1991

CAPS as a requirements engineering tool

Luqi

CAPS AS A REQUIREMENTS ENGINEERING TOOL

Luqi
Robert Steigerwald
Gary Hughes
Valdis Berzins

Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

Abstract.[1] The process of determining user requirements for software systems is often plagued with uncertainty, ambiguity, and inconsistency. Rapid prototyping offers an iterative approach to requirements engineering to alleviate the problems inherent in the process. CAPS (the Computer Aided Prototyping System) has been built to help software engineers rapidly construct software prototypes of proposed software systems. We describe how CAPS as a prototyping tool helps firm up software requirements through iterative negotiations between customers and designers via examination of executable prototypes.

1. Introduction. A major problem with the traditional waterfall lifecycle approach is the lack of any guarantee that the resulting product will meet the customer's needs. In most cases the blame falls on the requirements phase of the lifecycle. Yourdon [Your89] cites studies that indicate 50% of errors or changes required in a delivered software product and 75% of the total cost of error removal are the results of inadequate, incorrect, or unstated requirements specifications. Often users will be able to indicate the true requirements only by observing the operation of the system. Unfortunately, the traditional life cycle yields executable programs too late in the software engineering process, at a point where major change is prohibitively expensive [Boar84].

To alleviate the problems inherent in requirements determination for large, parallel, distributed, real-time, or knowledge-based systems, current research suggests a revised software development life cycle based on rapid prototyping [BL88, Berz90, TY89]. As a software methodology, rapid prototyping provides the user with increasingly refined systems to test and the designer with ever better user feedback between each refinement. The result is more user involvement and ownership throughout the development/specification process, and consequently better engineered software [Ng90].

1. This research was supported in part by the DoD Ada Joint Program Office under grant number DWAM10100 (Ada Technology Insertion Program) and by the National Science Foundation under grant number CCR-9058453.

1991 ACM 0-89791-445-7/91/1000-0075

2. The Computer Aided Prototyping System (CAPS).
The problem with requirements engineering is amplified in the case of hard real-time systems, where the potential for inconsistencies is greater [Beam89, BA91, NGCR, SR88]. One of the major differences between a real-time system and a conventional system is required precision and accuracy of the application software. The response time of each individual operation may be a significant aspect of the associated requirements, especially for operations whose purpose is to maintain the state of some external system within a specified region. These response times, or deadlines, must be met or the system will fail to function, possibly with catastrophic consequences. These requirements are difficult for the user to provide and for the analysts to determine. Toward this end, an integrated set of software engineering tools, the Computer Aided Prototyping System [LK88], has been designed to support quick prototyping of such complex systems by using easy to understand visual graphics [PDW89] mapped to a tight specification language, which in turn automatically generates executable Ada [Booc87, Gonz91, Ada83] code. The main components of CAPS are the prototype system description language (PSDL), user interface, software database system, and execution support system (see Fig. 1).

2.1 Prototype System Description Language (PSDL).
The prototype system description language (PSDL) [LBY88] is the key component of CAPS. It serves as an executable prototyping language at a specification or design level and has special features for real-time system design. The PSDL model is based on data flow under real-time constraints and uses an enhanced data flow diagram that includes non-procedural control and timing constraints.

