1990

Rapid Software Prototyping

Luqi; Steigerwald, R.

Rapid Software Prototyping

Luqi
R. Steigerwald

Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

Abstract*

Rapid software prototyping is an iterative software development methodology aimed at improving the analysis, design, and development of proposed systems. This paper describes rapid prototyping at the system and software levels and reviews the characteristics of computer-aided prototyping. We then describe the state-of-the-art in rapid prototyping and discuss technologies that improve the future outlook for prototyping, such as prototyping languages, software reuse, and designer interfaces. To add some cohesion to the concepts, we describe the characteristics of a computer-aided rapid prototyping system. Finally, we provide summaries of the outstanding papers that comprise the rapid prototyping mini-track.

1: Introduction

Prototypes are developed as an aid for analysis and design of proposed systems. A prototype is a simplified model of a proposed system that is built for a specific purpose, such as:

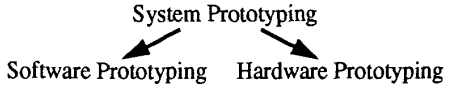
1. Formulating and evaluating requirements, specifications, and designs.
2. Demonstrating feasibility, system behavior, performance, etc.
3. Identifying and reducing risks of system mis-development.
4. Communicating ideas, especially among diverse groups.
5. Answering questions about specific properties of proposed systems.

Prototyping is a method for addressing problems in the design and development of systems via prototypes [Tani89, HICS90]. Prototyping is most useful for complex first-of-a-kind systems that must be reliable, such as aircraft control systems, financial systems, medical systems, and military systems.

1.1: System prototyping

Prototyping applies to all kinds of systems - software, hardware, people, or any combination of these. Prototyping is an accepted part of most branches of engineering, but has been applied to software only in recent years.

Levels of prototyping: Prototyping can be applied at all levels of a system hierarchy. It applies equally well to an aircraft, to a radar system, to a radar processor, to an ALU or memory within a processor, to one or more circuit boards, to logic modules, to hybrid circuits, and to individual chips. It applies as well to non-physical things, including software in particular, at all of its levels of design, and the activities of humans, particularly as they interact with and affect the behavior of systems of interest.



Prototyping is also useful at the highest system level, for resolving questions regarding resource allocation relative to the feasibility of timing constraints. There is a trade-off between software functionality, required response time, and hardware resources. Hardware and software is often developed in independent efforts based on separate and fixed requirements for each. To ensure proper system integration at the end of the project, it is useful to explore the hardware configurations required to support fixed software functions and timing constraints as well as the various combinations of software functions and response times that are feasible on given hardware configurations before committing to particular hardware or software requirements. This is particularly important for systems that will be using custom-built hardware. Simulations allow various parameters of the hardware configuration to be varied and optimized relative to the software design.

*This research was supported in part by the National Science Foundation under grant #CCR-9058453.

0073-1129-1/92 $3.00

A prototype may be implemented, in part or in whole, with the actual elements of a system (i.e., the most current versions of the elements); a prototype of an aircraft may, for example, include the current version of the actual airframe, but not the real navigation system. Some elements of a prototype may be implemented with lower level prototypes or with models; a computer, for example, might be implemented with prototype memory and logic that may or may not be functional; signal processing algorithms may, similarly, be modeled by software that simply approximates their execution time, with no regard for function.

Models in prototyping: Models are used extensively in prototypes. A model may be used to represent or describe some aspect of a system that has not been actualized in the real hardware, software, or human operators of a system or to represent or describe some aspect of a system more conveniently and efficiently than the actualized version. For example, one would normally use a model of the sensor input to a signal processing system prototype, rather than use the actual sensors, because the models afford greater flexibility and convenience in evaluating the prototype characteristics.

Models are utilized in prototypes as they are appropriate to the purpose of the prototype. Models may be mathematical, logical, electrical, mechanical, software, etc.; they take the form appropriate to the situation. For example, a model of a computer in a prototype constructed to evaluate cabin space requirements in a space ship may simply be a cardboard box with specific dimensions. A model of the same computer in a functional prototype of the space ship might consist of 50,000 lines of VHDL code describing its structure, timing, and behavior at a micro-architecture level.

Prototyping of embedded systems: Many systems of interest contain embedded software systems that must meet real-time constraints to maintain control of the surrounding system. These real-time constraints introduce a coupling between the software design and the hardware design, because the response time depends on the number or instructions per second that can be executed by the processor hardware and the number of bits per second that can be transferred by the network and storage hardware, in addition to the number of instructions that must be executed and the amount of data needed to complete a software algorithm. A context diagram for a software system, together with models of the behaviors of the interacting external systems and models of the host hardware for the software form the basis for evaluation, optimization, and acceptance of the entire system configuration. This systems-level evaluation is especially important for proposed real-time systems because the feasibility of the entire system depends on all of these factors, and is in doubt until all of the elements are fixed and their interactions can be evaluated. System level prototypes are used to establish rough feasibility assurances early in the design, to identify the aspects of the design that have the largest impact on the feasibility of the whole, and to track and focus attention on the critical areas of the design as it becomes more solid, more refined, and less risky.

