Requirements for a Common Prototyping System

Chairman: Robert Balzer
Editor: Richard P. Gabriel

Common Prototyping Working Group: Frank Belz, Robert Dewar, David Fisher, John Guttag, Paul Hudak, Mitchell Wand.

Dated: Mar 2, 2010 23:36

CONTENTS

Executive Summary

Prototyping is the process of writing programs for the purpose of obtaining information prior to constructing a production version. Prototyping is used to increase the probability that the first production version will be satisfactory. It is thus a tool for reducing risk. Prototyping differs from production programming in that efficiency and completeness are often sacrificed in the interests of rapid development and ease of obtaining information.

At present there is no widely available language or system designed explicitly to support prototyping. Prototypes are built using conventional implementation languages and software development tools.

Recently, DARPA-ISTO indicated its interest in a research and development program leading to a widely applicable and well-engineered, prototyping facility, preferably within four years. Though the primary users of this facility are to come from the Ada community, the facility is to support interoperability between prototypes and multiple implementation environments, particularly Common Lisp. This draft report identifies some of the main technical requirements of such a facility. This draft report identifies some of the main technical requirements for such a facility, and is intended to provide a basis for involving broad segments of the software community in the process of refining these preliminary requirements.

None of the individual requirements seems beyond the range of current technology. The challenge lies in bringing together disparate technologies and integrating them into a single coherent well-engineered system. The major difficulties are as follows:

- Wide Applicability: This forces the prototyping language to have an extremely broad scope including parallel, distributed, real-time, and knowledge-based applications.
- Multi-Language Interoperability: This forces the runtime environment to couple with target environments beyond Ada, handling their data formats, invocation and argument passing conventions, and exceptions.
- Four Year Time Frame: This limits the amount of preparatory research and new system development that can be undertaken.
- Prototyping: The unique aspects of prototyping impose additional constraints and result in new technological requirements some of which currently have no known complete solution.

Given these difficulties, the report suggests adopting an iterative approach in which interim systems are built, each embodying different subsets of the identified requirements. An important aspect of this will be to adopt a flexible, open architecture.

Background

It is widely believed that the "waterfall" software lifecycle often used commercially and required by DoD has served to exacerbate the problems of software development and maintenance by delaying the discovery of incorrect or inappropriate specifications and requirements until the testing phase that follows implementation. It has been estimated that the cost of correcting such an error or bad decision increases by a factor of 10 for each phase of the waterfall lifecycle through which it passes undetected.

One of the groups that recently considered this issue was the Defense Science Board Task Force on Military Software 1987. That Task Force observed that writing a specification is like writing a program without the benefit of a computer to help check your work. Current software systems are so complex that even the most diligent thought process cannot envision them precisely and thoroughly enough that a correct and appropriate specification can be produced. Two common errors are to specify over-rich functionality and to poorly envision user interfaces. The trade-offs that need to be made during detailed design and implementation cannot be foreseen by even the best human specifier. [Brooks 1987]

The Task Force therefore proposed the use of *operational prototypes* to refine and validate specifications through trial use and feedback, and an increased role for it within the lifecycle of military software.

However, despite the apparent importance of prototyping and despite the fact that this discrepancy had been noted in previous studies [Brooks 1987] [Packard 1986], no strong prototyping tradition exists within either the commercial or the military software communities. In fact, the importance of prototyping contrasts sharply with the existing fragmented support for prototyping. No widely available languages or environments exist specifically to support prototyping. Most prototypes are written in a conventional language, usually the expected implementation language for the final product. There are few tools for extracting information from prototypes or for integrating prototypes with other prototypes and existing systems. Finally, few tools exist for directly reusing existing components (either prototype or target language) or for encapsulating them to adapt them for reuse. In short, with few exceptions prototyping relies on conventional implementation languages and software development tools. This limits both the use of and benefits from prototyping.

Recognizing this state of affairs, DARPA-ISTO indicated its interest in a research and development program leading to a widely applicable, widely accessible, well-engineered, prototyping facility supporting the development, evolution, and use of prototypes for software products ultimately to be developed in the Ada *target language*.

They invited the authors of this report to work together and with the software community to:

1. Determine whether technology exists or could be developed to build such a common prototyping system within four years and, if so,

2. Identify requirements that such an integrated language and environment ought to satisfy.

The authors of this report decided that it was important to answer these questions and that it would be beneficial to involve broad segments of the software community in the process. Toward that end, the authors organized themselves into a Common Prototyping Working Group and have prepared this draft report to provide preliminary responses to these questions to solicit comments and suggestions from the research, development, and user communities.