2.2 User Interface. The graphic editor, in the User Interface, is a tool which permits the user/software engineer

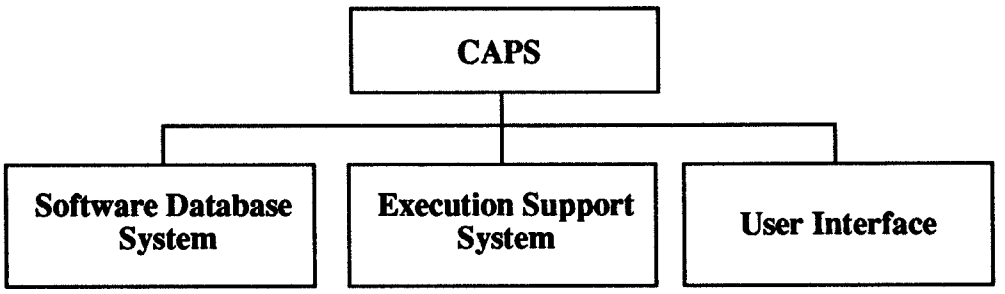

Fig. 1 High Level Structure of CAPS

to construct a prototype for the intended system using graphical objects to represent the system [LVC89, TAE]. The current version of the user interface uses TAE+ to generate the windows and buttons in the user interface. The graphical editor is implemented by augmenting the Idraw editor provided by InterViews. The graphical objects presented to the designer include operators, inputs, outputs, data flows, and operator loops. The syntax directed editor is used by the user/software engineer to enter additional annotations to the graphics. A browser allows the analyst to view reusable components in the software base. An expert system provides the capability to generate English text descriptions of PSDL specifications. Together, these tools facilitate common understanding of PSDL components by users and software engineers alike, thereby reducing design errors.

2.3 Software Database System. The software database system provides reusable software components for realizing given functional (PSDL) specifications, and consists of a design database, software base, and software design management system.

The design database [Nest86] contains PSDL prototype descriptions for all software projects developed using CAPS. The software base contains PSDL descriptions and implementations for all reusable software components developed using CAPS. Prototyping with the software base speeds up evolution by providing many different versions of commonly used components [SLM91], making it easier to try out alternative designs. The software design management system manages and retrieves the versions, refinements and alternatives of the prototypes in the design database, as well as the reusable components in the software base.

2.4 Execution Support System. The execution support system [Bori89] consists of a translator, a static scheduler, a dynamic scheduler, and a debugger. The translator generates code that binds together the reusable components extracted from the software base. Its main functions are to implement data streams, control constraints, and timers. The static scheduler allocates time slots for operators with real time constraints before execution begins. If the allocator succeeds, all operators are guaranteed to meet their deadlines even with the worst case execution times. If the static scheduler fails to find a valid schedule, it provides diagnostic information useful for determining the cause of the difficulty and whether or not the difficulty can be solved by adding more processors. As execution proceeds, the dynamic scheduler invokes operators without real-time constraints in the time slots not used by operators with real-time constraints [Mok85]. The debugger allows the designer to interact with the execution support system. The debugger has facilities for initiating the execution of a prototype, displaying execution results or tracing information of the execution, and gathering statistics about a prototype's behavior and performance.

3. CAPS as a Requirements Engineering Tool.

3.1 Prototyping. The Computer Aided Prototyping System (CAPS) is used to create software prototypes, which are mechanically processable and executable descriptions of simplified models of proposed software systems. It is also used to modify these models frequently in an iterative prototype evolution process for the purpose of firming up the requirements. Fig. 2 illustrates the prototyping process which consists of two stages: prototype construction and code generation [Luqi89].

Prototype construction is an iterative process that starts out with the user defining the requirements for the critical aspects of the envisioned system. Based on these requirements, the designer then constructs a model or

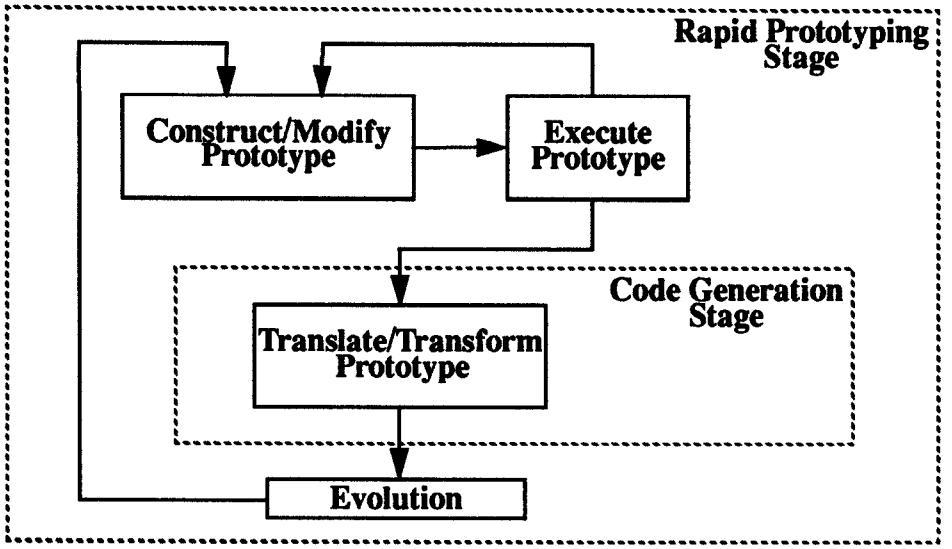

Fig. 2 Rapid Prototyping Process

prototype of the system in a high-level, prototype description language and examines the execution of this prototype with the user. If the prototype fails to execute properly, the user then redefines the requirements and the prototype is modified accordingly. This process continues until the user determines that the prototype successfully meets the critical aspects of the envisioned system. Following this validation, the designer uses the validated requirements as a basis for the design of the production software.