This motivates the need for prototyping of entire embedded systems, not just the hardware or the software, to assess design decisions related to resource allocation and performance. The result of such an effort is a hybrid prototype, that can model different subsystems at different levels of detail: parts of the system that are to run on existing types of hardware can be prototypes on the actual equipment, while parts that are to run on proposed new types of hardware can be evaluated with respect to software simulations of the hardware.

Hardware prototyping: Hardware prototypes are used largely to measure and evaluate aspects of proposed designs that are difficult to determine analytically. For example, simulation is widely used to estimate throughput and device utilizations for proposed hardware architectures. Although software prototypes can be used in a similar way, to determine time and memory requirements for proposed designs, the focus of software prototyping is usually to evaluate the accuracy of the problem formulation, to explore the range of solutions possible, and to determine the required interactions between the proposed system and its partially unknown environment.

1.2: Iterative prototyping process

Software in general and the formulation of adequate software requirements in particular have become limiting factors in applications of computers. As systems get larger and serve more diversified user communities, formulating requirements that accurately represent the customers' needs becomes very difficult. Different people have partially overlapping and sometimes contradictory views of different aspects of the problem. Analysts must create precise, formal models of unfamiliar problems, based on imprecise communication with people who have a partial understanding of what is needed. This is particularly difficult because people with different backgrounds tend to use the same words in different ways.

The transition from fluctuating informal views of the problem to a fixed formal model is fundamentally uncertain. It is difficult to synthesize all its requirements logically and consistently all in one operation. Reasonably accurate models can be created via an iterative system development method that produces a series of related prototypes to converge on a consensus about the requirements. An iterative prototyping process can be defined to produce a series of software prototypes

S[i], i = 1, ... , n , ...

which are increasingly accurate approximations to the envisioned system S.

Information gained from analyzing and criticizing S[i] is used to construct S[i+1]. Assume that the modification to prototype S[i] is represented by ΔS[i], we have

$$S[i] + \Delta S[i] \approx S[i+1]$$

where S[i+1] is a better approximation of S than S[i] if we can assume convergence of such a series based on human cognitive ability, then

$$S = \lim S[i] \text{ when } i \to \infty.$$

Practically, resources are limited, and an integer

$$n = N$$

must be chosen such that S[N] approximates S and the imperfectness or the difference ΔS[N] is small enough to be acceptable relative to available budget and resources.

In the experiences gained from software maintenance and development, the size of N does indicate how well the system solves customer's problem. N represents the number of times the system has been fixed. Of course, exceptional random human error must be controlled to support continual improvement. To be effective, prototypes must be constructed rapidly and at low cost, and they must be easy to change.Due to the fact that programming is a labor intensive task, constructing the sequence S[i], 1, ... , N could be an exhaustive process. If we apply the process at the specification level and use S[i] for the versions of the specifications produced during the modification of prototype series, we have a substantially reduced task for the prototyping effort, but the difficult problem of how to execute the specifications remains.

If a transformation function

$$T: S \to C$$

can be defined, where S is the specification and C is the target programming language code, we have

$$T(S[i]) \to C_i \text{ and } \lim C_i = C \text{ when } i \to \infty$$

where S is the specification of C. Such a function T can be realized as a family of functions

$$T\{t[k], k=1,...,m\}.$$

For example, in our CAPS system, we define t[1] as a translator, t[2] as a static scheduler, t[3] as a dynamic scheduler, t[4] as a debugger generator, and t[5] as a software base retrieval system.

1.3: Computer-aided prototyping

The key to rapid prototyping is computer-aided design. Prototypes must lend insight into the proposed system. Code that is manually created in a conventional programming language is not the preferred way to create a prototype rapidly, because in such a context quick development often implies sloppy design and missing documentation. Since the series of prototypes must undergo frequent and radical changes, all of the usual problems of maintaining software systems are intensified, and the results of such an approach can be quite disappointing.

The alternative is to use special prototyping languages supported by extensive computer-aided design tools. These tools automate many of the processes that are carried out manually in conventional systems development, and include support for system construction and modification based on the specifications. This ensures that the specifications remain current despite the rapid changes to the prototype, helps designers and tools to determine and analyze the intended behavior of the system, and provides the basis for automated synthesis of code, real-time schedules, and various details of the design. In order to automate more of the design process, the system must have formal representations of more information about the design that is represented only in the minds of the designers using conventional software development techniques. These representations must be coupled with tools that can use the representations to synthesize, analyze, and check various aspects of the design.

To be effective, prototypes must be constructed and modified rapidly, accurately, and cheaply. They do not have to be efficient, complete, portable, or robust, and they do not have to use the same hardware, system software, or implementation language as the delivered system. Automated construction of programs is useful in this context even if the resulting programs are not very efficient.