We have set up an electronic forum for anyone interested in providing comments and suggestions on these issues or in monitoring such discussions. Send your electronic address to CPS-REQUEST@VAX.DARPA.MIL to be put on the mailing list for this forum. The forum itself is CPS@VAX.DARPA.MIL; messsages for distribution should be sent directly there. In addition, we will be hosting sessions at the Software Development Environments and Principles of Programming Languages Conferences to discuss these issues.

After a period of discussion, we expect to issue a final report. In the meantime, we see great benefit from raising these issues, involving the community, and providing access to these discussions to DARPA-ISTO and others interested in pursuing prototyping.

What is Prototyping?

Prototypes are often developed in order to determine whether some functionality may be obtained in a software product, or to determine the possibility of effectively programming some task. Some people refer to this as "exploratory programming." In such situations, prototyping is part of the discovery and invention process.

Prototypes are written in order to enhance the exploration process that goes into designing the specification of a software. In the spiral model advocated by the Defense Science Board Task Force, prototyping is used to assess risks associated with proposed solutions, algorithms, expected performance, and coding strategies. In the exploratory programming model, prototyping is used as an instrument to extend one's understanding of the problem area and solution space. We call this "prototyping to learn."

Prototypes also provide a means for a group of designers, developers, customers, and users to build a consensus and a shared context for understanding and discussing the software. Often there is a tradeoff between the operational needs for particular software and the technical capabilities of the designers and developers. Prototyping is a means of assessing the best tradeoff.

Prototyping is a tool for checking designs and for obtaining information about the nature of and requirements for the final program. Prototypes are routinely used throughout the engineering disciplines, where they are called mockups, breadboards, or simulations. Given a proposed solution to a problem, prototyping is used to answer three types of question: is it in fact a solution to the posed problem; does it have acceptable performance, production cost, and reliability; and is it a solution to the right problem?

From these considerations we are able to propose the following definition of prototyping:

> ***Prototyping*** *is the process of constructing software for the purpose of obtaining information about the adequacy and appropriateness of the designers' conception of a software product. Prototyping is usually done as a precursor to writing a* ***production*** *system, and a prototype is distinguished from a production system by typically being more quickly developed, more readily adapted, less efficient and/or complete, and more easily instrumented and monitored. Prototyping is useful to the extent that it enables information to be gained quickly and at low cost.*

For prototyping to be most effective, it must be possible to quickly build the prototype (possibly using existing software components), to easily gather data during its use, and to rapidly modify it based on what was learned during its use.

Primary Distinguishing Characteristics of Prototyping

We separate the discussion of prototyping into a discussion of the distinguishing characteristics of prototypes, the kinds of prototypes, and their lifecycle roles. In our discussions of these issues we will often contrast ***prototypes*** and ***prototyping*** with ***systems*** and ***programming***.

Prototyping is distinguished from other software development activities by two basic characteristics:

- ***Prototypes are used to obtain information*** about the behavior and performance of a running system. To provide such information, prototypes should be capable of being fully instrumented; the result of such instrumentation is that the execution of the prototype results in the generation of data that either supplies or enables the inference of the needed information. Examples of relevant instrumentation data are: what the prototype did, the relative performance of the components, the frequency of use of components and data structures, and information about control flow.
- ***Prototyping is a quick process.*** In a well-supported prototyping environment, producing a working prototype should take significantly less time and effort than it would to produce a working deliverable program with the same functionality.

The main things that can be sacrificed to achieve these capabilities are prototype efficiency and completeness. Neither can be sacrificed completely, but each can be sacrificed to the extent that the resulting models contain the necessary behavior and structure needed to answer the posed questions while retaining sufficient performance under instrumentation to gather that information. Occasionally one or the other cannot be sacrificed at all.

Kinds of Prototypes

Because prototypes are primarily used to obtain information, it is natural to categorize them on the basis of the kinds of information they provide: ***behavioral*** and ***structural***. In general, real prototypes combine elements of both. Over their lifetime this mix may change as the questions being asked of the prototype change. The flexibility implied by this changing mix is important.