The code generation stage focuses on transforming and augmenting the prototype to generate the production code. Prototypes are built to gain information to guide analysis and design, and support automatic generation of the production code.

To create production code from a prototype, it may be necessary to clean up the decomposition, add missing functions, and optimize performance. Prototypes go through many changes in the prototype construction stage, so that the structure of the final version may partially reflect past versions of the requirements that were proposed and rejected. Once the requirements and the desired behavior for the prototype have stabilized, it is useful to transform the structure of the prototype to simplify the decomposition and to remove features that are no longer supported by the final version of the requirements.

A prototype may not implement all of the functions of the proposed system, since the prototyping effort is focused in the aspects of the requirements that are unknown or uncertain. After the requirements have stabilized, the design and the structure of the prototype must be augmented to account for these additional functions. These augmentations can be expressed in the prototyping language to provide an early check on the adequacy of the final version of the system structure.

A prototype may not meet all of the performance requirements, or may not operate in the same hardware and software environments as the proposed system. The structure of the prototype may have to be transformed to optimize its performance and to account for differences between the host environment for the prototype and the operating environment for the proposed system. It is desirable to record the desired transformations as annotations on the prototype, and to generate the transformed decomposition automatically based on the annotations. Such an approach preserves the structure of the prototype prior to optimization, so that a version of the prototype with this structure can help to evaluate system changes that are proposed after the system is placed in production. The unoptimized version of the prototype is better suited for modification because the optimization transformations generally complicate the structure of the design and destroy the independence of its parts, thus making future modifications more difficult. This approach may provide the benefits of rapid prototyping in both the requirements analysis and system maintenance activities.

An example showing how to use a prototype to test requirements for future "maintenance" modifications can be found in [Luqi89]. In the example, a later modification to the requirements was evaluated using the original prototype. An on-line CAPS tool supports the idea and provides robust syntactic and semantic help for such activities.

3.2 Domain Specificity and Requirements Traceability. Using CAPS to engineer requirements offers clear advantages over determining requirements manually. The prototype system description language is focussed on the domain of hard real-time systems and as such offers a common baseline from which users and software engineers describe requirements. Defining requirements in a *domain specific* language results in more efficiency and fewer errors because it constrains the way users and engineers can describe a particular requirement. In addition, the interpretations of requirements stated in a domain specific language such as PSDL are unambiguous, whereas requirements stated in English are often misunderstood.The discipline imposed on analysts by using a domain-specific requirements language is analogous to the discipline imposed on software designers and implementors by using Ada. In both cases, the use of formal notations helps to expose incompletely thought-out ideas and missing aspects of the documents under development.

In most software engineering efforts requirements are volatile, changing often over the course of the software development. Requirements traceability is essential to accurately map changed requirements into the implementation. CAPS offers basic requirements traceability through the "by requirements" statement in the PSDL grammar. This statement allows software engineers to associate actual requirements with the definitions of module interfaces and constraints by annotating the interface or constraint definition with an identifier. This method allows engineers using the design database to readily locate the modules and the portions of each interface that implement a particular requirement and make the appropriate changes during the evolution of the prototype. This feature offers substantial savings over manual methods of requirements tracing.

3.3 Requirements Engineering. The requirements for a software system are expressed at different levels of abstraction and with different degrees of formality. The highest level requirements are usually informal and imprecise, but they are understood best by the customers. The lower levels are more technical and more precise, are better suited for the needs of the system analysts and designers, but they are further removed from the users' experiences and less well understood by the customers. Because of the differences in the kinds of descriptions needed by customers and developers, it is not likely that any single representation for requirements can be the "best" one for supporting the entire prototyping process.

