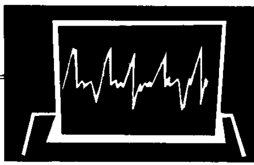


STATUS REPORT: COMPUTER-AIDED PROTOTYPING

This workshop report assesses the role of computer-aided prototyping in software development, identifies the supporting technology necessary for prototyping to reach its potential, and suggests some directions for future work.

LUQI
Naval Postgraduate School

WINSTON ROYCE
TRW

Prototyping — the construction and analysis of an executable model that approximates a proposed system — is an accepted part of most branches of engineering, but has only recently been applied to software engineering. Software prototypes are used somewhat differently than hardware prototypes.

For the most part, hardware prototypes are used 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 utilization in proposed hardware architectures. Although software prototypes can be used likewise to determine time and memory requirements, they usually focus on evaluating the accuracy of problem formulation, exploring the range of possible solutions, and determining the required interactions between the proposed system and its environment.

A prototype differs from the proposed system in that it may run on different hardware and under different system software, it may have different performance and capacity properties, and it may not include all details. For example, early prototype versions often assume that users don't make mistakes and operating environments don't malfunction and so omit many error-handling capabilities.

Partial coverage of system behavior in the initial prototype reduces costs and simplifies the problem. But it takes good judgment to decide which aspects of the proposed system are the most important and least understood. A prototype should help clarify the most uncertain aspects so that development effort will not be wasted on functions that do not fulfill the customers' desires.

Prototyping has three main benefits:

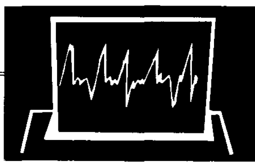
♦ It improves communication through demonstration, which enables earlier, more effective dialogue between users and developers, helps to expose unstated assumptions, and triggers some of the inevitable requirements changes early in the process. Prototypes thus aid requirements engineering and reduce rebuilding.

♦ It reduces risk by making communication between users and developers more certain, by helping to determine a proposed design's unknown properties, and by providing a basis for assessing the feasibility and performance of alternative designs.

♦ It is the most feasible way to validate specifications. Validation attempts to ensure that all parties — clients and developers — interpret the specifications in the same way. Without this assurance, developers run a high risk of developing and testing software built on incorrect specifications. Prototyping should be integrated with the formulation and recording of specifications and the assessment of a design's feasibility and performance.

Prototyping can also provide many of the same benefits when a system's requirements change after delivery. For small changes, such as adjustments to screen formats, the delivered system can serve as a basis for diagnosing the problem and evaluating proposed changes. However, when the proposed changes are so drastic that they fundamentally alter a system's goals, the exist-

0740-7459/91/1100/0077/$03.00 © IEEE



DECIDING WHAT A SYSTEM SHOULD DO

Determining what a proposed system or enhancement should do is becoming a dominant problem in software development. Systems analysts must determine what functions a proposed system should provide to help its users carry out their tasks most effectively, subject to cost and performance constraints. This is difficult because analysts usually do not know at the outset who the users will be, what their responsibilities are, and how they will carry out their responsibilities.

Analysts, then, must determine a great deal of information about the users and their responsibilities by communicating with people, who usually share many unconscious assumptions and do not have an accurate understanding of the potential capabilities and limitations of software.

Introducing a new system also often triggers organizational restructuring, fundamentally changing how the organization does its business. This restructuring may drastically change users' responsibilities and procedures, thus triggering corresponding changes in what role they want the system to take. The system itself thus becomes a driving force for changes in its requirements.

The traditional approach to requirements engineering is to interview potential users and other stakeholders and prepare a requirements document, which is reviewed and modified until there are no more objections. This approach has not worked well in practice because users have not been effective in discovering requirements errors when they review the documents.

Some specific difficulties:

♦ *Ambiguity*. Reviewers (users) and analysts and developers interpret words differently, and the miscommunication is not detected.

♦ *Obscurity*. Documents are so complicated that reviewers never fully understand them, and hence miss some undesirable features.

♦ *Habitual assumptions*. Stakeholders overlook errors of omission because some of the missing information is such a fundamental assumption in their business that it is unconscious to them. Unlike developers who aren't familiar with the application, reviewers often can't imagine any other possibility.

♦ *Novelty*. When a new system stands to change business processes fundamentally, stakeholders cannot visualize all of its effects.

♦ *Assessing feasibility*. It is very difficult for developers to assess performance and costs without developing designs and measuring resource consumption.

