Software Evolution Through Rapid Prototyping

Luqi
Naval Postgraduate School

Software evolution refers to all activities that change a software system, including responses to requirements changes, improvements to performance or clarity, and repairs for bugs. The older term "maintenance" refers to the same activities in the context of the traditional life cycle, with a connotation that maintenance is done after the initial development. In more recent process models such as rapid prototyping, evolution activities are interleaved with the initial development and continue after the delivery of the initial version of the system. Since software evolution accounts for more than half of the total software cost, great interest has focused on reducing the effort required. Prototyping provides one promising approach to achieving this goal.[1,2]

In this article, a prototype is a concrete executable model of selected aspects of a proposed system. Rapid prototyping is the process of quickly building and evaluating a series of prototypes.

Figure 1 illustrates the iterative prototyping cycle. The user and the designer work together to define the requirements and specifications for the critical parts of the envisioned system. The designer then constructs a model or prototype of the system in a prototype description language at the specification level. The resulting prototype is a partial representation of the system, including only those attributes necessary for meeting the requirements. It serves as an aid in analysis and design rather than as production software.

Rapid prototyping supports software evolution as well as initial development. Computer-aided prototyping tools and object-based methods support evolution of both prototypes and production software.

During demonstrations of the prototype, the user evaluates the prototype's actual behavior against its expected behavior. If the prototype fails to execute properly, the user identifies problems and works with the designer to redefine the requirements. This process continues until the user determines that the prototype successfully captures the critical aspects of the envisioned system.

The designer uses the validated requirements as a basis for designing the production software. Additional work is often needed to construct a production version of the system. For example, the prototype

(1) might not include all aspects of the intended system,
(2) might have been implemented using resources that will not be available in the actual operating environment,
(3) might not be able to handle the full workload of the intended system, or
(4) might meet its timing constraints only with respect to linearly scaled simulated time.

Experience with production use of a delivered system often leads to new customer goals, triggering further iterations of the prototyping cycle.

The traditional model of software development relied on the assumption that designers could stabilize and freeze the requirements. In practice, however, the design of accurate and stable requirements cannot be completed until users gain some experience with the proposed software system. Thus, requirements often must change after the initial implementation.

In traditional approaches, these requirements changes trigger changes to the production version of the system during the maintenance phase. In prototyping approaches, an appreciable fraction of the requirements changes trigger changes in a prototype version of the system. This is useful because a prototype description

(1) is significantly simpler than the production code,
(2) is expressed in a notation tailored to support modifications, and
(3) is suitable for processing by software tools in a computer-aided prototyping environment.

These factors make it possible to modify a prototype more easily than a production version of the system. They make prototyping especially attractive for unfamiliar application areas with uncertain requirements.

In the approach to rapid prototyping we will look at here, software systems are delivered incrementally and requirements analysis continues throughout the process, interleaved with implementation and evolution.[3] We will focus on reducing requirements errors through prototyping before undertaking the incremental implementation effort for each deliverable version of the system. Incremental delivery lets users gain early experience with the software in the actual production environment. It also lets developers adjust the requirements to reflect the effects of the initial versions of the system on the customers' perceptions of their problems. Thus, incremental delivery extends the advantages of prototyping to the production environment.

The problems of software evolution are especially prominent during rapid prototyping because prototypes are subject to frequent and repeated changes. The potential benefits of prototyping depend critically on the ability to modify the prototype's behavior with substantially less effort than required to modify the production software. Computer-aided prototyping and object-based prototyping provide the solutions to this problem. Computer-aided prototyping provides mechanical assistance, and object-based prototyping provides conceptual simplicity.

Computer-aided rapid prototyping improves the efficiency and accuracy of evolutionary development by introducing software tools that assist the designer in constructing and executing the prototype quickly and systematically. These tools make it attractive to use prototypes for evaluating evolutionary changes after a version of the system has been delivered as well as for the initial version.

Object-based prototyping is based on data abstraction and inheritance. Objects encapsulating the data in the prototype system serve as the basis for design and implementation. Since the data in an application is generally more stable than the processing steps, this leads to system descriptions that are easier to modify than those based primarily on procedural abstractions. Inheritance helps to reduce the labor involved in constructing a system by allowing inclusion of common aspects of the code in many different contexts without explicitly repeating the details. Objects also provide convenient components for code reuse, parallel execution, and version control. Thus, object-based approaches make prototypes more flexible and automation easier to achieve.[4]

Evolution based on the bare program code is very difficult or impossible to achieve, because we need information about the requirements, specifications, and design to change the code without damaging it. Most tools supporting evolution at the program level are primitive and language specific, such as facilities for generating cross-reference listings and for editing and storing different versions of program documentation. While such facilities can reduce the mechanical work involved, few software maintenance tools operate on the semantic level and few good ideas address the general software maintenance problem. To support the software evolution process, tools operating at the semantic level should help manage the relationships among the implementation, the prototype description, and the requirements.

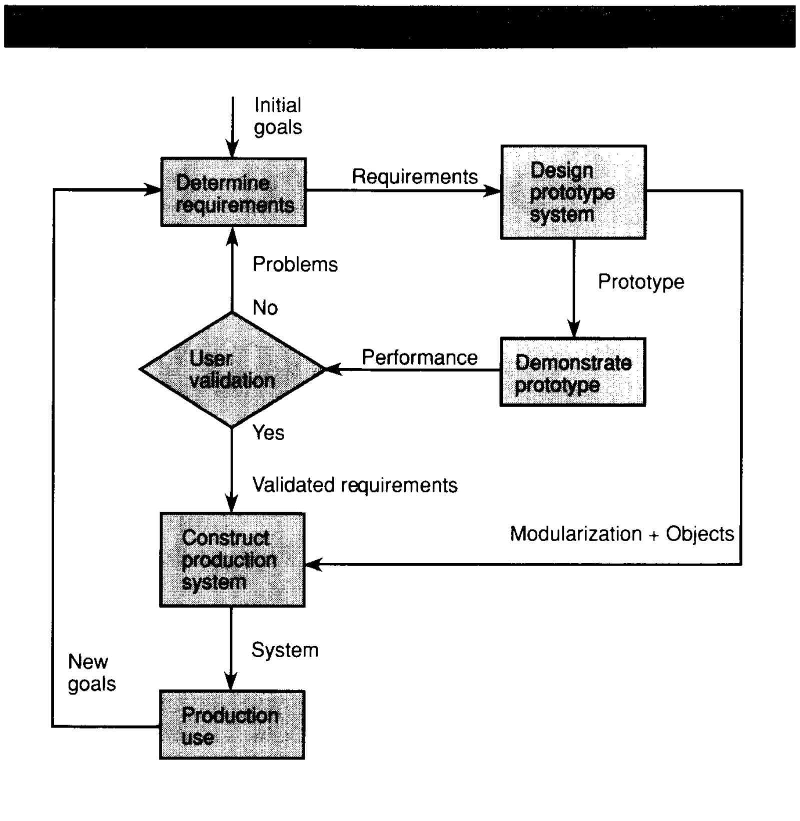


Figure 1. The prototyping cycle.