During the process of stabilizing the requirements via prototyping, it is necessary to repeatedly move from high-level requirements to details of system behavior, and from system behavior back to high-level requirements. The prototype designers must guess the intentions of the customers based on their informal statements, and embody their vision in a prototype design that can be demonstrated to the users. This process is imperfect, and the demonstrated behavior will help the customers identify differences between what they need and how the analysts interpreted their requests. When a bug in the system behavior is discovered, it must be traced back to the requirements to identify the specific guesses proposed by the analysts that are inaccurate. After the faulty decisions have been identified and new versions have been proposed, it is necessary to trace the effects of the change back down the refinement structure to find the parts of the prototype design that are affected, so that they can be adjusted and the next approximation to the requirements can be demonstrated.

In the context of prototyping, the requirements are used as a means for bridging between the informal terms in which users and customers communicate and the formal structures comprising a prototype. We believe that a useful representation for this information is a hierarchical goal structure, where informal customer goals are refined and defined by several levels of increasingly formal and precise subgoals, with different notations used at different levels. We expect natural language to be used at the highest levels, and the protoyping language to be used at the most detailed levels, with mixtures and possibly several additional notations appearing in the intermediate levels.

The subgoals of a goal in the hierarchy are proposed interpretations for the informal parent goals. We adopt the convention that a parent goal is met whenever all of its subgoals are met. The layers of the subgoal structure correspond to decisions about proposed system behavior and how it can be packaged and presented to users. The most specific subgoals at the leaf nodes of the hierarchy are tied directly to elements of the prototype design.

We are currently exploring guidelines for organizing such a subgoal hierarchy and design database structures to provide automated support for maintaining and traversing this hierarchy, for recording past configurations of the requirements and prototype, for keeping track of the change history and the rationale for the requirements evolution that occurs during the prototyping process, and for finding the parts of this structure that are relevant for each of the tasks performed by the designers and analysts.These tools are analogous to library browsers and syntax-directed editors for Ada, which exploit the DIANA tree structure to help designers navigate through and maintain the consistency of complex software structures.

4. Example. To illustrate the concepts described above, we

describe a small sample application generated in CAPS. The purpose of the exercise is to verify the requirements for a robot control system by creating a prototype software system using CAPS. By performing the exercise, the prototype should help us answer the following questions:

a. Are the real-time constraints specified feasible?
b. Is the specified response time sufficient to provide adequate control?
c. Are the user interface mechanisms sufficient from a usability standpoint?

4.1 Informal Specification. The following is an informal specification of the system to be developed.

I. System inputs

1.1 The system shall use a **Keypad** (actual or simulated) to input changes in robot velocity.

1.1.1 Changes to velocity shall be input via a keypad of four directional arrow keys. UP and DOWN shall control the robot's velocity in the Y direction—positive and negative respectively. LEFT and RIGHT shall control the robot's X velocity in the same way (LEFT being negative, RIGHT positive).

1.1.2 Time from a key-press to a corresponding change in robot velocity should be no more than 0.25 seconds (250 ms).

1.2 The Navigation Unit shall measure the robot's current velocity via an **Accelerometer.**

1.2.1 The accelerometer shall report current velocity via an analog-to-digital converter.

1.2.2 Velocity shall be reported in X, Y coordinates relative to a fixed location (0.0,0.0).

1.2.3 Velocity coordinates shall be real numbers with at least six decimal digits precision.

1.2.4 Velocity values shall be sampled at precisely 0.25 second intervals.

1.2.5 The velocity reported by the Accelerometer shall be accurate to within 0.05 meters/sec of actual velocity in any direction.

II. Navigation.

2.1 The **Navigation Unit** shall monitor the robot's status (position and velocity) in two dimensions, in real time.

2.1.1 Both position and velocity will be represented by two dimensional vectors with X, Y coordinates as stated in paragraphs 1.2.2 and 1.2.3.

2.1.2 Units of position will be meters, velocity will be meters per second.

2.2. The robot's position shall be calculated by an inertial navigation system.

2.2.1 Change in position shall be calculated with a standard inertial navigation algorithm for integrating over a sample of five velocity readings for the past 1.0 second interval.