2: Software prototyping

2.1: Current practice

Currently, the most effective systems for automatically generating programs are focused on relatively narrow problem domains. These systems gain their power from generic solution strategies specific to the application domain that are embodied in program generation schemes. The best known systems of this type create database applications based on graphical interactions with end-users (non-programmers). Other examples include interactive tools for creating graphical user interfaces, and attribute-grammar tools for generating translators, syntax-directed editors, and other language-based systems. [Budd84, Reit84, Budg84, Lame84]

2.2: Current research

A computer-aided prototyping system of considerable power can be built by integrating application generators covering several different domains and extending them with tools for generating programs from general-purpose problem specifications. The application generators can be completely automatic while the general tools can be interactive, can be supported by a software base of reusable components, and can fall back on manual programming if all else fails [NSWC91].

Fundamental work on mathematical models of

applications and the complete redesign of existing application generators will be required to achieve integration because current application generators are based on incompatible models. Clear and simple mathematical models are required for successful automation. A standard set of abstract data types forming the machine representations of a standard prototyping language are also needed. These data types should achieve persistence through an object-oriented design database, that provides a record of the evolution of the prototype. The persistent data types should also provide a consistent set of tool interfaces, where different tools can have different views of the data. Development of object-oriented database systems that support type hierarchies with multiple inheritance is needed to fully realize this vision.

To produce deliverable software, prototyping tools must be extended with optimization capabilities to produce programs whose efficiency is comparable to the designs of competent programmers. The beginnings of the required technologies are visible: correctness-preserving transformations and performance estimation techniques to guide derivation strategies. Work on reasoning support and methods for interactively supporting software engineers, such as the Programmer's Apprentice effort [Rich90], are also critical for achieving the next level of automation.

2.3: Future effort

In the long run, a new kind of language[Luqi91] may be needed that combines the flexibility of an interpreted language with a powerful set of features for selectively declaring various kinds of compile-time constraints as consistent refinements. Some critical aspects of such a language are smooth integration of optional explicit storage management policies with a default policy of garbage collection, optional explicit synchronization policies with a default of mutual exclusion on multiprocess interactions, and optional explicit type declarations with defaults based on type inference procedures. Realization of these goals will enable the construction of flexible prototypes that can be smoothly and consistently optimized by adding detailed constraints.

3: Technologies for prototyping

Over the past 40 years computer hardware has become dramatically cheaper, faster, and more reliable, but advances in software technology have been relatively modest compared to increases in demand. Compilers made a major step in automating the programming process by redefining programming as the design of algorithms and data structures instead of the design of machine code. The next major steps in automation will redefine programming to become the formulation of requirements and the design of system interfaces. Automated design of unrestricted algorithms and data structures is extremely difficult. Before complete automation in a general setting, partially automated or computer-aided software design will be applied to software prototyping.

The most important emerging technologies for computer-aided prototyping include prototyping languages, support for reuse, and designer interfaces that provide decision support functions.

3.1: Prototyping languages

The goal of computer aided prototyping is to automate the design effort at the early phases of software development. The only way to reach this goal is to create mechanically processable and executable documents at the specification level. Prototyping languages combine the functions and benefits of specification, design, and programming languages. Fig. 1 illustrates the relationship of prototyping languages to the prototyping process, and compares it to the languages used in traditional software life cycles.

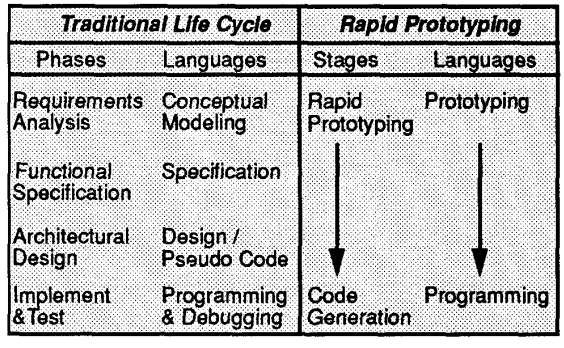


Fig. 1 Software Language Hierarchy

Prototyping languages form a new category in the family of computer languages. The purpose of a prototyping language is to define an executable model of a system, using both black-box and clear-box descriptions. A prototyping language has no obligation to give detailed algorithms for all components of the system as long as it is descriptive and executable. We briefly compare prototyping languages to specification languages, design languages, and programming languages. Specification languages are used for recording external interfaces in the functional specification stage and for recording internal interfaces during architectural design at the highest levels of abstraction. They are also used in verifying the correctness and completeness of a design or implementation. Design languages are used for recording conventions and interconnections during architectural design and module design.