Computer-aided prototyping tools for evolution

An integrated set of computer-aided software tools, the Computer-Aided Prototyping System (CAPS),[5] has been designed to support prototyping of complex software systems, such as control systems with hard real-time constraints. The requirements for such systems are especially difficult to determine, and their feasibility is hard to establish without constructing an executable model of the envisioned system.[6]

If carried out manually, the prototyping process has limited benefits because of the time and effort involved. CAPS can increase the leverage of the prototyping strategy by reducing the effort the designer puts into producing and adapting a prototype to perceived user needs.

The evolution of a prototype starts after one pass through the prototyping cycle shown in Figure 1. The analysts have determined the initial requirements by talking to the customer, constructed an initial prototype, and demonstrated it to the customer, who finds some of the prototype's behavior unacceptable and requests modifications.

Initially, the facilities provided by CAPS help adapt the prototype to the new requirements. We can implement modifications to the production software by using CAPS to

(1) add changes to prototype systems,
(2) retrieve software components from the software base,
(3) generate production code if needed,
(4) assemble production systems through the prototyping cycle, and
(5) manage the process using the prototyping database.

The main components of CAPS are a special prototyping language and a set of tools, illustrated in Figure 2. The main subsystems of CAPS are the user interface, the software database system, and the execution support system. The rest of this section describes these components in detail.

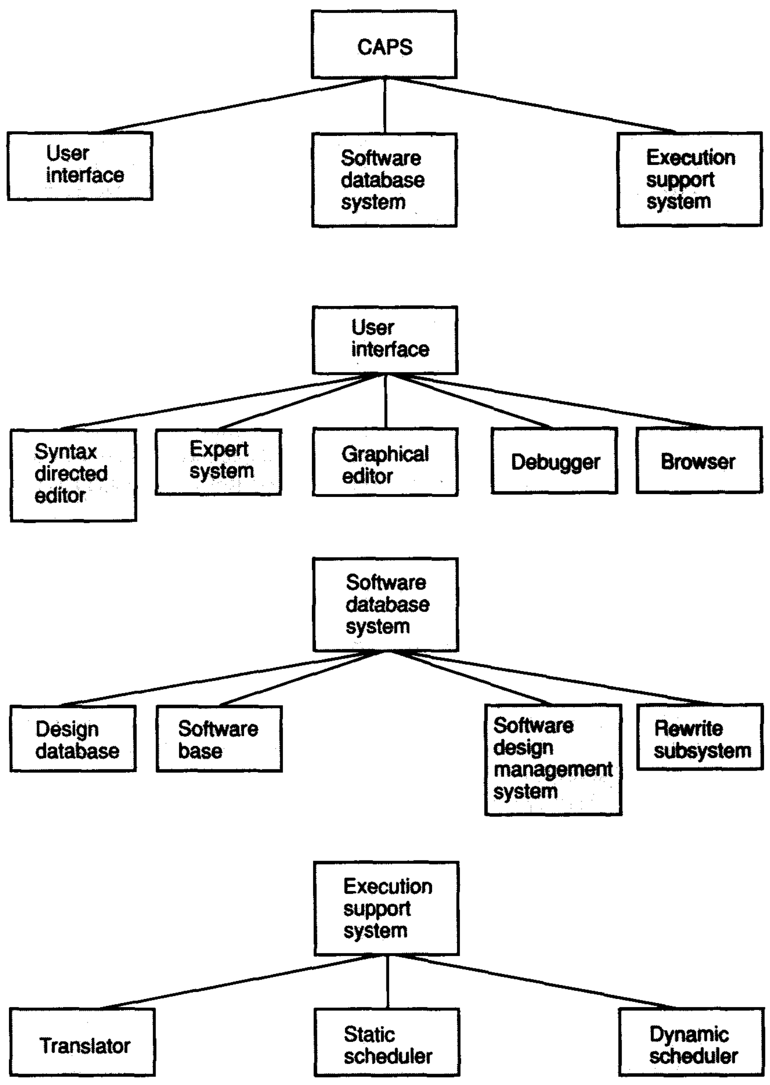


Figure 2. Main Computer-Aided Prototyping System tools.

Prototyping language features supporting modifications. CAPS tools communicate by means of the Prototype System Description Language (PSDL),[7] which integrates the tools and provides the prototype designer with a uniform conceptual framework and a high-level description of the system. PSDL supports frequent design modifications by meeting the following subgoals.

Modularity. The language must make it easy for the system designer to create a prototype with a high degree of module independence and to preserve its good modularity properties across many modifications. Good modularity is essential for easy modification.

An experimental study showed many of the problems that arise in modifying software result from interactions between widely separated pieces of code.[8] Locality of information was an important design goal of PSDL. The underlying computational model was chosen to make all interactions between components explicit. This model supports a system decomposition criterion that combines dataflow and control flow considerations.[2]

Good modularity means the prototype should be realized by a set of independent modules with narrow and explicitly specified interfaces. PSDL supports this concept via operators and data streams. An important property of the language is that two distinct operators can communicate or affect each other's behavior only by means of the data streams explicitly connecting them, either directly or indirectly.

This locality property is important for maintenance. It allows the set of modules that can potentially interact with a given module to be determined through a simple mechanical analysis of the dataflow network. This allows the software tools to guarantee that all aspects of a proposed change have been covered. The locality property also encourages designs containing an independent component for each major design decision. Such designs are easier to modify because the information required to change a design decision is localized in one region of the code.

The locality property is embodied by the PSDL scoping rules and mechanically enforced. The implementation of an operator can only refer to the explicitly declared input and output streams of the operator and to data streams local to the implementation of the operator. Implementations of operators representing state machines can contain closed loops consisting of local data streams.

Simplicity. The language should be simple and easy to use. PSDL is simple and easy to use because it contains a small number of powerful constructs. Designs are described in PSDL as networks of operators connected by data streams.

Such networks can be represented as dataflow diagrams augmented with timing and control constraints. The user interface uses the diagrams to provide a convenient means for presenting the system structure to the designer. The operators in the network can be either functions or state machines. The data streams can carry exception conditions or values of arbitrary abstract data types.

Reuse. The language should be suitable for specifying the retrieval of reusable modules from a software base. PSDL supports reusable components with uniform specifications suitable for retrieving modules from a software base. The specification part of a PSDL component contains several attributes that describe the interface and behavior of the component. These attributes help automatically generate a uniform specification for the reusable component.[9] These uniform specifications are used both for retrieval of reusable components and for organizing the software base.

Adaptability. The language should support small modifications to the behavior of a module without the need to examine its implementation. PSDL supports small modifications to modules by means of control constraints. We can use control constraints to impose preconditions on the execution of a module, to add filters to the output of a module, to suppress or raise exceptions in specified conditions, and to control timers. These facilities allow small modifications to the behavior of a module to be expressed independently of its implementation.