2.2.2 A new position must be calculated when required with a 0.05 second response time (50 ms).

III. System outputs

3.1 The robot's velocity is controlled by a set of four **Thrusters** as illustrated in Fig. 3.

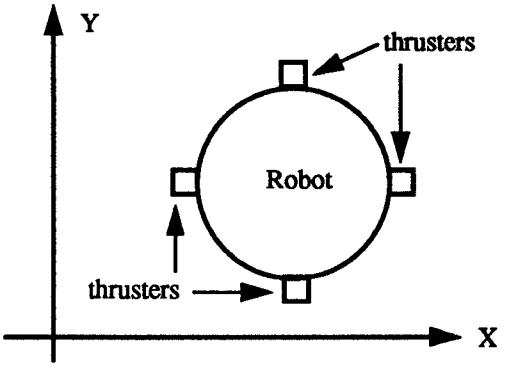

Fig. 3 Schematic of the Robot

3.1.1 Velocity change requests result in thruster pulses which have a direct effect on robot velocity.

3.1.2 Each unit of velocity change requested shall result in 0.05 seconds of thruster pulse which will result in a velocity change of 0.5 meters per second.

3.1.3 Pulse commands to any one thruster may only be in increments of 0.05 seconds.

3.1.4 Concurrent pulse commands to opposing thrusters are not allowed, but commands to adjacent thrusters are.

3.1.5 Giving a single command representing multiple pulse requests uses less fuel and is more desirable than multiple commands of one pulse each. Thus, pulses shall be combined whenever possible.

3.2 The robot's current status (position and velocity) shall be displayed on a **CRT display** in a fixed location on the screen.

3.2.1 Position and velocity should be labeled, and shown to within 0.1 meters.

3.2.2 The status display shall be automatically updated at least once each second.

3.2.3 The robot's position and velocity shall be initialized at 0.0.

4.2 Graphic Editor. Given the requirements for the robot, the CAPS graphic editor is used to model the system and generate an initial PSDL specification. Figure 4 shows the CAPS graphic editor and the model for the robot application. The circles model operators and the arrows represent data flows. Also note that each operator has a corresponding maximum execution time above it.

4.3 PSDL Code. After the model in the graphic editor is complete, a partial PSDL specification is automatically generated. The PSDL code corresponding to the robot application is shown below. The part of the PSDL code generated by the graphic editor is boxed below. The particular characteristics of each operator such as *period* and *triggered by* information, must be supplied by the user within the PSDL editor.

```
OPERATOR robot
   SPECIFICATION
   DESCRIPTION {A simple robot control system.}
END
IMPLEMENTATION
 GRAPH
   VERTEX Get_Keys : 100 ms
   VERTEX Update_Thrust_Req : 10 ms
   VERTEX Fire_Thrusters : 50 ms
   VERTEX Accelerometer : 10 ms
   VERTEX Update_Acceleration : 10 ms
   VERTEX Update_Display : 50 ms
   VERTEX Calculate_Position : 10 ms
   EDGE keys Get_Keys -> Update_Thrust_Req
   EDGE thrust_queue_ptr Update_Thrust_Req ->
            Fire_Thrusters
   EDGE vel_chg Fire_Thrusters -> Accelerometer
   EDGE velocity Accelerometer -> Update_Acceleration
   EDGE accel Update_Acceleration ->
            Calculate_Position
   EDGE status Calculate_Position -> Update_Display
   DATA STREAM
      keys : KEY_PRESSES,
      thrust_queue_ptr : THRUST_QUEUE,
      vel_chg : VECTOR,
      velocity : VECTOR,
      accel : ACCELERATION,
      status : POS_VEL
 CONTROL CONSTRAINTS
   OPERATOR Get_Keys
      PERIOD 50 ms
   OPERATOR Update_Thrust_Req
      TRIGGERED BY SOME keys
      PERIOD 50 ms
   OPERATOR Fire_Thrusters
      TRIGGERED BY SOME thrust_queue_ptr
      PERIOD 50 ms
   OPERATOR Accelerometer
      TRIGGERED BY SOME vel_chg
      PERIOD 50 ms
   OPERATOR Update_Acceleration
      TRIGGERED BY SOME velocity
      PERIOD 50 ms
   OPERATOR Calculate_Position
      TRIGGERED BY SOME accel
      PERIOD 100 ms
   OPERATOR Update_Display
      TRIGGERED BY SOME status
      PERIOD 100 ms
END
```