The *difference* between specification and design languages is the difference between interface and mechanism: a specification says what is to be done, and a design says how to do it. The evaluation criterion for both specification and design languages is the ability to support simple, concise, and humanly understandable descriptions of complex behavior. It is useful for specification and design languages to be executable, but simplicity of expression takes precedence when the two considerations conflict. Computer aid is desirable for determining the properties of a specification and certifying that a design realizes a specification. Execution can help attain these goals, but it is not the only way to do so, and it is not necessarily the most effective way.

The *difference* between a design and a program is the difference between a plan and a finished product: a design records the early decisions that determine an implementation strategy, while a program contains all the details necessary to get an efficiently executable system. The primary goal of a design is documentation rather than execution, while the primary goal of a program is usually efficient execution.

Common *strengths* of specification languages are simplicity, abstraction, clarity of expression, and means for rigorous logical reasoning. Common *strengths* of design languages are expressiveness and support for recording goals and justifications. A common *weakness* of specification and design languages is lack of efficient facilities for execution or lack of any effective means for execution. The *strength* of most programming languages is supporting efficient execution, while common *weaknesses* are the need for specifying many details and lack of facilities for recording goals and justifications in a formal way. The contribution of a prototyping language is to integrate the functions of specification and design languages with the capability for execution. However, because of the wide range of goals for prototyping languages, they may not be as effective for any of the purposes mentioned above as a language optimized just for that purpose.

3.2: Software reuse

Software reusability offers a tremendous advantage for rapid prototyping, that of increased productivity. A rapid prototyping paradigm that relies on a library of reusable modules has the potential to generate software prototypes in a much more rapid fashion than prototyping by manual means. However, technological barriers impede the progress of code reuse. Among them are component classification, retrieval, and integration problems. To confound the technical barriers, there exist managerial, economic, and cultural barriers as well.

Classification and retrieval: Storing and maintaining a large collection of software components, i.e. a software repository, requires some sort of classification scheme to support classifying, identifying, and retrieving components. The problem is analogous to the document storage and retrieval problem. Exercising this analogy, some researchers have employed keyword and multi-attribute paradigms [Prie91b, Brow90] while others have explored natural language [Burt87] and expert system [Wood88] techniques. These approaches all have merits but tend to focus on descriptors of the software product rather than the product itself. Thus, they have a broad applicability to various forms of software products, not just source code.

Computer software, i.e. actual source code, has characteristics that set it apart from the analogy with technical documents. If we assume that the reusable component is a subprogram or abstract data type, then we are assured the component will have *syntax* and a *semantics*.

The syntax of the component is its interface, that is, the number and types of parameters in its specification. The semantics of a component are the statements found in the body. Some researchers have focussed on syntax alone [runc & toyn] while others have exploited both syntax and semantics [Stei92, Roll90, Honi86].

Taken a step further, a theorem proving methodology could further refine the search for a component, using formal specifications for representing the component's syntax and semantics. In fact, this may be a fundamental requirement if the retrieval tool is to become fully automated. From what we have seen, a general purpose software classification and retrieval tool will need the following features to be successful:

1. A browser
2. Keyword search
3. Multi-attribute (or faceted) classification and search
4. Syntactic and semantic search
5. Theorem prover

Component integration: Most of the retrieval mechanisms described above require manual integration of the software component with the user's system once the component has been found. It seems feasible, however, given mapping information for a candidate component that satisfies a query, to build automatically a wrapper around the candidate that provides the interface expected by the query. While this may not be an efficient integration, it provides the required functionality to the user and would be acceptable in a rapid prototyping environment.

Future of reuse: Widespread software reuse is elusive not only because of the technical barriers but also because of the absence of organizational structures to support the process. Prieto-Diaz [Prie91a] argues that "the most important issues influencing software reuse [are]