For example, a common problem discovered in prototype demonstrations is that an operator has the intended behavior most of the time but not always. The PSDL control constraints governing conditional execution of operators can help solve the problem. We could add a control constraint in the form of an input guard predicate, where the guard predicate describes the circumstances in which the execution of the operator will produce the intended result and disables the execution of the operator in cases where it would not. This allows the addition of another operator for producing the correct output in the remaining cases, controlled by a complementary guard predicate.

Abstraction. The language should support a set of abstractions suitable for describing complex software systems with real-time constraints. PSDL provides abstractions suitable for describing large systems and real-time constraints. These include the nonprocedural control constraints mentioned above, timing constraints, timers, functional abstractions, and data abstractions.

Examples of timing constraints include the maximum execution time, the maximum response time, and the minimum calling period. Timing constraints implicitly determine when operators with hard real-time constraints will execute. This simplifies evolution by removing explicit scheduling decisions from the design, thus allowing a software tool rather than the designer to handle rescheduling caused by design changes.

Requirements tracing. The language should support requirements tracing. PSDL supports requirements tracing by means of a construct for declaring the requirements associated with each part of the prototype. Requirements tracing is important because the prototype must adapt to the changing perceptions of the requirements resulting from demonstrations of prototype behavior. The links between each requirement and the parts of the prototype realizing the requirement determine which parts of the prototype to modify when a requirement is changed or dropped.

To prevent the structure of the design from being corrupted by multiple modifications, we must remove parts of the code no longer supported by an updated set of requirements. This cannot be done safely unless the correspondence between the requirements and the code is recorded and kept up to date.

The facilities for recording requirements trace information in PSDL are used by software tools in CAPS to provide automated aid in maintaining and using this information.

User interface for interactive control of prototypes. The user interface aids evolution by providing facilities for entering information about the requirements and design, presenting the results of prototype execution to the customer, guiding the choice of which aspects of the prototype to demonstrate, and helping the designer propagate the effects of a change. The user interface consists of a syntax-directed editor with graphics capabilities, an expert system for communicating with end users, a browser, and a debugger.

The editor enables convenient entry of PSDL descriptions into the system while preventing syntax errors. It also supports displaying graphical summary views of the prototype, maintaining the requirements trace, and locating parts of the prototype design related to particular requirements or data streams.

The expert system provides a paraphrasing capability that generates English text from PSDL descriptions. This allows end users to directly examine the prototype without being familiar with PSDL.

The browser allows the designer to interact with the software database. It has facilities for retrieving and examining reusable components stored in the software database system.

The debugger allows the designer to interact with the execution support system. It has facilities for initiating execution of the prototype, displaying results or trace information, and gathering statistics about prototype behavior and performance. A facility for recording test case coverage information helps guide the choice of scenarios for a demonstration run.

The user interface helps the prototype design team identify the tasks required to update the prototype. The user interface maintains the correspondence between requirements and parts of the prototype, along with lists of unresolved new requirements and unresolved modified requirements. Whenever a member of the design team is ready for a new task, the system presents the lists and lets the designer pick an item to resolve. If the designer chooses a modified requirement, the interface returns a list of modules previously supporting the requirement and lets the designer check them off as they are adapted or determined to be still valid.

The effort required for this task coordination is minimized by presenting the lists as menus and allowing the designer to pick items using a pointing device. Choosing an item results in a summary view of the affected modules, which can be browsed and updated as required.

The user interface speeds up the process of adapting the prototype by

(1) helping to coordinate tasks performed by a team of designers,
(2) helping to focus the designer's attention on the information relevant to a task,
(3) providing summary views of the system or selected components, and
(4) locating all potentially relevant parts of the prototype.

Software database for managing descriptions and building blocks. The software database system consists of a design database, a software base, a software design-management system, and a rewrite subsystem. The design database contains the PSDL prototype descriptions for each software development project using CAPS. The software base contains PSDL descriptions and code for all available reusable software components. The software design-management system manages and retrieves the versions, refinements, and alternatives of the prototypes in the design database and the reusable components in the software base. The rewrite subsystem translates PSDL specifications into a normalized form used by the design-management system to retrieve reusable components from the software base.[9]

The components of the software database actively contribute to the process of adapting the prototype to new requirements. The software design-management system helps maintain the design history and locate relevant reusable software components. The design history consists of the relationship between each version of the requirements and the corresponding versions of parts of the prototype. This information is useful because the customer will sometimes retreat to previous versions of the requirements. Situations in which this might happen include cases where the customer gives up on an ambitious requirement in response to cost or performance estimates resulting from examination of the prototype. In such cases, parts of the requirements revert to previous configurations. The system helps restore the corresponding parts of the prototype to their previous configurations.

The design database also provides concurrency control functions that allow multiple designers to update parts of the prototype without unintentional interference. In the interest of minimizing delay, the design database will not lock out read-only access to any part of the design, even while the design is being updated. Instead, the system will allow examination of the previous version of the component, with a warning that a new version is currently in preparation. On request, the system will provide information about the reason for modification of the component (such as a new or modified requirement). Enhancements to alternative versions can be explored in parallel, thus speeding up exploratory evolution.

The software base provides reusable software components for realizing given PSDL specifications. The software base speeds up evolution by providing many different versions of commonly used components, making it easier to try out alternative designs. In the PSDL prototyping method,[2] modules are realized by three main mechanisms:

(1) Retrieval of a suitable component from the software base. The software base contains generic modules with parameters determined as part of the retrieval process. It also contains rules for matching a specification by means of a composite operator realized by a network of operators, at least one of which must be an available reusable component.[9] The retrieval mechanism can therefore perform some routine aspects of bottom-up design, freeing the designer from the need to be familiar with all the reusable components in the software base.

(2) Decomposition of the component into a network of simpler components. The designer does this if the component cannot be retrieved directly from the software base and if the component is sufficiently complex to benefit from decomposition into simpler parts.

(3) Direct implementation in a programming language. The designer does this if the software base does not contain a component that performs the required function with the required speed.

The essential problem in the organization of object-based databases for managing reusable components is to allow the representation and retrieval of an unbounded number of components given finite memory and processor speed. We must consider an unbounded number of components because software designs can contain arbitrary user-defined abstract data types, and, to be useful, the reusable components in the component database must be applicable to all of the types in this infinite set.

A practical approach to this problem regards the database as containing all the components that can be generated from a finite set of explicitly stored components by finite combinations of a set of primitive *component constructors*. Examples of component constructors are transformations that instantiate generic parameters or that create a composite component by interconnecting a pair of available components.