The PSDL code above describes the requirements of the robot. Additional PSDL is required to define the specific parameters of each operator within the *robot* application. Most of this code is also generated automatically from the graphic editor, but the user must fill in details such as data types and descriptions. As an example, the PSDL description for the accelerometer operator is:

```
OPERATOR Accelerometer
   SPECIFICATION
      INPUT
         vel_chg : VECTOR
      OUTPUT
         velocity : VECTOR
      STATES current_velocity : VECTOR initially
         null_vector
      MAXIMUM EXECUTION TIME 10 ms
      DESCRIPTION {Generates an approximate
         current velocity by factoring in changes in
         velocity due to engine thrusts.}
   END
   IMPLEMENTATION ADA Accelerometer
END
```

4.4 Static Scheduler. The next step is to ensure that the specification is feasible with respect to the timing constraints given in the PSDL. The CAPS static scheduler performs an analysis of the constraints and reports when a schedule is not feasible. The static scheduler uses a variety of algorithms [Lev91] to construct an Ada program representing a schedule. This Ada program encodes the starting and ending times of the execution intervals reserved for each operator with hard real-time constraints. The execution pattern encoded in the static schedule is repeated indefinitely. Operators without hard real-time constraints can be executed only in the time slots that are not used by the static schedule. In the case of the robot application, the timing requirements as given are feasible.