Many of the requirements concerning ease of change, the ability to couple to other languages, and the first-class nature of many things are derived from the need for this degree of flexibility.

Behavioral prototypes model *what* the system being prototyped is supposed to do. These black-box models exhibit responses to stimuli. They include both functional models and user interface mockups and may employ a variety of mechanisms—such as table lookup and human intervention—to provide those responses. They are the most common type of prototype and are used to test whether the user's informally stated needs have been satisfied, i.e. for validation.

Structural prototypes model *how* the system being prototyped will accomplish its black-box behavior. These clear-box models exhibit aspects of the internal structure and organization of the system being prototyped. They are used to determine feasibility, explore design alternatives, and estimate implementation and execution costs.

Prototyping Lifecycle Roles

In the waterfall model common practice is to regard prototypes as throw-aways. There are, however, many additional benefits to treating prototyping as an iterative process that begins with system definition and continues through the development and maintenance phases. As a simplified high-level model of a system, a prototype can continue to answer many questions about a system throughout the life of that system, but only if the prototype tracks the system. Therefore, this prototyping process needs tools to help designers and programmers economically coordinate prototypes and production systems.

During the lifetime of a prototype, the concerns of different parts of the prototype may repeatedly and independently move between behavior and structure. In general, prototypes will not be able to answer every question; frequently questions about performance of the production versions of the program's components must be answered by the production versions themselves. Therefore, to be most effective in this situation, the prototyping process must be able to accommodate mixtures of production and exploratory level components. This also facilitates reuse of components, adaptation of existing systems, and prototyping components embedded in another system.

There are three basic ways of coupling the prototype into the software lifecycle:

- The prototype could be the basis for writing a specification. This is the coupling currently employed in software practice where prototypes simply provide insight for subsequent but decoupled specification and development.
- The prototype is the specification itself. As a formal model any prototype is at least a partial specification of a system. Further, since it is written in a language that is executable, it can be validated by testing. However, as a specification it is likely to be over-determined.
- The prototype is the initial implementation. Frequently, a prototype has components that are quite inefficient and so may be far removed from production versions. There is, however, the possibility of moving to a ***generative basis*** in which production implementations are derived from prototypes by applying transformations to improve efficiency. Currently, such generative

processes are almost exclusively manual, but formal processes such as transformations, annotations (compiler pragmas), and special purpose application generators are beginning to be used, and have the advantage of automatically preserving functionality.

These last two options pull prototypes in opposite directions. A prototype used as an initial implementation might be more specific and operational, while a prototype used as a specification might be more abstract and declarative. Because we believe that prototypes should be specific and operational to answer behavioral and structural questions, we have eliminated the goal of using prototypes as formal specifications. Henceforth, when we use the term specification in conjunction with prototyping, it should be understood in its informal, non-technical sense.

Relationship to Programming Languages

Almost all the experience the software community has is with *programming* languages—that is, languages used to produce operational production programs. Over the years, these languages have become higher level by incorporating abstractions whose implementations are determined by the languages' compilers, thus freeing the programmer from this level of concern. This progression will surely continue.

Many programming languages have been used for prototyping. Most prototyping experience has been pragmatic and opportunistic, making use of existing technology because no alternative existed.

A good prototyping facility must include a good prototyping language in order to be able to express both behavioral and structural prototypes. But it must also support prototyping-specific capabilities (detailed later in this report) for handling incompleteness, for supporting both early and late bindings, for permitting easy modification, for allowing both imperative and declarative styles, and for tight coupling with a target language. In addition the prototyping language may not be a good programming language because it doesn't meet production requirements for performance and/or robustness.

Ideally, no language or environment boundary should be crossed in moving from a prototype to a delivered system. But rather than doing prototyping in production languages and environments as is the practice today, we envision developing, delivering, and maintaining software products in prototyping languages and environments. This means raising our languages and environments to the prototyping level, incorporating all the capabilities described in the rest of this report, while simultaneously maintaining the current production qualities. This is one of the most promising research directions and is described more fully in the Chapter "Related Research."

Relationship to Specification Languages

The research community is developing specification languages and experimenting with integrating them into the software lifecycle. This is a young but rapidly emerging field, developing, in our view, its own distinct progression of languages. These languages differ markedly from programming languages: Rather than addressing efficiency concerns, they have abandoned efficiency

entirely in an effort to describe the desired effects and properties of computations.