Retrievals from such databases will generally involve a limited degree of logical inference, to determine whether a component matching the query can be constructed from available components within a given limited number of constructor applications. Limits are needed to make sure retrievals will always terminate. These logical inferences are performed according to rules stored in the knowledge base associated with the component library.[9]

Execution support for demonstrating effects of changes. The PSDL execution support system contains a translator, a static scheduler, and a dynamic scheduler.[5] 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 allocation succeeds, all operators are guaranteed to meet their deadlines even with worst-case execution times. As execution proceeds, the dynamic scheduler invokes operators without real-time constraints in the time slots not used by operators with real-time constraints.

The execution support system helps speed up design changes by providing a localized view of the processes in the prototype, analyzing the prototype's timing properties, and providing the ability to quickly demonstrate the consequences of design decisions through prototype execu-

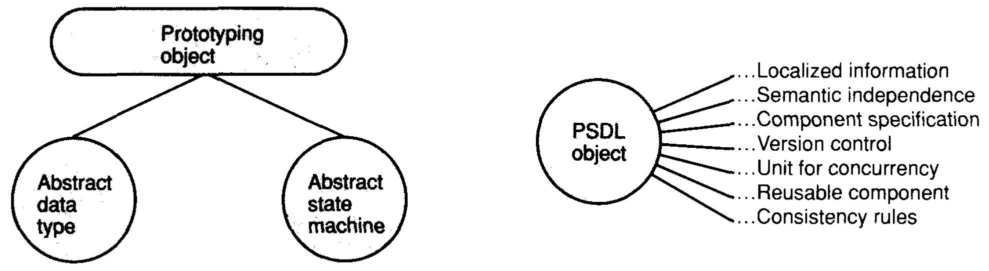


Figure 3. Objects and general properties.

tion. These features are especially important for prototyping real-time systems.

At the programming language level, implementations of real-time systems are difficult to understand because the instructions of several logically independent processes must often be interleaved to meet timing constraints.[10] PSDL presents a view to the designer in which logically distinct processes are represented as separate independent components. The PSDL execution support system contains a translator that mechanically transforms this independent representation into the corresponding programming language representation, adding the necessary interleaving in a fashion transparent to the designer.[11]

If the static scheduler succeeds in constructing a schedule, the operators in the schedule are guaranteed to meet their timing constraints even under worst-case operating conditions. 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 resolved by adding more processors.[12] These functions are important because the timing constraints in complex systems can have complicated interactions that are difficult to analyze manually.

Software evolution through rapid prototyping

CAPS supports software evolution through object-based prototyping and reusable software components. Object-based prototyping is the rapid construction of software systems using objects that encapsulate data as building blocks. PSDL includes two kinds of objects, corresponding to abstract data types (PSDL types) and abstract state machines (PSDL operators). Figure 3 shows the general properties of the PSDL objects.

The most important function of objects used in prototyping is to localize information. This design principle allows us to understand, analyze, and execute each object independently of other objects, reducing the conceptual complexity of the prototype system. Since the semantics of such objects is independent of the context in which they appear, they are likely to be reusable. They also provide a convenient basis for version control in an evolving system.

Objects can also serve as natural units of work in a parallel implementation, since they can execute without interfering with each other. Parallel implementations are attractive in systems with tight real-time constraints because multiprocessor schedules exist for many real-time constraints that cannot be met on a single processor.

One of the main difficulties of software evolution in traditional contexts is the lack of accurate requirements, specifications, and design documents.[13] We need precise documentation to reliably change the system. Especially for older systems, information other than the source code is often unavailable or obsolete because of the large amount of time and effort required to manually create and maintain it.

In PSDL, specifications and justification links to the requirements are part of the prototype description, and the implementation descriptions are provided at a design level. This information can be systematically recorded and kept in the tools during the prototyping process and automatically supplied by the tools during evolution. In other words, CAPS tools use the higher level information to aid the designer in modifying the prototype.

PSDL can describe both the prototype and the production versions of the system. A PSDL implementation has two parts: a skeleton consisting of the modules in the system and their interconnections, and a set of reusable components containing implementations of the atomic components in a conventional programming language such as Ada. The main activities in the system implementation phase involve refining partially defined facilities and optimizing implementations. These activities take place at both the PSDL level and the programming language level.

Refinements are initially expressed in PSDL by (1) adding more constraints to the specifications and retrieving new reusable components or (2) doing further decompositions to make the implementation correspond to the refined specification.

Optimizations are performed at the PSDL level by introducing alternative decompositions that eliminate unnecessary processing or allow more efficient algorithms.

The performance tuning process continues at the programming language level. There, efficient custom implementations for operators with tight real-time constraints or frequently executed non-time-critical operators are created and added to the software base together with corresponding PSDL specifications. This process maintains the correspondence between the implementation, design, specifications, and requirements. In addition, we can use the same tools and techniques to develop the production version of the system.

Changes to the production version of the system require changes in the PSDL specifications. We can meet the changed specifications by using reusable components from

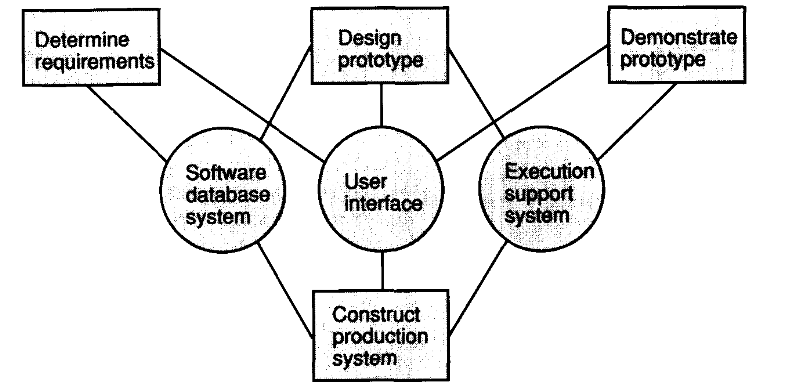


Figure 4. Tool usage in the prototyping cycle.

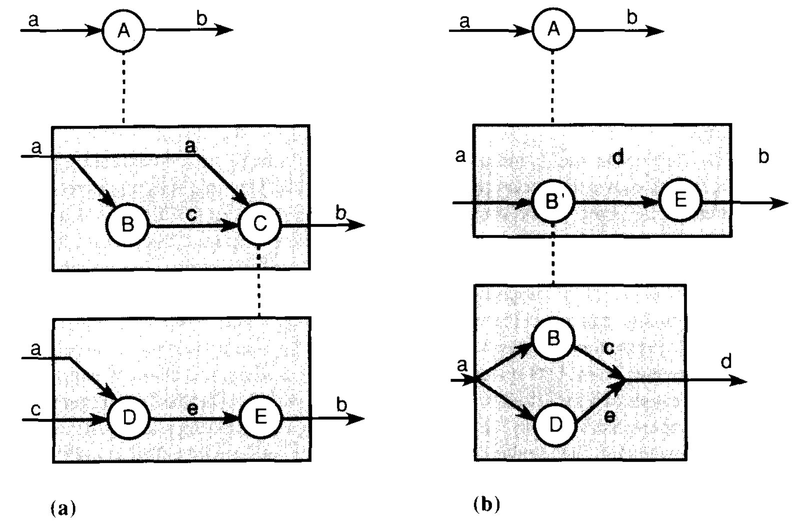


Figure 5. Regrouping the operators in a design, showing (a) the initial architecture and (b) the regrouped architecture.

the software base in a flexible manner. After the design stabilizes, we can optimize the modified portions of the system.

