. Modifying the Prototypes

There are many problems involved when modifying a PSDL prototype. These problems stem from the fact that operators in the hierarchical structure of the *design database* inherit information both up and down. In addition there are both graphical and textual views of an operator. These views actually hold different versions of the same information. A change in one view requires a change in the other. A detailed discussion on the modifications of PSDL in terms of designer's and tool's views can be found in [Luqi-89].

4.2.1. Modifying an Operator

If an operator is deleted, a simple solution is to delete the entire subtree that has that operator as a root. This action is very severe and the *design database* should record a historical version of the prototype at this time. If it is later shown that this deletion was an improper choice, this version of the prototype can be restored. A deletion also requires the modification of the data flow diagram and link statements of the parent operator, where the deleted operator is first defined.

The addition of a new operator requires similar action. The new operator is added to the *design database* tree and the construction mode of the user interface is entered. Construction continues until the new subtree is completely defined. In both deletion and insertion the data flow diagram of the parent operator must be modified to reflect the changes in its subtree.

The most significant problem in modifying an operator occurs when small changes in the specifications or control constraints of an operator are made. A change in the specification could cause changes in every node of its subtree. Similarly, a specification change could cause an atomic operator to become a complex operator if the search of the *software base* no longer yields a match. The search on modified specifications may yield a match that was not previously obtainable, therefore deleting a subtree and replacing it with an atomic operator.

This level-to-level consistency problem can move up the tree as well. In addition, a change at a child node may cause it to be different from the node required for the decomposition of the parent.

4.2.2. Consistency of Views

A change of a textual component of PSDL may be propagated to the graphic representation. For example, when the name of a data stream is changed in the implementation section of a PSDL operator, the name change must also be reflected in the link statement and also in the graphic record. The graphical view of a change might be the best indication of the problems caused by deletions or changes. More effort is required in the area of prototype modification, if the same assurances of valid PSDL prototypes that are present in the construction mode are expected during modification.

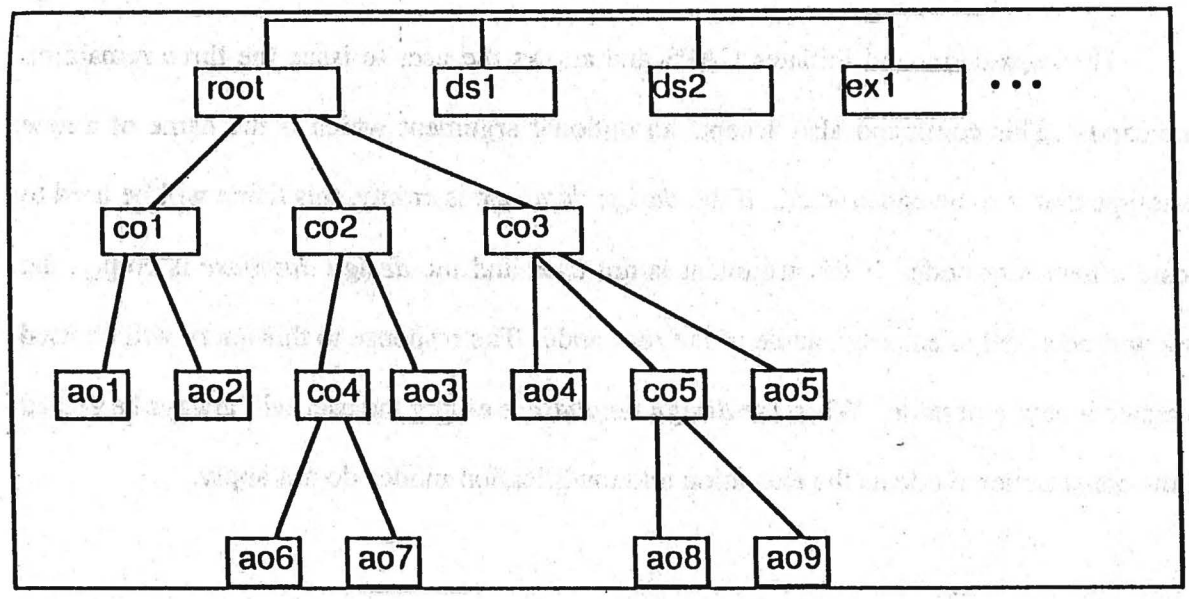
5. SEQUENCE CONTROL DURING PROTOTYPE DEVELOPMENT