We are not attempting to advance the specification language progression. An important purpose of prototyping is answering questions concerning the *dynamics* of a system, relating to both its behavior and performance. To answer those questions the prototype must be executed, and its execution must be sufficiently rapid to support gathering data. Prototyping thus emphasizes executing models, not describing them, and is therefore distinct from specification efforts.

Executable specification languages were developed to meet the needs of prototyping during specification. Experience indicates that prototyping can be done in executable specification languages, but many prototyping concerns (such as the ability to create structural prototypes, handling incompleteness, supporting both early and late bindings, allowing both imperative and declarative styles, and tightly coupling with a target language) are inadequately addressed by the predominately descriptive nature of specification languages.

Feasibility

Much research remains to be done to understand the nature of prototyping and the support it requires. One eventual outcome could be that prototypes could be evolved into deliverable systems within a single language and environment that supports both prototyping and programming.

The problem of raising our application languages and environments to the prototyping level is a long term research effort. But even if there were such a language and environment suitable for both prototyping and programming, little immediate benefit would be realized unless some *existing* application language and environment were a subset of that language. That is, because of the large investments in existing application systems, significant immediate prototyping benefits only arise from systems that support the prototyping of software that ultimately will be delivered in an existing application language.

DARPA-ISTO's short-term (four year) goal of constructing a widely applicable, widely accessible, well-engineered, prototyping facility for software ultimately to be developed in Ada satisfies this criteria. The Working Group felt that it would represent a big step forward in the availability of prototyping capabilities for the Ada community and could act as a bridge between the Ada and Common Lisp communities via its interoperability between those two languages. If properly done, such an interim system could serve as a framework in which to incorporate future advances.

This report identifies requirements that such a system ***should*** satisfy. They are summaried at the end of this chapter and detailed in Chapters 2, 3, and 4. Our criteria were that each was an important part of such a prototyping facility and that each was individually feasible, using either existing technology or advances we could envision from a focussed research effort within a few years.

Much of the technology needed to support these requirements already exists and has been incorporated and used in some running system, and the remainder appears manageable with respect to

both effort and time scale.

However, the challenge is in bringing together all these separate technologies and integrating them into a single coherent well-engineered system rather than in satisfying any particular individual requirement. In total, this challenge is daunting because of the breadth of issues covered. The main sources of difficulty arising from the DARPA-ISTO goal are as follows:

- **Wide Applicability**: This forces the prototyping language to have an extremely broad scope including parallel, distributed, real-time, and knowledge-based applications. Many languages have addressed these application areas individually, but few except our most general purpose languages has tried to cover them all.
- **Multi-Language Interoperability**: This forces the runtime environment to couple with target environments beyond Ada, handling their data formats, invocation and argument passing conventions, and exceptions.
- **Four Year Time Frame**: This limits the amount of preparatory research and new system development that can be undertaken.
- **Prototyping**: The unique aspects of prototyping impose additional constraints and result in new technological requirements some of which currently have no known complete solution.

To reduce some of these difficulties, the Working Group adopted an open-architecture approach in which we limited the scope of many requirements or merely noted their desirability without requiring them, with the expectation that over time more advanced versions of those capabilities could be incorporated.

Thus we envision the growth of the prototyping facility to itself follow the prototyping paradigm both during the definitional phase and during later enhancements.

Nevertheless, because we did not take overall cost and effort into account in determining requirements—only their individual technical feasibility—the aggregate scope and magnitude of the engineering is quite aggressive and unlikely to be feasible using a waterfall approach attempting to embody all these requirements.

However, by adopting the iterative prototyping approach, we believe many useful interim systems could be built, embodying different subsets of the identified requirements, which satisfy the general goal of providing immediate significant benefits by prototyping software ultimately to be delivered in Ada. Choosing appropriate subsets involves both technical and policy considerations. For instance, as the understanding of prototyping improves, these requirements will evolve and be refined. We are also concerned that the aggregate scope and magnitude of the engineering needed to satisfy too many requirements in a single system may induce too conservative a design, omitting the technical advances we feel are crucial for success.

An important task not carried out by the Working Group was to carefully evaluate existing languages and environments with respect to the set of requirements for prototyping. Although it is felt that it is unlikely that any one existing language or environment meets all the stated requirements in its current form, it is clear that a number of languages and environments, including for

example Lisp, Prolog, SETL, APL, SmallTalk, ML, and others meet many of the requirements. A careful analysis in each case will be helpful both in the task of better understanding the requirements as they stand, and in determining whether one or more of these languages or environments might serve as an appropriate design base for a prototyping facility.