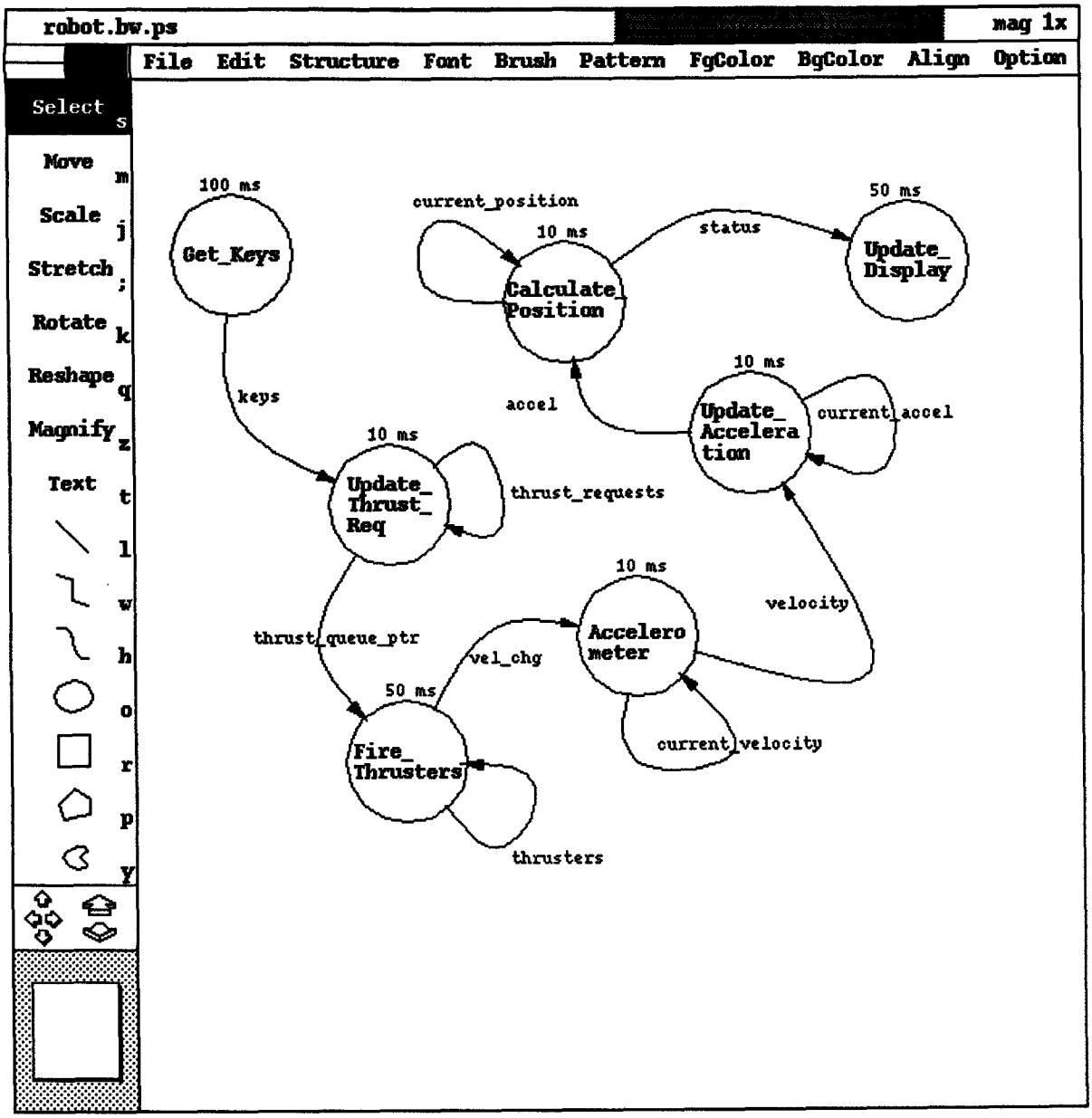

Fig. 4 CAPS Graphic Editor

4.5 Ada Code Generation. After the user has filled in the additional details into the PSDL specification, Ada code is automatically generated to meet the specification. Obviously, the code generator cannot predict the coding details of each individual operator, so the percentage of code automatically generated with respect to the total lines of code in the application depends on the complexity of the operators. In the case of this exercise, about 40% of the final code was automatically generated by CAPS and another 35% came from reusable software components. The code generated for the robot example is too long to include with this paper, but samples of generated Ada code for a C3I System can be found in [Luqi91].

Each operator of the robot application was tested independently before integrating them. It was clear from these tests that the maximum execution times specified for the operators were too restrictive and consequently, the period of each operator was not feasible. This discovery prompted changes in the informal specification which led to a more realistic requirements definition.

5. Conclusion. Rapid prototyping offers an iterative

approach to requirements engineering to alleviate the problems of uncertainty, ambiguity, and inconsistency inherent in the process. CAPS (the Computer Aided Prototyping System) has been built to help software engineers rapidly construct software prototypes of proposed software systems. CAPS helps firm up software requirements through iterative negotiations between customers and designers via examination of executable prototypes. Using a prototype system description language enables engineers and users to quickly focus on the pertinent requirements of their system resulting in increased efficiency and fewer requirement errors.The CAPS system is currently in the process of extension and redesign. Versions suitable for release will be available next year.

ACKNOWLEDGEMENTS

We would like to thank Capt. Patrick Barnes (USAF) for his courage to use CAPS for the robot example used in this paper. We also thank his students for their diligent efforts on the robot project.

REFERENCES

[Ada83] ANSI/MIL-STD-1815A-1983, Reference Manual for the Ada Programming Language, DoD, American National Standards Institute, Feb 17, 1983.

[BA91] Boyes, J. and Andriole, S., Principles of Command & Control, AFCEA International Press, 1987.

[Beam89] Beam, W. R., Command, Control, and Communications Engineering, McGraw-Hill, 1989.

[Berz90] Berztiss, A., "The Specification and Prototyping Language SF", Report 78, Systems Development and Artificial Intelligence Laboratory, Department of Computer and Systems Science, Stockholm University, 1990.

[BL88] Berzins, V., and Luqi, "Rapidly Prototyping Real-Time Systems", *IEEE Software*, September 1988.

[BL91] Berzins, V., and Luqi, Software Engineering with Abstractions, Addison-Wesley, 1991.

[Boar84] Boar, B. H., Application Prototyping: A Requirements Definition Strategy for the 80's, John Wiley and Sons, Inc., 1984.

[Booc87] Booch, G., Software Engineering With Ada, Benjamin/Cummings Publishing Company, Inc., 1987.

[Bori89] Borison, E., "Program Changes and Cost of Selective Recompilation", Technical Report CMU-CS-89-205, Computer Science Department, Carnegie-Mellon University, July 1989.

[Gonz91] Gonzalez, D. W., Ada Programmer's Handbook and Language Reference Manual, Benjamin-Cummings, 1991.

[LBY88] Luqi, Berzins, V., and Yeh, R., "A Prototyping Language for Real-Time Software", *IEEE Transactions on Software Engineering*, October 1988.

