Foundations for the Arcadia Environment Architecture

Richard N. Taylor, Frank C. Belz†, Lori A. Clarke*, Leon Osterweil**,
Richard W. Selby, Jack C. Wileden*, Alexander L. Wolf‡, Michal Young

Department of Information and Computer Science
University of California, Irvine[1]

*Department of Computer and Information Science
University of Massachusetts, Amherst[2]

**Department of Computer Science
University of Colorado, Boulder[3]

†TRW[4]
Redondo Beach, California

‡AT&T Bell Laboratories,
600 Mountain Avenue
Murray Hill, NJ

Abstract

Early software environments have supported a narrow range of activities (*programming* environments) or else been restricted to a single "hard-wired" software development process. The Arcadia research project is investigating the construction of software environments that are tightly integrated, yet flexible and extensible enough to support experimentation with alternative software processes and tools. This has led us to view an environment as being composed of two distinct, cooperating parts. One is the *variant* part, consisting of process programs and the tools and objects used and defined by those programs. The other is the fixed part, or *infrastructure*, supporting creation, execution, and change to the constituents of the variant part. The major components of the infrastructure are a process programming language and interpreter, object management system, and user interface management system. Process programming facilitates precise definition and automated support of software development and maintenance activities. The object management system provides typing, relationships, persistence, distribution and concurrency control capabilities. The user interface management system mediates communication between human users and executing processes, providing pleasant and uniform access to all facilities of the environment. Research in each of these areas and the interaction among them is described.

1 INTRODUCTION

The purpose of a software environment is to support users in their software development and maintenance activities. Past attempts to do this have indicated the vast scope and complexity of this problem. The Arcadia project is a consortium research effort aimed at addressing an unusually broad range of software environment issues. In particular, the Arcadia project is concerned with simultaneously investigating and developing prototype demonstrations of:

- environment architectures for organizing large collections of tools and facilitating their interactions with users as well as with each other,
- tools to facilitate the testing and analysis of concurrent and sequential software, and
- frameworks for environment and tool evaluation.

This paper describes the research rationale and approach being taken by Arcadia researchers in investigating environment architecture issues. Although details concerning the tool suite and the evaluation framework are outside the scope of this paper, attempting to assemble these components into a coherent environment will provide a non-trivial test case for experimentally evaluating this architecture.

The remainder of this section presents a high-level overview of our proposed environment architecture. The major components of the architecture are described. Each of these represents a major research subarea being investigated as part of this project. The ensuing sections describe the major goals and rationale for each of these subareas. Although each subarea is an interesting research project in its own right, the most challenging questions are often raised by the interactions among subareas. Indeed, it is the importance and complexity of these interactions that requires research in the subareas be pursued cooperatively. One of the novel features of the Arcadia project is that it is synergistically exploring many of these issues.

[1]This work was supported in part by the National Science Foundation under grant CCR-8704311, with cooperation from the Defense Advanced Research Projects Agency (ARPA Order 6108, Program Code 7T10), by the National Science Foundation under grants CCR-8451421 and CCR-8521398, Hughes Aircraft (PYI program), and TRW (PYI program).

[2]This work was supported in part by the National Science Foundation under grant CCR-87-04478, with cooperation from the Defense Advanced Research Projects Agency (ARPA Order 6104), and by the National Science Foundation under grants DCR-8404217 and DCR-8408143.

[3]This work was supported in part by the National Science Foundation under grant CCR-8705162, with cooperation from the Defense Advanced Research Projects Agency (ARPA Order 6100, Program Code 7E20), by the National Science Foundation under grant DCR-0403341, and by the U.S. Department of Energy under grant DE-FG02-84ER13283.

[4]This work was sponsored in part by TRW and by the Defense Advanced Research Projects Agency/Information Systems Technology Office, ARPA Order 9152, issued by the Space and Naval Warfare Systems Command under contract N00039-88-C-0047.

Recommended by: *Stowe Boyd*

1.1 Arcadia Environment Architecture Goals

Osterweil observed [40] that a software environment must be broad in scope, highly flexible and extensible, and very well integrated. Subsequent research has underscored how essential those characteristics are in a software environment, but has also indicated that there are some fundamental tensions among them. Most strikingly, it seems that a well-integrated environment is easiest to achieve if the environment is limited in scope and static in its content and organization. Conversely, broad and dynamic environments are typically loosely coupled and poorly integrated. Unfortunately, poorly integrated environments impose excessive burdens upon users, and small static environments are quickly outgrown. Thus, it is necessary to conduct research on environment flexibility/extensibility techniques, on environment integration techniques, and on understanding the tensions between these two often-conflicting goals. This is precisely the intent of the Arcadia research project, which is developing a prototype environment architecture directed toward maximizing both flexibility/extensibility and integration and toward understanding the tradeoffs between them.