The construction of PSDL prototype is done by recursively invoking the ***construct*** command of the user interface (detail in section **6**). The user interface controls a loop of traversal which continues to search the tree for nodes in the *design database* without an implementation part. The first incomplete node found in the *design database* and its specification are used to search the *software base*. If a match is found, that node is considered atomic and the Ada code is placed in the implementation section of that operator. If there is no match the user is asked to either decompose or write the Ada implementation. If hand coding is done, this operator is again atomic and the Ada code becomes the implementation part. Finally, if the user chooses to decompose the operator, the *graphic editor* draws the data flow diagram and produces the link statements. The user interface reads these link statements and writes the partial specification for all newly created operators. New nodes in the *design database* are created for each new operator. The *syntax-directed editor* is then invoked to complete the PSDL for the original composite

operator.

The construct loop ends when all leaf nodes of the *design database* are atomic. During the creation of a prototype, a rapid growth in the number of nodes is expected, as the high level operators are decomposed. Eventually the Ada implementation for the lower level operators would be found in the *software base* and the growth of the tree stops.

The construction process deals only with the production of operators. New data streams are produced in each operator. If these data streams are not atomic they must be defined in PSDL. All user defined data streams (ds) appear outside the tree of operators on a level in the *design database* equal to the root operator. Exceptions (ex) also appear at this level, which is similar to a global type definition in Pascal. The following figure illustrates this structure. The tree of operators contains both composite operators (co) and atomic operators (ao).



During execution phase the *translator* and *static scheduler* may be invoked in any order, or simultaneously in a multitasking environment. After the Static Schedule is produced and a non-time critical operators identified, these operators must be grouped in a package for use in the Dynamic Scheduler. The *translator* output is compiled and used in both the Static Schedule and the non-time critical package. These two packages then become part of the Dynamic Schedule, which must be compiled and linked before it is executed.

6. Top Level User Interface Commands

At the top level, CAPS accepts four commands: ***caps***, ***construct***, ***execute*** and ***modify***. This section describes these commands and the environment the user will be in when these commands are executed. The principles of CAPS interface design is "simplicity."

6.1. The *caps* Command

The ***caps*** command initiates CAPS and allows the user to issue the three remaining commands. This command also accepts an optional argument which is the name of a new prototype that is to be constructed. If the *design database* is empty, this name will be used to create a new root node. If the argument is not used and the *design database* is empty, the user will be asked to enter the name of the root node. The response to this query will be used to create a new root node. When the *design database* is empty the user will always be placed in the construction mode as the execution and modification modes do not apply.

6.2. The *construct* Command

The ***construct*** command is used to construct a prototype. In this mode the user is directed into the Syntax-Directed Editor and *graphic editor* to create the PSDL program.

This command will place the user in the location where the PSDL construction can begin or continue. The process is monitored by CAPS to insure the production of a complete and valid PSDL specification. Other than the manipulations of the two editors, the search of the *software base*, the storage and retrieval of components in the *design database*, and the semantic checking of PSDL prototypes are all transparent to the user.

The user is advised of the results of the software search and the completion of the construction with the below dialogs:

- *Software Search Complete – no match found.* This notifies the user that the search for an Ada implementation for the given specification was unsuccessful. This would be followed by the question: *Do you want to decompose, y or n.* Based on the response the user will be placed in the *graphic editor* or Ada editor.

- *Software Search Complete – implementation found.* This indicates a successful retrieval of an Ada implementation. The user is then asked to choose the next operator for implementation.

- *Select the next operator for implementation.* This dialog presents the user with a list of incomplete operators. The user then enters the name of the desired operator. This question will follow the completion of any implementation.