♦ *Assessing effectiveness*. Without experimental evaluation, it is difficult for developers to predict the effectiveness of control strategies for embedded systems or decision-support information.

ing system may not be a good basis for evaluating them. In these cases, prototyping can reduce uncertainty and the number of times the operational system must be changed before a satisfactory result is obtained. This approach leads to fewer operational failures during the transition to the new version, better stability of operating procedures, and reduced retraining overhead.

To be useful, prototypes must be built rapidly and designed in such a way that they can be modified rapidly; designers use an iterative process of demonstration and adjustment to improve their accuracy. Software tools facilitate and speed prototyping and help analysts formulate, understand, and communicate a proposed system's properties to users. A computer-aided prototyping environment should be integrated with tools for measuring, optimizing, and refining the prototype design into a production-quality product.

SOFTWARE VERSUS SYSTEM

There is a close relationship between software and system prototyping. Especially in the case of real-time systems, it is very difficult to separate the formulation processes for software requirements and the requirements for the larger system in which it will be embedded.

Many systems contain embedded control-software systems that must meet real-time constraints. Real-time constraints couple the embedded system's software and hardware design because response time depends on the number of instructions per second the processors can execute, the number of bits per second the network and storage devices can transfer, the number of instructions that must execute, and the amount of data a software action needs to complete.

To evaluate, optimize, and accept the entire system configuration, designers use behavioral models of the software system and any interacting external systems, together with capacity models for the host hardware. This systems-level evaluation is especially important for real-time systems because the feasibility of the entire system remains in doubt until all three factors are specified and their interactions evaluated.

System prototypes. Analysts use system prototypes to establish rough feasibility assurances early in design, identify the aspects of the design that most affect the feasibility of the entire system, and track and focus attention on critical areas as the design becomes more solid, more refined, and less risky. Analysts must prototype the entire embedded system, not just the hardware or software, to assess design decisions on resource allocation and system performance.

The result is a hybrid prototype that models subsystems at different levels of detail: The parts of the system that will run on existing hardware are eval-

uated on actual equipment; parts that will run on new hardware are evaluated on software simulations of the hardware.

In this context, prototyping answers questions about resource allocation relative to the feasibility of timing constraints. There is a trade-off between software function, required response times, and hardware resources. To ensure proper system integration, analysts should explore the hardware necessary to support fixed software functions and timing constraints — and the combinations of software functions and response times that a given hardware configuration can support — *before* they commit to particular formulations of either hardware or software requirements. In current practice, a real-time system's hardware and software components are often developed independently, each based on separate and fixed requirements.

Measuring a prototype's properties helps designers when parameters of the hardware configuration can be varied to optimize a given software design, or when software functions can be varied to best use a fixed hardware configuration.[1] To be effective, designers must be able to evaluate such parameterized hardware models and portable software representations on various hardware configurations without modification.

THROWAWAY VERSUS EVOLUTIONARY

Before the availability of computer-aided design, prototype construction was done with quick-and-dirty manual coding and the elimination of "luxuries" like design documentation and written requirements. Manual construction did not work very well: It was neither very rapid nor inexpensive, and it resulted in prototypes that could not be changed very effectively when radical reformulations were required.

Prototyping approaches have evolved significantly. Today, two main approaches are used, throwaway and evolutionary.

Throwaway. Throwaway prototypes are sometimes perceived as a waste of effort, and there is some justification for this point of view. The simple-minded, management argument is that developing code that will be thrown away is a waste of resources. It is true that prototype code is often too inefficient and insufficiently general to be directly incorporated into a final product. But this argument ignores the fact that production-quality code often must be discarded because it is based on incorrect requirements. It is more cost-effective to correct the requirements by evaluating and discarding a relatively inexpensive prototype instead of an expensive implementation.

In addition, those who reject throwaway prototypes fail to recognize that their main contribution is not code, but the insight they give analysts into correct system behavior and the structure of a feasible design. However, a prototyping effort should produce more tangible results than just improved understanding. If it doesn't, the lessons learned may be imperfectly transferred to the final product and the insight gained may disappear when analysts change jobs.

Reliance on throwaway prototypes signals insufficient technological support for recording, transforming, tracking, and implementing specifications and designs.

Evolutionary. The availability of powerful design environments has given rise to evolutionary prototyping, in which a series of prototypes is produced that converges on an acceptable version of system behavior via client feedback from prototype demonstrations.[2] Parts of the description and design of each version are reused in the next version to the extent that the two versions share common requirements, subfunctions, and data.

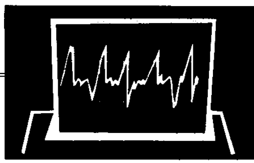
> Prototyping approaches have evolved a great deal. The two most popular today are throwaway and evolutionary.