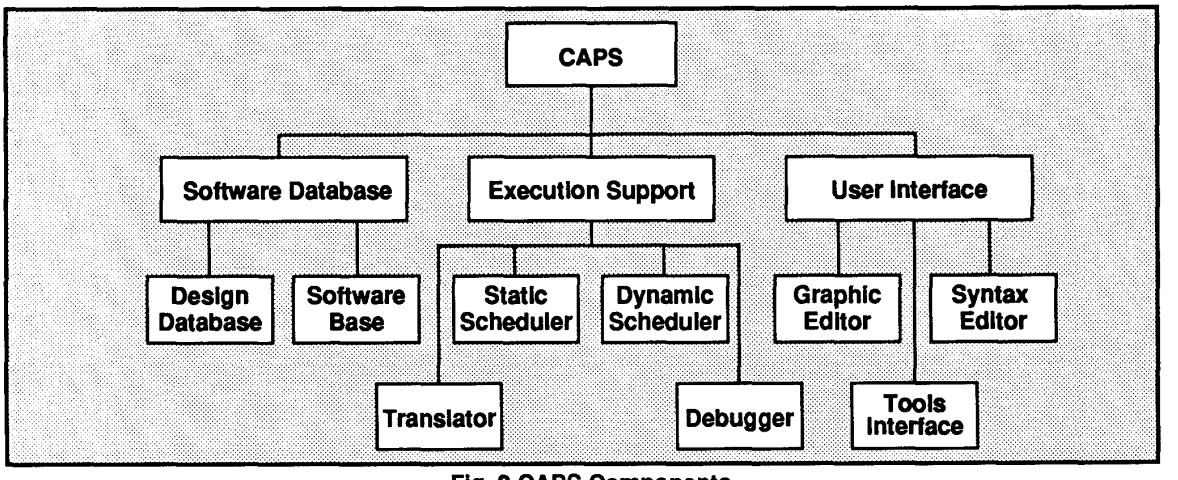


Fig. 2 CAPS Components

managerial, economic, legal, cultural, and social." In addition to conquering the technical barriers, we must strive to build the proper organizational infrastructures. Success can be achieved in the interim if we bound the scope initially and work toward an ideal model incrementally.

3.3: Designer interfaces and decision support

The designer interface of a prototyping system should match the prototyping method and model the designer's decision process. The interface should shield the designer from the details of data management and the boundaries between the tools in the environment. A graphical interface is useful for providing summary views of the prototype, especially for representing system decompositions. Technologies for creating graphical interfaces such as InterViews [Lint89] and graphical editors such as Idraw [Vlis88] are needed for building the tools to support these aspects. It is important to build the user interface portions of the system via toolkits or user interface generation systems because the details of the prototyping method are likely to change, both as more refined methods are developed and in order to adapt to application-specific and organization-specific differences.

The graphical interface and the associated text annotations should be integrated by tools that provide guarantees of consistency and automatic constraint propagation. Attribute grammar technology is relatively well developed for realizing constraint propagation and consistency checking for text expressed in formal languages. A remaining challenge is providing the analogous capabilities across the boundary between the graphics and the text.

Another important aspect of computer-aided prototyping is technology for managing the evolution of the prototype [Luqi90]. The prototype is expected to go through many different versions, and configuration control is one of the areas where decision support for the designer is important. Such decision support can be based on a formal model of software evolution such as [Luqi90]. Such a model provides the basis for managing the evolution of a complex prototype, coordinating the concurrent efforts of a team of prototype designers, and exploring various combinations of several design alternatives. Better high level models of these processes are needed, coupled with tool support for keeping track of the configurations, automatically identifying various configurations in terms that make sense to the designer, organizing the configuration structure coherently, and automatically materializing new combinations of existing configurations with consistency checking.

4: A computer-aided prototyping system

The computer aided prototyping system (CAPS) [Luqi88b] is an integrated environment aimed at rapidly prototyping hard real-time embedded systems. It is comprised of an integrated set of software tools that include an execution support system, a rewrite system, a syntax directed editor with graphics capabilities, a software base, a design database, and a design management system. Fig. 2 shows the high level organization of CAPS.

Embodied within the CAPS software development approach is a systematic design method for rapid prototype construction [Luqi88c]. System or subsystem descriptions are stated at a problem-oriented, abstract level and iteratively refined into a hierarchically structured prototype using a uniform decomposition method that combines the advantages of data flow and control flow. At each level of the hierarchy, the designer focuses only on the details important at that level.

CAPS is based on a prototyping language called PSDL (Prototype System Description Language) [Luqi88a]

designed to serve as an executable prototyping language at the specification or design level. PSDL is based on a mathematical model [Luqi88a].

To generate a prototype, the designer of the prototype uses the graphic editor to create a graphic representation of the proposed system. The graphic representation is used to generate part of an executable description of the proposed system, represented in PSDL. PSDL descriptions are used to search the software base to find reusable components that match the specifications. A transformation schema is then used to transform the PSDL into Ada and bind the retrieved reusable components. The prototype is then compiled and executed. The end user of the proposed system evaluates the prototype's behavior against the expected behavior. Successive iterations of this process should lead to a system that ultimately satisfies the user's requirements. [Cumm90]

CAPS is divided into three major subsystems. They are the software database, the execution support system, and the user interface. The following sections describe each in turn.

4.1: Software database

The software database has two primary subsystems, the design or engineering database and the repository of reusable components, called the software base.

An engineering database should provide the following facilities to support computer-aided software development environments [Dwye91]:

- Persistence
- Concurrency control
- Version control
- Reuse of past design objects
- Configuration control
- A wide variety of data storage
- Guarantees data will not be corrupted due to system or media failure

The engineering database of CAPS supports all of these facilities using an object-oriented database management system[Nestor86] supporting a graph model of software evolution [Luqi90].

The second subsystem, the software base, is a repository for reusable software components. An object-oriented database management system (OODBMS) [Onto91] provides the basis for the underlying component storage. Tools have been built on top of the OODBMS to perform browsing, syntactic search [McDo91], and semantic search [Stei91] for components.