- *Construction Complete.* This message indicates the completion of the PSDL prototype. The user is then placed in the user interface portion of CAPS where execution or modification can be selected.

6.3. The *execute* Command

The ***execute*** command places the user in the ***execution support system*** interactive where the constructed prototype is executed to test the real-time performance. This command first checks for the existence of a completed prototype. A warning message *No Completed Prototype Available* will be issued if there a PSDL prototype cannot be found. When a complete prototype is available, the *translator*, *static scheduler* and *Dynamic Schedule* are called in succession. The use of these components, as well as the Ada compiler and linker, is transparent to the user. The user is informed of the status within the ***execution support system*** with the below messages.

- Translation Complete.
- Static Scheduler Complete.
- Dynamic Scheduler Complete.
- Compilation Complete.
- Linking Complete.
- Execution Complete.

In the event of a problem in the scheduling or execution of the prototype, the user will be notified by the *debugger*. The user has the option to make temporary corrections to the prototype in an attempt to achieve proper execution. All permanent changes must be

made in the appropriate editor through the use of the modify command.

6.4. The *modify* Command

The ***modify*** command is used to make changes to the prototype. The user is placed in the modification mode that insures that all changes are made consistently throughout the various levels in the *design database*. This command asks the user for the name of the operator to be modified. The user must then use the *syntax-directed editor* or *graphic editor* to make the required changes to the operator. The user interface insures that the appropriate changes are made in the higher and lower levels of the *design database*. The user will be asked to resolve the conflicts that will arise as these changes are carried out. If necessary, the user may be required to enter the construction mode to complete the modified prototype.

7. CONCLUSIONS AND FUTURE WORK

We have presented the user interface that supports the interactive construction, execution and modification of executable prototypes. First we defined the requirements of a CAPS interface and then designed the interface that meets these requirements. All the important issues related to the user interface were further tested via outlined implementation. This interface has shown great promise in the demonstration of the feasibility of most of the components of CAPS. This computer-aided prototyping tool is ideal in prototyping the production of real-time embedded systems, and it is easy to handle and requires only minimum user training.

CAPS has demonstrated the potential as a significant time– and cost–savings tool in

the development of software systems. The primary benefit of incorporating the user interface as part of CAPS is that it helps CAPS develop into a more powerful and advanced form. The user interface of CAPS is different from a conventional one in that it is also a tool manager, task sequencer, and real-time dispatcher. It is designed as an expert system that is capable of monitoring all phases of prototyping software systems through the interaction with users. This type of interface may be useful for other software tools to increase the degree of automation of such tools [Barstow-84]. One may also employ the prototyping methodology to construct a user interface [Lewis-*et.al.*-89]. Another benefit of our approach is that a tool that uses both graphical and textual data entry and display can utilize the user interface control and achieve data consistency between the two views more easily [Chang-86, Dumas-88, McDermid-85, Sanders-McCormick-87].

The advanced areas that are expected to substantially improve the capabilities of CAPS in future research are as follows:

(1) Primary Data Entry. The original design of CAPS called for the majority of data entry to be done in the *syntax-directed editor*. In the process of system development, the *graphic editor* has proven to be the primary tool for the entry of new data. In developing PSDL prototypes, new operators and data streams are first defined in the enhanced data flow diagram. The information from the *graphic editor* is used by user interface to produce the partial specification of the newly constructed operators and the data stream declarations. The current version of the *graphic editor* only names data streams, whose types of must be added using the *syntax-directed editor*. The inclusion of types in the *graphic editor*, and in link statements, would eliminate the need to return to the *syntax-directed editor* to complete the specifications and data stream lists.

(2) Execution Monitoring. The current version of CAPS does not include a means of monitoring the execution of a prototype. Some means of producing both a trace and a view of the execution of a prototype would greatly improve the ability to debug and verify the prototype's performance. The ability to trace a problem in the execution in relation to the original requirements would aid in the validation or modification of these requirements.