Objectives

We are concerned both with the language issues involved in a prototyping system and with support for the process of developing and using prototypes. We believe the following are the key objectives to be addressed by prototyping:

- Ability to provide information: The main use of prototyping is to extract information from executing a prototype or from the process of creating it. This information includes the functionality of the prototype, its performance, its structure, and the design decisions it is based on. This information must both be comprehensible and accessible. Tools and facilities for gathering, analyzing, and presenting both static and dynamic information are required.
- Ability to quickly and easily produce a prototype: Prototyping will only be used if the benefit to cost ratio is favorable. An important cost factor is the ability to use existing modules and programs as components of the prototype; another is the expressiveness of the prototyping language.
- Low Inertia: Obtaining information from a prototype sometimes requires further changes to the prototype. The language and environment must facilitate such changes by minimizing the effort required to make them and perform experiments.
- Mixed Behavioral and Structural Use: To facilitate the use of prototypes throughout the iterative exploratory and definitional phases, the language must support mixtures of behavioral and structural prototypes and provide mechanisms to enable this mixture to vary over time.

To this list of prototyping objectives, we add the following growth objective:

- Anticipatory: To facilitate incorporation of future improvements in prototyping technology, a prototyping facility must be engineered to be adaptable.

Summary of Requirements and Desired Features

We have determined that the focus should be on ***prototyping systems***, where a prototyping system $\mathcal{PS}$ is a prototyping language $\mathcal{PL}$, along with a prototyping environment $\mathcal{PE}$.

This section summarizes our key requirements and remarks on desired features. In the following

chapters, we describe the reasoning that support these requirements and remarks, and provide the complete set of such requirements and remarks on desired features.

We ***require*** something when we believe that leaving it out will seriously or fatally affect the success of $\mathcal{PS}$ as a prototyping system. We remark that something is a ***desired feature*** when we believe that it will improve $\mathcal{PS}$ as a prototyping system. We will always use the word ***shall*** to mean that the statement is a requirement.

A requirement statement shall have the following format:

Requirement: ***[Requirement Format]*** All requirement statements shall have this form.

A remark regarding a desired feature will have the following format:

Remark: ***[Desired Features]*** All remarks regarding desired features should be pithy and interesting.

A System

We address in this section those issues that are not wholely contained in either $\mathcal{PL}$ or $\mathcal{PE}$.

Architecture

The success of $\mathcal{PS}$ will depend as much on the degree of engineering put into it as on the scientific advances it represents. The primary challenge for $\mathcal{PS}$ is in integrating existing technology into a coherent well-engineered system.

But $\mathcal{PS}$ is an example of a system which is far too complex to be fully envisioned in advance. Furthermore, many of the capabilities it should ultimately contain can only be partially supported initially. It must therefore support and facilitate growth.

We note that the most successful systems in terms of evolutionary growth are those that employ an ***open*** architecture, facilitate user additions and modifications, and provide some facility for fortifying, integrating, and incorporating user prototypes into the evolving system. An open architecture is one with visible, coherent, open-ended internal and external interfaces. Examples of such systems include X-Windows, GNU Emacs, Lisp, Smalltalk, Unix, and Tenex.

$\mathcal{PS}$ shall employ an open architecture and for its implementors to accompany it with a facility for fortifying, integrating, and incorporating user prototypes into the evolving $\mathcal{PS}$.

Multi-Language Interoperability

Prototyping does not exist in a vacuum: A typical organization engaged in prototyping has already produced a volume of software in the target language or languages. It has a number of modules and other pieces of software that will form parts of future production software. To minimize the effort to construct prototypes and to maximize their accuracy, it is desirable to reuse these components.

Such reuse should neither be restricted to source text nor to single-language systems. That is, it should be possible to separately compile parts of a large system and use those parts in the prototype. The process of putting together fragments of existing code in a prototype is called ***composition***, and this functionality should be expressible in the prototyping language $\mathcal{PL}$ and supported by the prototyping environment $\mathcal{PE}$.

This support shall enable components written in $\mathcal{PL}$ to invoke components written in the implementation languages (Call-Out) and vice versa (Call-In), and shall support all forms of invocation supported by those implementation language as well as all value and/or exception passing mechanisms.