[Lev91] Levine, J., "Efficient Static Schedulers for the CAPS System". MS Thesis, Naval Postgraduate School, Computer Science Department, Sep. 1991.

[LK88] Luqi, and Ketabchi, M., "A Computer-Aided Prototyping System", *IEEE Transactions on Software Engineering*, October 1988.

[LVC89] Linton, M. A., Vlissides, J. M., and Calder P. R., "Composing User Interfaces with InterViews", *IEEE Computer*, February 1989.

[Luqi89] Luqi, "Software Evolution Through Rapid Prototyping", *IEEE Computer*, May 1989.

[Luqi91] Luqi, "Rapid Prototyping of Command and Control Software Using CAPS", to appear in IEEE Software, 1991.

[Mok85] Mok, A., "A Graph Based Computational Model for Real-Time Systems", Proceedings of the IEEE International Conference on Parallel Processing, Pennsylvania State University, 1985.

[Nest86] Nestor, J., "Toward a Persistent Object Base", in Advanced Programming Environments, vol. 244, Lecture Notes in Computer Science, Springer-Verlag, 1986, p.372-394.

[Ng90] Ng, P. and Yeh, R., Modern Software Engineering Foundations and Current Perspectives, Van Nostrand Reinhold, 1990.

[NGCR] Naval Research Advisory Committee, Next Generation Computer Resources, Committee Report, February 1989.

[PDW89] PDW 120-S-00533(Rev.B, Change 4), Over-the-Horizon Targeting (OTH-T) Gold Reporting Format, Naval Tactical Interoperability Support Activity, 30 June 1989.

[SLM91] Steigerwald, R., Luqi, and McDowell, J., "A CASE Tool for Reusable Component Storage and Retrieval in Rapid Prototyping", Proceedings of the Third International Conference on Software Engineering and Knowledge Engineering (SEKE '91), Skokie, IL, June, 1991.

[SR88] Stankovic, J. and Ramamritham, K., Hard Real-Time Systems Tutorial, Computer Society Press, 1988.

[TAE] Transportable Applications Environment (TAE) Plus, National Aeronautics and Space Administration, Goddard Space Flight Center, January 1990.

[TY88] Tyszberowicz, s, and Yehudai, A., "OBSERV Object-Oriented Specification, Execution, and Rapid Verification System", 3rd Israeli Conference on Computer Systems and Software Engineering, Tel-Aviv, Israel, June 1988.

[TY89] Tanik, M. and Yeh, R., "The Role of Rapid Prototyping in Software Development", *IEEE Computer*, v. 22, n. 5,pp. 9-10, May 1989.

[VL88] Vlissides, J. M., and Linton, M. A., "Applying Object-Oriented Design to Structured Graphics", Proceedings of the 1988 USENIX C++ Conference, October 1988.

[Your89] Yourdon, E., Modern Structured Analysis, YOURDON Press, 1989.

BIOGRAPHIES

Luqi is an Associate Professor of computer science at the Naval Postgraduate School, Monterey, California. She has been involved with research in rapid prototyping, specification languages, software interfaces, design methodology, real-time systems, software tool integration, and software reusability with Ada. Luqi received a Ph.D. in computer science from the University of Minnesota in 1986.

Robert A. Steigerwald is a Ph.D. student at the Naval Postgraduate School, Monterey, California, currently doing research in the area of reusable software component retrieval. Steigerwald received his bachelor's degree in computer science in 1981 from the US Air Force Academy, Colorado Springs, Colorado and his master's degree in computer science in 1985 from the University of Illinois, Urbana, Illinois.

CDR Gary Hughes, USN, Associate Chairman of Computer Science at the Naval Postgraduate School. He has managed data centers that include large software development sections. His current research interests include software engineering and computer-aided design and ADP security. Hughes received his master's degree from the Naval Postgraduate School in 1982.

Valdis Berzins, Associate Chairman of Computer Science at the Naval Postgraduate School. His research interests include software engineering and computer-aided design. His recent work includes papers on software merging, specification languages, VLSI design, and engineering databases. He received B.S., M.S., E.E., and Ph.D. degrees from MIT, served as an Assistant Professor at the University of Texas, and as an Associate Professor the University of Minnesota. He has developed a number of specification languages and software tools. His current address is NPS 052, Monterey, CA 93943.