(3) Prototype Modification. A desirable modification mode is one that not only allows changes to a prototype, but provides the same assurances of valid PSDL prototypes as the construction mode does. To do that, a debugger will have to identify the parts of the prototype that are not performing correctly and provide fixes so that the prototype's behavior will be consistent with the system requirements. Such a debugger would be similar to that discussed in [Dershowitz-Lee-87]. With the addition of such modification system in CAPS, one will be able to perform rapid debugging during prototype development.

REFERENCES

Altizer, C., Implementation of a Language Translator for a Computer Aided Prototyping System, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.

Barstow, D., *et.al.*, Interactive Programming Environments, McGraw-Hill Book Company, 1984.

Booch, G., Software Engineering with Ada, 2nd ed., Benjamin Cummings Publishing Co., Inc., 1987.

Chang, S.-K., *et.al.*, Visual Languages, pp. 155-157, Plenum Press, 1986.

Dershowitz, N., Lee, Y., Deductive Debugging, in IEEE Proceedings of the Symposium on Logic Programming, San Francisco, California, September 1987.

Douglas, B., Design Database Schema for the Computer Aided Prototyping System, Master's thesis, March 1989.

Dumas, J., Designing User Interfaces for Software, Prentice-Hall, 1988.

Galik, D., A Conceptual Design of a Software Base Management System for the Computer Aided Prototyping System, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.

Janson, D., Luqi, A Static Scheduler for the Computer Aided Prototyping System, in Proceedings of COMPSAC 88, pp. 92-98, Gaithersburg, MD, June 1988.

Lewis, T. G., *et.al.*, "Prototypes from Standard User Interface Management Systems", Proceedings of the 22nd Hawaii International Conference on System Science, Kona, Hawaii, January 1989.

Luqi, Berzins, V., "Rapidly Prototyping Real-Time Systems," IEEE Software, v. 5, pp. 25-36, September 1988.

Luqi, Berzins, V., and Yeh, R., "A Prototyping Language for Real-Time Software," IEEE TSE, October 1988.

Luqi, Ketabchi, M., "A Computer Aided Prototyping System," IEEE Software, v. 5, pp. 66-72, March 1988.

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

Marlowe, L., A Scheduler for Critical Timing Constraints, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.

McDermid, J., Intergrated Project Support Environments, Peter Peregrinus Ltd., 1985.

O'Hern, T., A Conceptual Level Design for a Static Scheduler for Hard Real- Time Systems, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1988.

Porter, S., A Design of a Syntax-Directed Editor for CAPS, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.

Pressman, R., Software Engineering: A Beginners Guide, pp. 1-93, McGraw- Hill Book Company, 1988.

Raum, H., Design and Implementation of an Expert User Interface for the Computer-Aided Prototyping System, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.

Sanders, M., and McCormick, E., Human Factors in Engineering and Design, McGraw-Hill Book Company, 1987.

Teitelbaum, T., and Reps, T., "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," CACM, v. 24:9, pp. 563-573, September 1981.

Thorstenson, R., A Graphical Editor for the Computer Aided Prototyping System, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.

Wood, M., Run-Time Support for Rapid Prototyping, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Office of Naval Research Office of the Chief of Naval Research Attn. CDR Michael Gehl, Code 1224 800 N. Quincy Street Arlington, Virginia 22217-5000	1
4.	Space and Naval Warfare Systems Command Attn. Dr. Knudsen, Code PD 50 Washington, D.C. 20363-5100	1
5.	Ada Joint Program Office OUSDRE(R&AT) Pentagon Washington, D.C. 20301	1
6.	Naval Sea Systems Command Attn. CAPT Joel Crandall National Center #2, Suite 7N06 Washington, D.C. 22202	1
7.	Office of the Secretary of Defense Attn. CDR Barber STARS Program Office Washington, D.C. 20301	1
8.	Office of the Secretary of Defense Attn. Mr. Joel Trimble STARS Program Office Washington, D.C. 20301	1
9.	Commanding Officer Naval Research Laboratory Code 5150 Attn. Dr. Elizabeth Wald Washington, D.C. 20375-5000	1

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

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

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

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

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

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

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

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

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

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