$\mathcal{PS}$ shall provide this coupling to both Ada and Common Lisp, and it is desirable to include other implementation languages as well. Because this facility is intended only to support prototyping composition, such coupling is required only between $\mathcal{PS}$ and such implementation languages, not between the eventual Ada-based software and these languages.

The ability to compose existing code as part of the prototype addresses the need to quickly and easily produce a prototype. It provides at least one means to mix behavioral and structural components in a prototype. Further, it provides a means for evolving a prototype into its production counterpart by substituting target language modules for prototyped modules.

We believe that $\mathcal{PL}$ should allow prototype code to be written at appropriate levels of abstraction and should enable prototypers to reuse existing components, such as Ada components. It should be possible to express incomplete prototypes in $\mathcal{PL}$ and to test and instrument them in $\mathcal{PE}$.

$\mathcal{PE}$ should integrate design, coding, testing, debugging, and information extraction.

Persistent Object Management

If our hopes about the software lifecycle are fulfilled, prototypes will continue to exist beyond their use in the specification phase. In this case, all documentation, specification, and informal data that is derived or produced during the early phases of prototyping will need to continue to exist for long periods of time. In addition, the use of prototyping throughout the software lifecycle will be facilitated if there are tools that assist in maintaining the linkages between changes in the production implementation and the prototype.

In our view, persistent object support is the means of achieving these benefits in a safe and controllable manner. We feel that prototyping will be more broadly accepted if prototyping tools facilitate the long-term nature of the software lifecycle.

Persistent object management is the management of structured values or objects that persist beyond the execution of the programs that created them. In a prototyping system there are three classes of objects that need to be persistent. One is the class of objects that prototypes create and which are subsequently used by prototypes or other systems.

The second class of objects are the prototypes themselves along with the database of information maintained by $\mathcal{PE}$. Prototyping involves the rapid evolution of programs and their components using a variety of tools provided by the prototyping system. This evolution is possible only

if there is a way to preserve the prototypes outside the individual tools that create, compose, analyze, translate, execute, or otherwise manipulate them. These tools also will maintain a set of information discovered during their execution. This information should persist beyond any prototyping session.

Among this second class are objects required by $\mathcal{PS}$ for configuration and version management.

The third class of objects are the definitions of the types and abstractions used or produced by a prototype. These should be persistent ***first-class*** objects because the validity of the data instances depend on them, and they are themselves components subject to reuse and sharing. In $\mathcal{PL}$, it should be possible to build abstractions of any kind of entity. A first-class object is one that can be passed to or returned from a procedure, stored in a data structure, or made persistent. As these definitions evolve, facilities must exist for managing their existing persistent instances.

Thus persistent object management involves a combination of mechanisms for type integrity, object identity, version management, and configuration control.

Binding Management

Binding is the process of associating values with names. In traditional programming languages, a value is bound to a name for some period during the evaluation of a program. There are several dimensions to the binding process: Binding can occur at different times, a particular binding can occur all at once or in stages, and bindings can be mutable or immutable.

From a prototyping viewpoint, the distinctions between the various binding times not only influence performance but also the ease and rapidity of implementation and modification of programs. The more requirements imposed by the language for early binding the more difficult to describe prototypes and the smaller the set of applications that can be implemented with ease.

Binding can occur all at once or in stages: Certain properties of a value can be bound at the point of type definition, others when subtypes are defined, still more when the type of a variable is declared, and the remainder when the value is specified. The ability to specify the various properties of values separately in a succession of states permits separation of issues and increases the ease and likelihood of correctness of a modification. We believe the ability to specify bindings in many stages is even more important to prototyping languages than to programming languages.

Bindings of value to names can be mutable or immutable. Immutable bindings make it easier to analyze and reason about programs, characteristics that are very important to prototyping. On the other hand, a prototype with only immutable bindings may be harder to write and modify. A prototyping language should provide both.

This flexibility of binding management has important systemic implications on the nature of $\mathcal{PS}$. It means that functionality that has usually been statically allocated to just one portion of the system (such as the compiler, loader, or runtime system), should now be dynamically shared between them on the basis of the particular bindings chosen by the prototyper. Consider for example type safety and abstraction security. Both are required within $\mathcal{PS}$. However, it is possible that neither could be completely handled by a $\mathcal{PL}$ compiler as they can in a compile-time type

inferencing language if types (and abstractions) are first-class objects that can be dynamically bound to names at runtime. Hence, some runtime checking might be needed to ensure type safety and abstraction security. On the other hand, we simultaneously demand that the compiler should do as much safety and security checking as possible.