The key for the syntactic and semantic search mechanism is a *query* written in PSDL augmented with an algebraic specification language. A user's query is a *requirement* for the prototyping system being built. Using syntactic and semantic normalization techniques, the search tool evaluates all components in the database to determine which ones will satisfy the query. The syntactic search mechanism quickly provides a set of candidates that then pass through semantic matching for subsequent ranking. The designer then selects the best candidate from the ordered list. A future tool will provide support for automatically integrating the retrieved component into the prototype.

4.2: Execution support system

The execution support system gives the designer the ability to execute the prototype. This support system consists of four major components: a translator, a static scheduler, a dynamic scheduler and a debugger. The translator generates code, binding together the reusable components retrieved from the software base. Its primary functions are to implement data streams and control constraints. The static scheduler allocates time slots for operators with real-time constraints before execution begins. If the allocation succeeds, all operators are guaranteed to meet their deadlines even with worst case execution times. The dynamic scheduler invokes operators without real-time constraints in the time slots not used by the operators with real-time constraints. The debugger offers designer support for locating logical errors during prototype execution. [Pala90]

4.3: User interface

The CAPS interface provides a cohesive software development environment integrating the tools of CAPS. A pictorial representation of this environment is given in Fig. 3. At the core of the environment is the host operating system. The windowing system, X-windows, is the next layer. InterViews, the toolkit chosen to develop the user interface, provides the interface between the upper layers of the environment and X-windows. The CAPS tools sit on top of InterViews and are surrounded by the tool interface. The tool interface provides all communication between the tools and the user interface. The outermost layer of the environment is the user interface. This layer hides the underlying implementation details from the designer. [Cumm90]

5: Overview of the papers in the mini-track

Many excellent papers were submitted this rapid prototyping mini-track. Nine of the papers were accepted in their entirety and four as two-page research summaries. This section gives a short synopsis of each of the papers. We classify the papers into four groups. The first group of papers are contributions to prototyping languages and software synthesis. The second group focuses on support for software reuse, an important aspect of rapid prototyping systems. The third group of papers offer contributions to

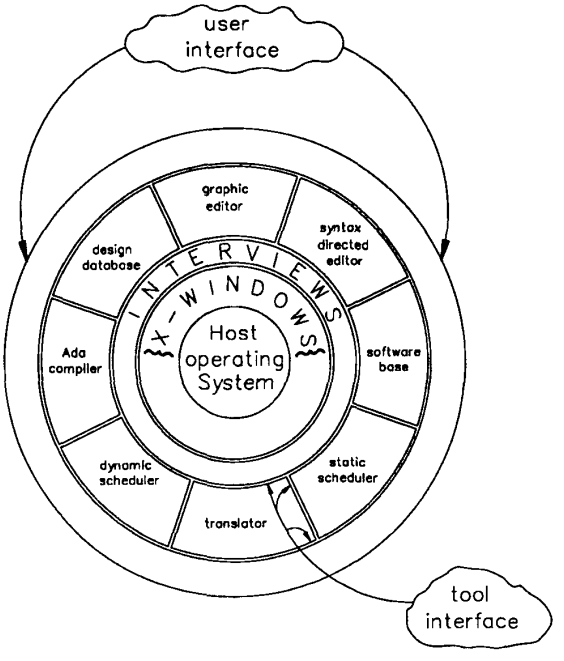


Fig. 3 CAPS Environment [Cumm90]

computer-aided prototype evolution. The final group is a forum of research summaries supporting rapid prototyping.

5.1: Prototyping languages and synthesis

"A Resolution Method for Predicate Logic Specification into Executable Code," by Kouichi Ono and others, introduces a resolution method for predicate calculus specifications resulting in a method to synthesize executable code. The main contribution is the introduction of transformation and resolution techniques for the predicate calculus formulae. Partial correctness of the resolution process is verified.

"Update Plans," by Hugh Osborne, discusses the syntax and semantics of "update plans." Update plans are based upon lamda calculus. The author demonstrates how update plans can be used to specify and prototype abstract machines. Syntax and translation schemes are given in appendices.

"Common Intermediate Design Language," by Henson Graves and Wolfgang Polak, introduces a language called "Common Intermediate Design Language." Concurrency aspects of CIDL are based upon Hoare's communicating sequential processes (CSP). This language is a high level system design language that can be executed directly or translated into Ada, LISP, or C. CIDL is used to describe "reactive" systems that involve "events" that may change "stores" over time. The major contribution of this paper includes a presentation of the key features of CIDL which has been used in a reuse environment to synthesize large real-time applications.

5.2: Support for software reuse

"On a Fundamental relationship Between Software Reuse and Software Synthesis," by Ann Gates and Dan Cooke, discusses language issues common to both software reusability and synthesis environments. A small example specification language is defined together with its synthesis environment leading to a definition of ambiguity. Major contributions include definition of inherent ambiguity and a proposed architecture that would include both software reusability and synthesis.