Figure 4 summarizes the phases of the prototyping cycle where each class of CAPS tools is used.

Designer's viewpoint. We can classify the modifications to a prototype performed by a designer as either static changes or dynamic changes. Static changes result from modifying the PSDL source code. They are tested by a complete regeneration of the executable model of the system. Dynamic changes are made using the debugging system during the demonstration run of a prototype. They provide immediate feedback to the customer about the effects of proposed alternatives. Both static and dynamic changes are necessary to effectively carry out the prototyping cycle.

Static changes are done off line, when the customer is not waiting and there is time for careful design, mechanical checking, scheduling, and translation. There are four kinds of static changes: regrouping, tuning an object, custom programming, and specification changes.

Regrouping refers to a change that rearranges a set of atomic operators. This kind of change localizes information and improves the logical coherence of a design. Figure 5 shows an example of this kind of change. Figure 5a shows the initial grouping, and 5b shows the modified grouping for a subsystem. The operators *B* and *D* are moved into the same subsystem *B*' because both of them use the same input stream *a*, and this stream is not needed in any other part of the system.

Regrouping simplifies the interfaces of the major subsystems and makes them more coherent. Exploratory prototyping often requires this kind of change because, in the initial stages, the functions of the proposed system are not clear. Once we know the parts of the system, the relationships between them become clearer. We then want to regroup the parts of the system so that related parts appear in the same subsystems and higher level groupings correspond to abstractions meaningful to the users.

Another common kind of regrouping transformation gathers all of the operators that use a state variable into a single state machine object, which then hides the state variable from the rest of the system.

Tuning refers to design changes that affect the implementation but not the specification of a composite object. Tuning is done at the PSDL level, by supplying an alternative decomposition for an object. The purpose is usually to simplify the implementation or to improve its performance. Figure 6 illustrates this kind of change, where 6a shows the initial decomposition for a composite object and 6b shows a simplified decomposition.

Custom programming refers to design changes that replace part of the implementation of the prototype system with an atomic object implemented directly in the programming language. The new atomic object produced in this way is added to the software base as a reusable component. While, in principle, changes in this category do not affect the specification of the object, in practice they might trigger some additional specification changes because the new object must fit into the software base.

The specification of the object might

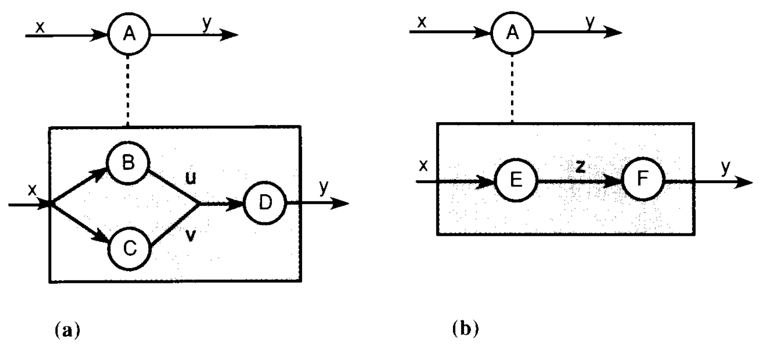


Figure 6. Tuning the implementation of an object, showing (a) the initial decomposition and (b) the simplified decomposition.

need refining to include additional constraints that distinguish it from similar objects already in the software base, since the specifications of the reusable components serve as keys (unique identifiers). This kind of refinement is needed if an object matching the specification of the subsystem is already present in the software base, was retrieved at an earlier stage, was included in the design, or was found lacking in some respect. For example, the original reusable component might perform the correct function but take too long to execute. The additional constraints added to the specifications describe the performance characteristics that distinguish the original implementation from a new, optimized implementation.

Specification changes are needed when the customer finds the demonstrated behavior of the prototype unacceptable. Consequently, the behavior of some objects in the prototype require adjustment. PSDL provides statements for recording which requirements justify each attribute of an object in the prototype. These links can be used in both directions, depending on the designer's working style. For example, the designer might be familiar with the design of the prototype and thus easily able to trace a complaint about an inappropriate response to a particular object. The system can automatically follow the requirements links to show the list of requirements supported by the offending object. The designer can then identify the subset of those requirements affected by the change, and the system can trace the requirements links in the other direction to generate a list of objects potentially affected by the change.

The CAPS system aids the designer in propagating the effects of changes by maintaining a list of operators potentially affected by a change. The system guides the designer through the process of reviewing the operators on the list by presenting a task menu.

A PSDL prototype has a hierarchical structure, which shows the decomposition of each composite object into more primitive objects. Specification changes must maintain the consistency of this hierarchy. If the specification of a subcomponent changes, the change can affect the specification of each composite object containing the subcomponent. The CAPS system adds all of the ancestors of a modified object in the subcomponent hierarchy to the list of objects reviewed by the designer. The system also has a set of heuristic rules for automatically propagating the effects of some types of specification changes, including changes to the maximum execution time, maximum response time, minimum calling period, and data types associated with the input streams and output streams of an object.

An example of automatic constraint propagation appears in Figure 7. In this example, the maximum execution time of the subcomponent B had to increase because it could not be implemented within the originally specified deadline. The object B is part of the implementation of the composite object A, as shown in Figure 7c. The operators in the graphical form of the implementation are annotated with the maximum execution times, and when the specification of B changes, the CAPS system automatically reflects the change in the implementation of a, as shown in Figure 7d. The constraint associated with maximum execution times requires the maximum execution time of a composite operator not to exceed the sum of the maximum execution times along the longest path in its dataflow graph. The change violates this constraint and causes the maximum execution time of the operator A to increase, as shown in Figure 7f.

Another reason for specification changes is to increase the probability that an object in the software base can be reused. A specification change designed to improve the reusability of a component usually involves a generalization, such as introducing some generic parameters for the object. This class of changes prevents cluttering the software base with large numbers of similar objects.

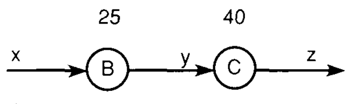
Dynamic changes are made using the debugger as the prototype executes, to quickly and roughly test out new ideas without going through complete recompilation and rescheduling. A classical problem caused by installing patches using debuggers is the danger of an undocumented divergence between the executable version of the system and the source code. CAPS protects the designer against this possibility by maintaining a record of the dynamic changes. This record allows the original version and each of the alternatives explored in a demonstration run to be restored at will during the demonstration. It also allows automatic insertion of selected changes into the PSDL source code.