A Language

The major requirements on $\mathcal{PL}$ arise from its being a prototyping language, from the need to support dual implementation languages, and from the need for it to be widely applicable.

Prototyping Language

It must be possible to construct both behavioral and structural prototypes in $\mathcal{PL}$. Therefore, $\mathcal{PL}$ must model both the functionality and the organizational structures and interfaces of the target languages. Furthermore, it must be possible to mix behavioral and structural components and have the mixture vary over time.

Since Ada is imperative, we believe that $\mathcal{PL}$ should be imperative but with declarative or functional constructs. The two most prominent examples of mixing the styles are ML and FX. We make no requirements regarding non-strict (lazy) versus strict semantics. However, if a non-strict semantics is chosen for any component of $\mathcal{PL}$, we require that they be reconciled with the imperative component of the language.

To retain its operationality so that prototypes can be executed to extract information, no component of $\mathcal{PL}$ should be non-effective (except those involving human intervention). However, $\mathcal{PL}$ should consistently incorporate the highest level, most expressive constructs that remain effective. Mathematics represents an idealized extreme in which effectiveness is not required, but many mathematical notions are effective, and they ought to be incorporated and broadly used. To the extent that expressiveness conflicts with target language integration, we felt that expressiveness should have the greater weight, i.e. the target language ought not overly influence the design of $\mathcal{PL}$.

$\mathcal{PL}$ should also allow non-determinism (an arbitrary choice over some spectrum of possibilities) to be explicitly expressed. Furthermore, it should be designed so that it will have acceptable operational performance when implemented using existing technology—a large prototype should run no worse than an order of magnitude slower than the same program in Ada.

In $\mathcal{PL}$, it should be possible to build abstractions of many kinds of entities, such as data values, expressions, commands, declarations, and L-values. These abstractions and all data values should be first class.

There will be a single abstract syntax and a single concrete syntax for $\mathcal{PL}$. The standard concrete syntax shall be used in all publications to reduce the difficulties that are associated with multiple concrete syntaxes for a single language.

Wide Applicability

To maximize the applicability of $\mathcal{PL}$ for the wide variety of military systems that use Ada, $\mathcal{PL}$ should incorporate constructs for parallel, distributed, real-time, and knowledge-based applications. While this requirement pushes the state of the art in none of these areas individually, integrating them together into a single coherent language provides one of the most difficult challenges for the designers of $\mathcal{PL}$.

Included under wide applicability is the ability to construct system tools and components, in particular portions of $\mathcal{PS}$ itself. This implies the capability in $\mathcal{PL}$ to treat types and abstractions interpretively as runtime bindings so that $\mathcal{PS}$ tools (such as debuggers and information extractors) can operate on the types and abstractions contained in user prototypes.

We suggest that major portions of $\mathcal{PS}$ be built in $\mathcal{PL}$. It is not required in order to avoid bootstrapping problems and unrealistic efficiency requirements on $\mathcal{PL}$.

An Environment

As with $\mathcal{PL}$, $\mathcal{PE}$ has both a component that it shares with programming systems, and an additional prototyping-specific component that is unique.

Programming Environment

A ***programming environment*** is a set of tools for helping a programmer get a program written and running. Often, the more advanced the programming environment the more open and scalable it is, and the more integrated are its tools. By ***integration*** we mean that the tools are interconnected in such a way that moving from tool to tool during a programming session is easy and that tools operate on common representations.

There are two distinct parts to the generic programming component of $\mathcal{PE}$. The first is the runtime system which supports the semantics of $\mathcal{PL}$. It performs memory management, ensures runtime type consistency, provides linkage among procedures and functions written in supported languages, and supports instrumentation and monitoring. We will always refer to this aspect of $\mathcal{PS}$ as the ***runtime system***. The second is the development environment, which provides programming tools like editors, debuggers, and browsers, along with the user interface, the dynamic loader, the window system, and presentation tools for instrumentation and monitoring.