The number of iterations required depends on many factors, including the skill of the analysts, their familiarity with the application domain, the complexity of the intended system, and how closely the required real-time performance approaches the target hardware's maximum capability. The accuracy of the requirements model improves with the number of iterations, which in practice is limited by available time and funds.

Once the series has converged, the result is turned into a software product by transformations that simplify and then optimize the design and code.[3] The transformations can dramatically change the code's appearance, although its principles of operation remain essentially the same.

Transformations require explicit representations of specifications and the high-level design that can be processed mechanically. If the transformation from prototype to production code is too laborious, the temptation will be to fix faults directly in the production code to meet deadline pressures. This will cause the production version to diverge from the prototype, and the prototype will gradually be abandoned.

If the prototype is not abandoned, it can be used as a starting point when maintenance efforts result in new re-



quirements. It is better to base maintenance changes on the flexible prototype instead of on the optimized implementation because transformations introduce conceptual dependencies that increase the fraction of the code affected by a requirements change.

STATE OF TECHNOLOGY

The most important emerging technologies are prototyping languages, support for reuse and program generation, and decision support for designers.[4]

Prototyping languages. A prototyping language defines an executable system model with both black-box and clear-box descriptions. A prototyping language is not obligated to give detailed algorithms for all system components as long as the model is descriptive and executable. Today, prototyping languages are distinct from specification, design, and programming languages.

Among the most important emerging technologies are prototyping languages and support for reuse and design decision-making.

♦ Specification languages record both external interfaces in the function-specification stage and internal interfaces at the highest abstraction levels during architectural design. They are also used to verify the correctness and completeness of a design or implementation.

Specification languages need not be executable and need not support clear-box descriptions. However, they must support formal reasoning, which is a lower priority for prototyping languages. Many specification languages contain unrestricted quantifiers because they are useful in proof systems and provide expressive power to designers and analysts. However, languages containing such quantifiers cannot be completely executable because they can define functions that are not computable. Executable specification languages must therefore have less expressive power than unrestricted specification languages. Executable specification languages can be used to prototype system function-ality, but most of them are weak on evaluating a proposed design's performance.

♦ Design languages are used to record conventions and interconnections during architectural and module design. They are usually not executable because they do not specify computations in complete detail.

♦ Programming languages are designed for efficient execution. Unlike prototyping languages, programming languages usually require algorithms and data structures to be defined completely before they can be executed, and usually do not record requirements, specification, and design information. A prototyping language is far more likely than a programming language to include default values for design parameters, and so incompletely specified functions can be executed and modifications can be made more easily.

The main challenge in designing a prototyping language is how to execute partial descriptions. This support can be provided by reusable code, transformation templates, and systems of default assumptions. However, to be useful the default assumptions must correspond to reasonable designs most of the time.[5] Better ways to support partial descriptions would be very useful — this is one of the central issues in prototyping research.

Reuse and program generation. Software reuse is critical for speeding implementation and for executing behavioral specifications that do not produce algorithms and data structures.

Evolutionary prototyping naturally emphasizes reuse at all levels. To develop and modify prototypes quickly, the designer must be able to easily combine compatible sets of reusable parts at various levels and in different ways. In addition to reusable functions and data types, designers need reusable templates for combining parts into larger structures with predictable pro-perties.

Sets of reusable parts are dependent on the application so making reuse work requires systematic and long-range planning, investment, and coordination by different organizations developing the same kinds of applications. So, in addition to technical issues, like component classification, retrieval, and integration, reuse involves managerial, economic, and cultural issues.

Domain models. If several systems for an application domain already exist, analysts can evolve a domain model from the experience gained in development. To reduce the investment required up front, the domain model can be incrementally extended to cover each new system in the domain as it is developed. This way, when a designer reuses parts of a domain model's requirements, design, and code, they get coverage of all the systems developed to date.

The main challenges in evolving domain models are how to automatically recognize two concepts that are variations on the same theme and how to generalize previous designs and code so that they cover all cases.

Once a domain has been well-explored, designers will be able to use generic domain models and their associated generic designs and generic programs to instantiate new prototypes. This will let them focus on determining the appropriate instantiation parameters. This goal of simplification is behind the recent efforts to develop domain-specific architectures.

However, prototyping is needed most in domains that are not well-understood. In these domains, reuse is much more difficult because components are being reused in contexts that were not anticipated when they were created.

Automated retrieval. Successful reuse in prototyping depends critically on the development of automated component retrieval. If we want to use components in new contexts, then retrieval technology must go beyond keyword and multiattribute paradigms. To accomplish this, some researchers focus on syntax alone; others try to exploit both syntax and semantics.