The set of dynamic changes supported by CAPS includes standard debugger functions such as examining and modifying the contents of data streams, displaying execution traces, setting breakpoints, and setting conditional data traps. Some less conventional capabilities include controlling the real-time clock, selectively disabling some threads of a parallel implementation, inserting parallel consistency checking operators, modifying triggering conditions of operators, and saving execution states so that the prototype can be restarted many times from the same intermediate point.

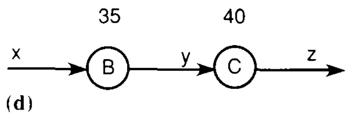
To allow meaningful debugging, the deadlines of the time-critical operators have to be set back by the *skew time*, which equals the time spent in the debugger plus the amount of time required to restore the execution state and resume execution. The execution support system dynamically monitors the execution of the prototype

```
OPERATOR B INPUT x: real OUTPUT y: real
  MAXIMUM EXECUTION TIME 25 ms
END
```
(a)

```
OPERATOR B INPUT x: real OUTPUT y: real
  MAXIMUM EXECUTION TIME 35 ms
END
```
(b)



(c)



(d)

```
OPERATOR A INPUT x: real OUTPUT z: real
  MAXIMUM EXECUTION TIME 65 ms
END
```
(e)

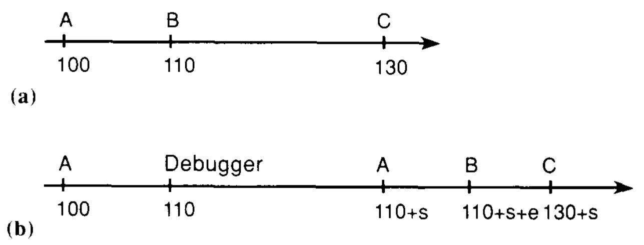
```
OPERATOR A INPUT x: real OUTPUT z: real
  MAXIMUM EXECUTION TIME 75 ms
END
```
(f)

Figure 7. Automatic constraint propagation, showing (a) the original subcomponent specification, (b) the modified subcomponent specification, (c) the original supercomponent implementation, (d) the modified supercomponent implementation, (e) the original supercomponent specification, and (f) the modified supercomponent specification.

and automatically traps to the debugger when a time-critical operator misses its deadline. At that point the designer can examine the state of the system to see if there is something wrong. The designer has the option of (1) resetting the real-time clock to give the operator some extra CPU time before its deadline arrives, (2) allowing it to exceed its deadline by a specified amount, or (3) abandoning execution.

Consider the example in Figure 8. In the example, the execution of operator *A* has exceeded its allotted time and control has passed to the debugger. If the designer chooses to allow *A* to pass its deadline and complete execution, then operator *B* will be delayed by the excess execution time *e* and will have that much less time to complete before its time slot runs out at time 130+*s*. Such an experiment will help determine whether the circumstances that cause *A* to exceed its deadline allow *B* to finish earlier to compensate. If the designer had decided to reset the real-time clock instead, the operator *A* would have gotten some invisible extra time, so *B* would still have a full 20 time units to meet its deadline after *A* completed execution.

Selectively disabling some threads of a parallel program helps get the maximum possible information out of a demonstration run. Figure 9 shows an example of this situation. Suppose that the output stream *c* from operator *A* is hopelessly in error and that a data trap has detected the problem and passed control to the debugger. If the designer is unable or unwilling to substi-



maximum_execution_time(A)=10
maximum_execution_time(B)=20
s=skew time
e=excess execution time for A

Figure 8. Debugging a real-time prototype. (a) Scheduled execution. (b) Actual execution.

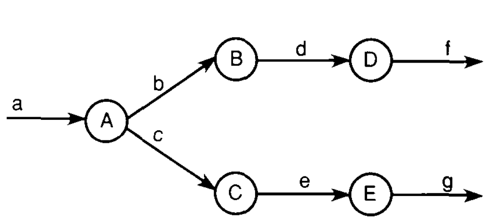


Figure 9. Disabling a parallel thread.

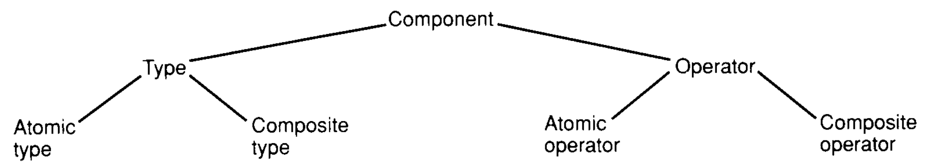


Figure 10. Class structure for prototype components.

```
CLASS Component
  SUPERCLASSES { }
  ATTRIBUTES
    name: string
    generic: set[parameter]
    description: string
    support: set[requirement]
END Component

CLASS Operator
  SUPERCLASSES {Component}
  ATTRIBUTES
    input, output, states: set[parameter]
    max_exec_time, max_resp_time,
      min_call_period, period,
      finish_within: time
    spec: assertion
END

CLASS AtomicOperator
  SUPERCLASSES {Operator}
  ATTRIBUTES
    language, code: string
END

CLASS CompositeOperator
  SUPERCLASSES {Operator}
  ATTRIBUTES
    dfd: graph
    constraints: set[control_constraint]
END
```

Figure 11. Attributes of prototype components.

tute a correct value for c, he or she can temporarily disable the execution of C and restart the system to see the results of executing B and D using the value of b, which was not affected by the fault in A. This is equivalent to removing a faulty data value from the data stream c and will not cause any further faults if operator E is triggered by the arrival of new input data. If E is triggered by a temporal event, then disabling C might cause another fault if E requires a fresh value of e every time it executes.

Consistency checking, as performed by dynamically inserted data traps, must not interfere with the timing properties of normal prototype execution. In a single-processor implementation this requires stopping the real-time clock while the consistency checks are performed. In multiprocessor implementations such consistency checks can be performed in real time if enough processors are available and if the consistency checks are shorter than the primary computations.

Tool viewpoint. The tools in CAPS are organized around an object-oriented database management system used to realize the design database and the software base.[14] The components of a prototype description are instances of the subclass hierarchy shown in Figure 10. Selected attributes of a representative subset of these object classes appear in Figure 11.

To effectively support modifications, the tools in CAPS must address several consistency problems. Some of the problems of consistency with respect to the subcomponent hierarchy were discussed in the previous section. Another problem is maintaining consistency between the graphical and textual views of a prototype.

Graphical views arise as part of the implementation of a composite object, while text views arise for the specification parts of the immediate components of a composite object. The graphical view and the textual view contain different forms of the same information, so when the designer changes one view, the other one must be automatically updated to maintain consistency. This process requires some care, since each view contains information not visible in the other view. Different strategies are appropriate for each.

If a new operator is added in the graphical view, it will lack information such as control constraints and the types of the values on its input and output streams. Since control constraints are optional, that part of the text view can be left empty. Since the data types of streams are not optional and an accurate value is not available, the corresponding slot in the textual view must be filled by a *completion term*. A completion term is a special value recognized by the syntax-directed editor as a placeholder for a missing part of the prototype. Its distinctive display reminds the designer that more information must be supplied before the prototype can execute.