"Using Hypertext to Locate Reusable Objects," by R. N. Robson, focuses on the use of hypertext links to locate code in reusable software component libraries. The main contribution of this paper is the introduction of a specification language, DYHARD, which is used to specify the automatic linking of reusable components for subsequent retrieval.

5.3: Computer-aided prototype evolution

"Supporting System Maintenance with Automatic Decomposition Schemes," by Rajeev Gopal and others, focuses on decomposition schemes to contend with maintenance issues. A relational model that allows a maintainer to project dependencies among variables and program statements is the major contribution. Using the resulting relational environment, the impact of a change can be predicted to some extent.

"Design Structuring and Allocation Optimization," by Steven L. Howell and others, presents of an approach for optimizing the design of large, complex, real-time systems in distributed and parallel environments. The main contribution is a method for defining and analyzing the ways in which system components react with each other through interfaces and through the sharing of global resources. The approach fuses heuristic, probabilistic, and deterministic methods to prototype resource sharing, parallel systems.

"Object-Oriented Design of an Expandable Hardware Description Language Analyzer for a High-Level Synthesis System," by Lian Yang and others, discusses the use of object-oriented programming language techniques to construct an expandable Hardware Description Language Analyzer. The main contribution of this approach is that it provides a better way to model high-level synthesis design entities in that data and methods are organized through a "class lattice." This leads to improved expandability, and active design entities at various design stages.

5.4: Forum

"Software Development with Transformable Components," by M. G. Christiansen and others, describes an

environment wherein it is possible to capture, manipulate, and integrate abstractions needed for software development. The main contribution is the description of an implemented system that assists a knowledge engineer in the construction of generic abstract components and their classification for subsequent reuse. They also describe implemented, automated mechanisms to develop software based upon the abstract components.

"Fast Static Timing Analysis of Real-Time Systems," by Albert Mo Kim Cheng, presents a General Analysis Algorithm (GAA) that performs a static syntactic and semantic check of a rule base in order to determine if a given program has a bounded response time. This program can analyze a large class of rule-based EQL programs to see if they can be used in safety-critical environments. As such the algorithm may provide a criteria for the reusability of software in safety critical domains.

"Rapid Prototyping in an Object-Oriented Pictorial Dataflow Language," by Michael Nelson and Ronald Byrnes describes an object-oriented approach to rapid prototyping using an off-the-shelf pictorial, object-oriented, dataflow language. The paper states that merging object-oriented and dataflow methods is effective for prototyping complex systems (such as the control software for an Autonomous Underwater Vehicle).

"Assessing Industrial Prototyping Projects," by Antoinette Kieback and others, describes six case studies that applied prototyping to software development. The projects studied ranged from 240 person-years to 2 person-years. The study differentiates between horizontal prototyping (i.e., where a layer of the system is prototyped) and vertical prototyping (i.e., where an entire component is prototyped). The results indicate that prototyping improves the ability of developers to plan.

RAPID PROTOTYPING
Session 1
"A Resolution Method for Predicate Logic Specification into Executable Code"
"Update Plans"
"Common Intermediate Design Language"
Session 2
"Supporting System Maintenance with Automatic Decomposition Schemes"
"Design Structuring and Allocation Optimization"
"Object-Oriented Design of an Expandable Hardware Description Language Analyzer for a High-Level Synthesis System"
Session 3
"On a Fundamental relationship Between Software Reuse and Software Synthesis"
"Using Hypertext to Locate Reusable Objects"
Forum
"Software Development with Transformable Components"
"Fast Static Timing Analysis of Real-Time Systems"
"Rapid Prototyping in an Object-Oriented Pictorial Dataflow Language"
"Assessing Industrial Prototyping Projects"

Table 1 - Session Schedule

6: Conclusions

Rapid software technology is a critical and active area of software engineering research. Aspects of computer-aided prototyping such as prototyping languages, software reuse, and decision support for prototype designers are some of the critical issues in the area. The papers in the rapid software prototyping track are representative of the current state of the art in the area. The organization of the papers in the track is listed in Table 1.

References

[Burt87] Burton, Bruce A., Wienk, Rhonda, and others, "The Reusable Software Library", *IEEE Software*, v. 4, pp. 25-33, July 1987.

[Brow90] Brown, James C., Lee, Taejae, and Werth, John, "Experimental Evaluation of a Reusability-Oriented Parallel Programming Environment", *IEEE Transactions on Software Engineering*, v. 16, pp. 111-120, February 1990.

[Budd84] Budde, R., and others, eds, *Approaches to Prototyping: Proceedings of the Working Conference*, Springer Verlag, 1984.

[Budg84] Budgen, D., "The Use of Prototyping in the Design of Large Concurrent Systems", *Approaches to Prototyping*, Springer-Verlag, 1984.