Successful retrieval will have to employ all these techniques, using the shallow but fast methods to reduce the number of candidate components and the more sophisticated techniques to improve selectivity and accuracy. Fast filtering can be based on a function's type signature and on test cases that eliminate candidates with clearly inappropriate behavior. Once the set of candidates is reduced to a manageable size, symbolic processing and theorem-proving techniques can rank the remaining components or certify that some of them do in fact meet the requirements.

Program generation. The set of potentially reusable programs is practically infinite, so each item in the reuse database should represent an unbounded family of generic components rather than an individual program. Generic components take (unboundedly) less work to create than the set of all possible individual instances, require less storage space, and are (unboundedly) more likely to be reused.

Program generation is therefore an important part of reuse: In addition to locating the proper reusable template, the retrieval process must also instantiate a generalized template that generates the specific program the designer needs. Ada's generic modules are a step in this direction, but much more powerful and flexible program-generation schemes are both possible and desirable.

Today's program-generation tools are rough models for the reusable templates of tomorrow. Progress has been made in domains like graphical user interfaces, formal-notation parsers, and syntax editors. We need more declarative ways to define program-generation schemes, more general methods to design such schemes for more varied applications, and methods and tools for analyzing such schemes. We also need ways to certify that all programs generated by a scheme have the specified properties and ways to satisfy a reuse query by identifying and generating the appropriate program-generation scheme if it exists.

Decision support. A prototyping environment's interface should shield the designer from data-management details and the boundaries between its tools. Technologies like Inter-Views and Idraw,[6] for creating graphical interfaces, help build tools that can do that. Just as important are syntax editors and attribute-grammar technologies,[7] which go beyond user-friendly interfaces. These technologies can automatically propagate design constraints to ensure consistency and forward-chain inferences to fill in the more mundane consequences of a designer's decisions.

Other aspects of decision support are technologies to manage the prototype's evolution,[8] generate scenarios for prototype demonstrations that expose unresolved issues, monitor and evaluate prototype execution, optimize prototype design, and analyze the consequences of timing constraints.[9]

Prototyping is an attractive approach to systems development, but its practical usefulness depends on technologies for computer-aided design and analysis. Progress to date has shown that it is feasible to develop this technology,[10] but a great deal of research and development remain before this technology realizes its full potential. ♦

ACKNOWLEDGMENTS

This report was developed out of a June workshop whose participants were members of the *IEEE Software* editorial and industry advisory boards. The editorial board participants were Peter Anderson, Rochester Institute of Technology, and Young-fu Chang, AT&T Bell Labs. The industry advisory board participants were Takashi Kojo, NEC; Hiroshi Isobe, Hitachi Ltd.; Pertti Lounamaa, Nokia Research Center; Tomoo Matsubara, independent consultant; Melissa Smartt Murphy, Sandia National Laboratories; Jack Wang, AT&T Bell Labs. Other participants were industry liaison Robert Lai, Software Productivity Consortium, and Takashi Arano, University of Illinois.

REFERENCES

1. H. Osborne, "Update Plans," *Proc. Int'l Hawaii Conf. System Sciences*, IEEE CS Press, Los Alamitos, Calif., 1992, pp. 488-496.
2. Luqi and V. Berzins, "Rapidly Prototyping Real-Time Systems," *IEEE Software*, Sept. 1988, pp. 25-36.
3. V. Berzins, Luqi, and A. Yehudai, "Using Transformations in Specification-Based Prototyping," *IEEE Trans. Software Eng.*, to appear.
4. Luqi and M. Ketabchi, "A Computer Aided Prototyping System," *IEEE Software*, Mar. 1988, pp. 66-72.
5. Luqi, "Computer-Aided Software Prototyping," *Computer*, Sept. 1991, pp. 111-112.
6. M. Linton, J. Vlissides, and P. Calder, "Composing User Interfaces with Inter-Views," *Computer*, Feb. 1989, pp. 8-22.
7. T. Reps and T. Teitelbaum, *The Synthesizer Generator: A System for Constructing Language-Based Editors*, Springer-Verlag, New York, 1988.
8. Luqi, "A Graph Model for Software Evolution," *IEEE Trans. Software Eng.*, Aug. 1990, pp. 917-927.
9. Luqi, "Real-Time Constraints in a Rapid Prototyping Language," *J. Computer Languages*, Apr. 1992, pp. 77-103.
10. Luqi, "Computer-Aided Prototyping for a Command-and-Control System Using CAPS," *IEEE Software*, Jan. 1992, pp. 56-67.