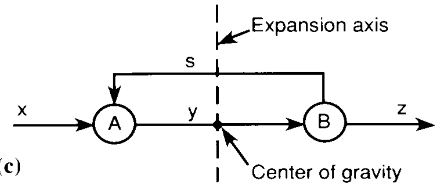
If a new operator is added in the text view, it will lack information about the position of the corresponding icon in the graphical view. Since an icon cannot be displayed without choosing a particular position, a heuristic generates a default position.

One method for doing this is to put the new icon at the center of gravity of the sources and destinations of all the inputs and outputs of the new operator. Next, cut the old display into two parts with a horizontal line through the position of the new icon, moving the old icons outward until the new icon has a minimum clearance from all the old ones. Then reconnect the broken arrows. Finally, do the same with a vertical line through the position of the new icon.

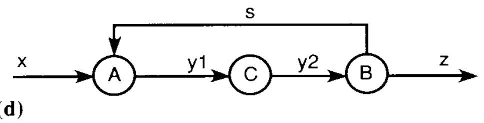
Figure 12 illustrates this process. Figure 12a shows the old text view, 12b shows the new text view, 12c shows the old graphical view with the default position and vertical expansion axis, and 12d shows the new graphical view. This heuristic has a global effect, since potentially it can transform the layouts of all the objects in the graphical view.

```
OPERATOR A INPUT x,s:t OUTPUT y:t END
OPERATOR B INPUT y:t OUTPUT z,s:t END
```
(a)

```
OPERATOR A INPUT x,s:t OUTPUT y1:t END
OPERATOR B INPUT y2:t OUTPUT z,s:t END
OPERATOR C INPUT y1:t OUTPUT y2:t END
```
(b)



(c)



(d)

Figure 12. Consistency between text and graphical views, showing (a) the initial text view, (b) the modified text view, (c) the initial graphical view, and (d) the modified graphical view.

A change to the specification of an operator invalidates its implementation. Such changes signal the project database management system to save the previous version of the operator. This prevents losing the results of previous efforts should the designer later want to restore the previous version. The tree of suboperators rooted at the modified operator is then removed from the new version of the prototype, and the software base is searched for an implementation of the new specifications. Since the operators containing the modified operator are invalidated by such a change, the system adds them to a list of action items for the designer.

Evolution of a hyperthermia system

To illustrate some typical prototype modifications, this section discusses part of the prototyping cycle for a hyperthermia system. The hyperthermia system treats brain tumors by using microwaves to heat the affected area to a temperature that will kill tumors but not normal tissue. An embedded software system controls the microwave power based on feedback from a temperature sensor inserted into the patient's brain. The goals of the prototyping effort are to evaluate the safety of the proposed system and to establish the feasibility of implementing the required control functions.

The initial PSDL specification for the top level of this system appears in Figure 13. The details of a PSDL decomposition for the initial version of this prototype can be found elsewhere,[7] so I will not repeat them here, although the information they contain is necessary to make the prototype executable. I have provided informal descriptions of lower level details as needed to explain the effects of the proposed modifications. The second-level decomposition of the brain tumor treatment system contains a simulated patient and the proposed software system for controlling the microwave power level.

Demonstration of the initial version of the prototype led to a question about the safety of the proposed system. In the initial version of the prototype design, information about the size of the tumor was extracted from the patient chart using an operation of the abstract data type *medical_history* called *get_tumor_diameter*. This information determines the initial microwave power level. The *get_tumor_diameter* operation raises an exception called *no-tumor* if the patient's chart does not contain a description of a tumor in the patient's brain. The response to the exception in the initial version of the prototype sets the microwave power level to zero and issues an immediate *treatment_finished* signal, which is plausible given the initial system interface specified in Figure 13. However, when this behavior was demonstrated, the potential users of the system pointed out that such a response can hardly be considered safe. If a healthy patient was mistakenly sent for hyperthermia treatment, the system would produce a response indicating something was wrong (an early *treatment-finished* signal) only after the *start_treatment* switch was pressed, which happens only after the temperature probe has been inserted into the patient's brain. A safe design should prevent such a dangerous procedure if not medically necessary.

Figure 14 shows a PSDL specification of the revised top level of the prototype, and Figure 15 shows the corresponding implementation. A new layer has been introduced because the safety question has changed the boundaries of the system to include the central hospital database. This change addresses the issue of how the patient is identified to the brain tumor treatment system.

The original version of the brain tumor treatment system has been included as a subsystem, as shown in Figure 15. This change minimizes the effects on the previously developed prototype and allows a quick demonstration to validate the newly proposed interface. However, it leaves some dead code in the original design, since the patient charts entering the brain tumor treatment system are now guaranteed to contain the description of a brain tumor.

In preparation for further refinements, the prototype design is cleaned up after the initial demonstration, as shown in Figure 16. Note that the specification part of the top level is not affected by the change. The new version reflects the restriction mentioned above, which has been used to tighten up the design by keeping unnecessary information out of the brain tumor treatment system (the system does not use information in the patient's chart other than the tumor's diameter).

The change affects the interface of both

```
OPERATOR brain_tumor_treatment_system
SPECIFICATION
  INPUT patient_chart: medical_history,
        treatment_switch: boolean
  OUTPUT treatment_finished: boolean
  STATES temperature: real
    INITIALLY 37.0
  DESCRIPTION
  { The brain tumor treatment system kills tumor cells
    by means of hyperthermia induced by microwaves.
  }
END
```

Figure 13. Initial top-level specification.

```
OPERATOR hospital_system
SPECIFICATION
  INPUT patient_id: string,
        treatment_switch: boolean
  OUTPUT treatment_finished: boolean,
        tumor_location: string
  DESCRIPTION
  { The hospital system provides hyperthermia treatment
    for brain tumors.
  }
END
```

Figure 14. Revised top-level specification.

the hospital database and the brain tumor treatment system. It also ripples through the lower levels of the system, resulting in the removal from the brain tumor treatment system of the *medical_history* type, the *no_tumor* exception, and the associated exception handlers.

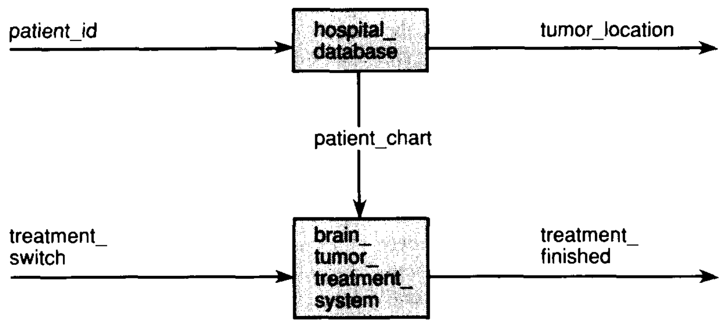
Such simplifications help keep future modifications easy to make and prevent remnants of abandoned alternatives from contaminating the design of the production version of the system. They often improve the performance of the system as well. The CAPS designers are exploring the feasibility of providing high-level editing commands for reducing the designer effort needed for such simplifications.