[Cumm90] Cummings, Mary Ann, *The Development of User Interface Tools for the Computer Aided Prototyping System*, Master's Thesis, Naval Postgraduate School, December 1990.

[Dwye91] Dwyer, Andrew P., and Lewis, Gary W., *The Development of a Design Database for the Computer Aided Prototyping System*, Master's Thesis, Naval Postgraduate School, September 1991.

[HICS90] Rapid Software Prototyping Mini-Track, Hawaii International Conference on System Sciences, pp. 198-266, January, 1990.

[Honi86] Honiden, S., and others, "Software Prototyping with Reusable Components", *Journal of Information Processing*, v. 9, pp. 123-129, March 1986.

[Lame84] Lamersdorf, W., and Schmidt, J., "Specification and Prototyping of Data Model Semantics", *Approaches to Prototyping: Proceedings of the Working Conference*, Springer Verlag, 1984.

[Lint89] Linton, M.A., Vlissides, J.M., and Calder, P.R., "Composing User Interfaces with InterViews", *IEEE Computer*, February 1989.

[Luqi88a] Luqi, Berzins, V., and Yeh, R., "A Prototyping Language for Real-time Software", *IEEE Transactions on Software Engineering*, pp. 1409-1423, October 1988.

[Luqi88b] Luqi, and Ketabchi, M., "A Computer Aided Prototyping System", *IEEE Software*, pp. 66-72, March 1988.

[Luqi88c] Luqi, and Berzins, V., "Rapidly Prototyping Real-Time Systems", *IEEE Software*, pp. 25-36, September 1988.

[Luqi89] Luqi, "Software Evolution via Rapid Prototyping", IEEE Computer 22, 5 (May 1989), P. 13-25.

[Luqi90] Luqi, "A Graph Model for Software Evolution", IEEE Trans. on Software Engineering 16, 8 (Aug. 1990), p. 917-927.

[Luqi91] Luqi, "Computer-Aided Software Prototyping", IEEE Computer 24, 9 (Sep. 1991), p. 111-112.

[McDo91] McDowell, John K., *A Reusable Component Retrieval System for Prototyping*, Master's Thesis, Naval Postgraduate School, September 1991.

[NSWC91] Report of the NSWC/ONT Workshop on Systems Evaluation and Assessment Technology, NSWC, Silver Spring, MD, Aug. 1991.

[Nest86] Nestor, J., "Toward a Persistent Object Base', *Advanced Programming Environments*, Springer LNCS 244, 1986, p. 372-394.

[Onto91] Ontologic, Inc., *Ontos Object Database Documentation Release 1.5*, Burlington, MA, 1991.

[Pala90] Palazzo, Frank V., *Integration of the Execution Support System for the Computer-Aided Prototyping System (CAPS)*, Master's Thesis, Naval Postgraduate School, September 1990.

[Prie91a] Prieto-Diaz, Ruben, "Making Software Reuse Work: An Implementation Model", *ACM Software Engineering Notes*, v. 16, pp.61-68, July 1991.

[Prie91b] Prieto-Diaz, Ruben, "Implementing Faceted Classification for Software Reuse", *Communications of the ACM*, v. 34, pp.88-97, May 1991.

[Reit84] Reitenspieb, M., and Merz, G., "Automatic Generation of Prototypes from Formally Specified Abstract Data Types", *Approaches to Prototyping: Proceedings of the Working Conference*, Springer Verlag, 1984.

[Rich90] Rich, C. and Waters, R., *The Programmer's Apprentice*, Addison-Wesley, 1990.

[Roll90] Rollins, Eugene J., and Wing, Jeanette M., "Specifications as Search Keys for SW Libraries: A Case Study using Lambda Prolog", CMU-CS-90-159, Carnegie Mellon University, 26 September 1990.

[Runc89] Runciman, Colin, and Toyn, Ian, "Retrieving Reusable Software Components by Polymorphic Type", in *Proceedings of the International Conference on Functional Programming and Computer Architecture (FPCA'89)*, New Orleans, 1989, pp. 166-173.

[Stei91] Steigerwald, Robert, *Retrieving Reusable Software Components via Normalized Algebraic Specifications*, Ph.D. Dissertation, Naval Postgraduate School, December 1991.

[Stei92] Steigerwald, Robert, Luqi, and Berzins, Valdis, "A Tool for Reusable Software Component Retrieval via Normalized Specifications", *Proceedings of the 25th Hawaii International Conference on System Sciences*, Koloa, Hawaii, Jan 7-10, 1992.

[Tani89] Tanik, M. and Yeh, R., "The Role of Rapid Prototyping in Software Development", *Proceedings of the Hawaii International Conference on System Sciences-22*, pp. 337-338, January 1989.

[Wood88] Wood, Murray, and Sommerville, Ian, "An Information Retrieval System for Software Components", *SIGIR Forum*, v. 22, pp. 11-28, Spring/Summer, 1988.