The prototyping environment, $\mathcal{PE}$, should encompass the most up-to-date programming environment techniques. It should be window-based, dynamic, incremental, and well integrated. Several existing systems, including the Lisp Machine, Smalltalk, APL, Cedar, and Rational Computer's Ada environment illustrate the current state of the art.

In addition to the capabilities found in these systems, $\mathcal{PE}$ should be built on a persistent object base, should support multiple versions of prototypes (both chronological and variants), and should support multiple users.

Prototyping Environment

Extracting information is one of the two activities that distinguishes prototyping from programming. This information is used to show what, how, and how well the prototype is doing. It can also be used to determine whether the prototype is functioning as intended and, if not, why not. That is, this information is used both for testing and debugging. Supporting information extraction requires mechanisms to do the following:

- building test harnesses needed for tests
- constructing scenarios to drive tests
- writing test evaluators
- reinitializing the state of an executing prototype in preparation for a test

It should be possible to instrument the prototype and its data in such a way that the instrumentation has access to the entire state of the computation including performance information.

It should be possible to satisfy the following additional higher-level instrumentation requirements:

- ***Specification:*** It should be possible to monitor changes to data satisfying a $\mathcal{PL}$ predicate.
- ***Composition:*** The user should be able to define compound events and to instrument those compound events. The presentation of instrumentation should be in those terms specified by the user.
- ***Low Inertia:*** The user should be able to quickly and easily turn instrumentation on or off and to change what is instrumented.

It should be easy for the user to run ***incomplete*** prototypes and to specify and change how $\mathcal{PE}$ handles unsupplied or incomplete components. A prototype is incomplete if not all procedures, functions, or types are defined or if they are partially defined. The following are examples of how $\mathcal{PE}$ might handle unsupplied or incomplete components:

- Invoke a condition handler
- Query the user
- Entering a break loop

This list is not intended to be exhaustive.

$\mathcal{PE}$ will need to enable the prototyper to modify running prototypes by altering data structure and procedure definitions.

$\mathcal{PE}$ should support the rapid development of sophisticated user interfaces employing windowing systems, pointing devices, high resolution graphical screens, and laser output devices. Because there is a trend towards designing the user interface before designing the rest of the system, user interface support is extremely important. Requirements engineering often entails providing a user

interface that can be tried out by a sample of users. Currently there are available user interface toolkits that are designed to help programmers design user interfaces by providing commonly used components along with a user interface to customize those components. We expect that $\mathcal{PE}$ will need to supply such a user interface building kit.

Research Issues

The development of $\mathcal{PS}$ raises certain research issues related to prototyping which should be investigated in a broader context and on a longer term basis.

- ***Suitability of $\mathcal{PL}$ as a production programming language***: As described above, a good prototyping language necessarily includes a good programming language. But whether a prototyping language can/should be used as a production programming language is an important open question.
- ***Mechanized Optimization and Translation***: Alternatively, efficient production programs in the target language might be mechanically derived from the prototypes. But much work remains to make this approach practical.
- ***Relationship between Specification and Prototypes***: It is tempting to hope that given a suitable prototyping language one could avoid separate specifications by treating the prototype as the specification. It appears, however, that specifications and prototypes are complementary. Understanding the nature of these complementary roles and determining what information needs to be contained in the specification of $\mathcal{PL}$ modules to facilitate their reuse and composition is important for the continued growth of prototyping.
- ***Indexing Reusable Libraries***: Libraries are not useful without indices. No semantics-based indexing and retrieval mechanism exists. Such a facility is sorely needed to support reuse.
- ***Parallel and Distributed Systems***: Though we require a machine model that is both parallel and distributed, we realize that it is premature to insist that the first $\mathcal{PS}$ support it. This type of machine will play a more major role in the future. Particularly important will be research into new parallel and distributed programming languages.
- ***Programming Languages and Environments***: In addition to their prototyping specific aspects, $\mathcal{PL}$ and $\mathcal{PE}$ both are envisioned to have extensive generic portions that rely heavily on existing programming languages and programming environments. Continued growth of these generic portions of $\mathcal{PL}$ and $\mathcal{PE}$ rely upon continued research in these areas.

References

[**Brooks 1987**] Brooks, Frederick P., *Report of the Defense Science Board on Military Software*, September, 1987.

[**Packard 1986**] Packard, David, *National Security Planning and Budgeting*, A Report to the President by the President's Blue Ribbon Commission on Defense Management, June 1986, May, 1986.