The effort required for evolution of a software system can be reduced through prototyping. Prototyping can stabilize the requirements for both new systems and proposed enhancements to existing systems. Feedback from demonstrations of proposed system behavior is essential for effectively validating complex requirements, such as those for large or embedded real-time systems. The customer and the developer must examine a series of changes to proposed system behavior and perceived requirements to reach a common understanding. It costs less to use a prototype than production-quality code to support this process because prototypes are simpler and easier to modify than production-quality implementations.

The effectiveness of prototyping is limited if carried out manually. A high-level language, a systematic prototyping method, and an integrated set of computer-aided prototyping tools are important for realizing the potential benefits of prototyping. Simplicity was the primary goal in designing the Computer-Aided Prototyping System, since the feasibility and efficiency of rapid prototyping depend on simplifying the tasks of the software engineer.

Prototyping is also aided by a powerful set of abstractions appropriate for a problem domain, especially if these abstractions are embodied in a set of reusable software components that can be automatically retrieved based on specifications of the desired behavior. We can save effort in the long run by building up a comprehensive library of such components for an application area if more than one software system must be developed for the same problem domain. An essential part of any practical computer-aided prototyping environment is a design database that maintains the hierarchical structure of a prototype and the relationships between mul-

```
IMPLEMENTATION GRAPH
```

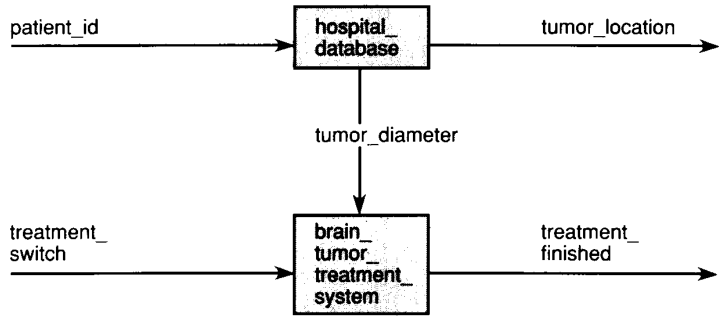


```
  DATA STREAM patient_chart: medical_history

  CONTROL CONSTRAINTS
    OPERATOR brain_tumor_treatment_system
      TRIGGERED BY ALL treatment_switch, patient_chart
END
```

Figure 15. PSDL implementation of the revised top level.

```
IMPLEMENTATION GRAPH
```



```
  DATA STREAM tumor_diameter: real

  CONTROL CONSTRAINTS
    OPERATOR brain_tumor_treatment_system
      TRIGGERED BY ALL treatment_switch, tumor_diameter
END
```

Figure 16. Cleaned-up implementation.

tiple versions developed during prototyping. An advanced design database should automatically propagate consequences of decisions made by a designer.

A group of people must work concurrently to evolve a typical software system within practical time schedules. We need better automatic methods and tools for coordinating the efforts of many people working on the same software system. Automatic means for combining components designed by different people and detecting potential conflicts would help to solve this problem. When coupled with a means for producing sets of alternative components compatible in all combinations, such tools would enable rapid tailoring of a prototype to match a user's needs.

In the future, systems will be maintained using prototype descriptions as the primary representation. To automatically generate production code, these representations will be augmented with descriptions of expected operating loads, environmental characteristics, optimization criteria, and design advice. This will preserve flexibility and enable global optimizations, which might block further evolution of the system if applied manually. □

Acknowledgment

This research was supported in part by the National Science Foundation under grant number CCR-8710737.

References

1. B. Boehm, "A Spiral Model of Software Development and Enhancement," *Computer*, Vol. 21, No. 5, May 1988, pp. 61-72.

2. Luqi and V. Berzins, "Rapidly Prototyping Real-Time Systems," *IEEE Software*, Vol. 5, No. 5, Sept. 1988, pp. 25-36.

3. R. Yeh and T. Welch, "Software Evolution: Forging a Paradigm," *Proc. Fall Joint Computer Conf.*, ACM and IEEE Computer Society, CS Press, Los Alamitos, Calif., Order No. FJ811, Oct. 1987, pp. 10-12.

4. V. Berzins, "Object-Oriented Rapid Prototyping," NPS 52-88-044, Computer Science Dept., Naval Postgraduate School, Sept. 1988.

5. Luqi and M. Ketabchi, "A Computer-Aided Prototyping System," *IEEE Software*, Vol. 5, No. 2, March 1988, pp. 66-72.

6. Luqi, "Handling Timing Constraints in Rapid Prototyping," *Proc. 22nd Ann. Hawaii Int'l Conf. System Sciences*, IEEE Computer Society, CS Press, Los Alamitos, Calif., Jan. 1989, pp. 417-424.

7. Luqi, V. Berzins, and R. Yeh, "A Prototyping Language for Real-Time Software," *IEEE Trans. Software Eng.*, Oct. 1988, pp. 1,409-1,423.

8. S. Letovsky and E. Soloway, "Delocalized Plans and Program Comprehension," *IEEE Software*, Vol. 3, No. 3, May 1986, pp. 41-49.

9. Luqi, "Knowledge Base Support for Rapid Prototyping," *IEEE Expert*, Vol. 3, No. 4, Nov. 1988, pp. 9-18.

10. S. Faulk and D. Parnas, "On Synchronization in Hard-Real-Time Systems," *Comm. ACM*, Vol. 31, No. 3, Mar. 1988, pp. 274-187.

11. C. Altizer, "Implementation of a Language Translator for a Computer-Aided Prototyping System," master's thesis, Computer Science, Naval Postgraduate School, Monterey, Calif., Dec. 1988.

12. L. Marlowe, "A Scheduler for Critical Timing Constraints," master's thesis, Computer Science, Naval Postgraduate School, Monterey, Calif., Dec. 1988.

13. V. Berzins and Luqi, *Software Engineering with Abstractions: An Integrated Approach to Software Development Using Ada*, Addison-Wesley Publishing Co., Reading, Mass., 1989.

14. H. Raum, "Design and Implementation of an Expert User Interface for the Computer-Aided Prototyping System," master's thesis, Computer Science, Naval Postgraduate School, Monterey, Calif., Dec. 1988.

Luqi is an assistant professor at the Naval Postgraduate School. She has also worked on software research and development and taught at the University of Minnesota. Her recent work includes rapid prototyping, real-time languages, design methodology, and software development tools.

Luqi received a BS in computational mathematics from Jilin University, China, and an MS and PhD in computer science from the University of Minnesota.

Address questions to the author at Naval Postgraduate School 052, Monterey, CA 93943.