The requirement that an environment be flexible and extensible springs from a variety of sources. First, users are different and perceive their needs differently. In addition, projects are different and have different support requirements. Further, as projects progress, their needs change and workers' perceptions of these needs also change. These factors all dictate that environments must be flexible enough to change the nature of their support for users as painlessly as possible. In addition, as new tools and technologies appear, a software environment must readily incorporate them. Otherwise the environment will become increasingly inefficient and obsolescent.

Arcadia research on environment flexibility/extensibility focuses on supporting the notion of process programming [39,41]. The basic idea is that process programs, written in a process programming language (PPL), will describe the diverse software processes that users want to employ in developing and maintaining software. The tools available to support software activities will be operators or functions in this language and the operands will be the various objects created by tools or users. Because of the complexity of software processes, a PPL will need to have at least the power of a general-purpose programming language, extended to satisfy the requirements of process programming.

With this environment model, flexibility is obtained by supporting alterations to process programs. Extensibility is achieved by writing new process programs or by modifying existing process programs to incorporate new tools, subprocesses, types or objects. For example, a new tool is incorporated into the environment by writing a new process program, or modifying an existing process program, that explicitly indicates the way this tool will interact with other tools in the environment and the types of objects this tool will use and create.

To assure that the flexibility/extensibility gained through process programming does not come at the expense of integration, Arcadia researchers are also investigating integration techniques. The requirement for tight integration has a variety of manifestations. Users should interact with the environment in a uniform way, instead of accommodating themselves to each tool's idiosyncratic interface. In addition, environment tools should share information among themselves, assuring that users are not pestered to supply the same information multiple times nor needlessly paying for recomputation of available information. Environment components should be shared whenever possible as well, to keep the size of the environment down and to prevent performance penalties due to excessive paging and thrashing. Integration issues such as these seem to divide neatly into internal and external integration issues.

In the Arcadia project, internal integration research focuses on the investigation of environment object management issues. In recent years it has become increasingly clear that environments require powerful object management mechanisms. The earliest software environments were little more than intelligent editors [28], whose object management needs were modest, in keeping with their modest functionality and scope. As environment power and scope have grown, however, effective object management has become a daunting task. Object managers must be capable of effectively storing and retrieving software objects for a broad spectrum of tools. Software objects may be internal data structures, such as parse trees, symbol tables, and abstract syntax graphs, or external products, such as source code, test plans, and designs. Software objects may vary in size from a single byte to millions of bytes, may persist in the object store for seconds or years, may be manipulated by transactions that are brief or last for months, and may be self contained or intricately interrelated to other objects. Further, if the environment is to be flexible and extensible, the object manager must be capable of reacting effectively to a wide variety of changes.

External integration research in the Arcadia project focuses on investigation of user interface management issues. The user's interactions with all of the tools and facilities of an environment must be as uniform and comfortable as possible. Because software tools are so varied, environments must support the effective use of a gamut of user interface technologies, ranging from simple textual interfaces to direct manipulation of complex graphical displays. Moreover, independent evolution of both the tool base and user interface technology must be accommodated. In addition, environments must support multiple users acting in various roles. An environment user interface management system must also be able to project multiple views of a software object and to maintain consistency among these views.

Several other research projects are exploring approaches for providing strong integration mechanisms, external or internal, or powerful flexibility and extensibility mechanisms. For the most part these issues are being investigated in isolation, and the results often appear to be inconsistent with each other. For example, there are object management systems (e.g., most classical databases) that seem to offer acceptable support for modest, relatively inflexible environments, but their support is inadequate for more complex, flexible, and extensible environments [4]. Similarly, some excellent user interfaces have been developed, but often these are closely tailored to fixed tool configurations. Environments that are more flexible seem to undermine the power of such user interfaces.

Arcadia researchers are simultaneously investigating environment integration and environment flexibility/extensibility to study the tensions and interactions between them. To experimentally evaluate our approach, we are developing a prototype environment. The architecture of this environment is designed to foster research in each of the major subareas and accelerate understanding of the interplay among them. The Arcadia project will yield research insights in each of the major subareas as well as an environment prototype that effectively synthesizes and demonstrates these research findings.

1.2 Architecture Overview

The high-level architecture of the Arcadia prototype environment, called Arcadia-1, is depicted in Figure 1. The Process

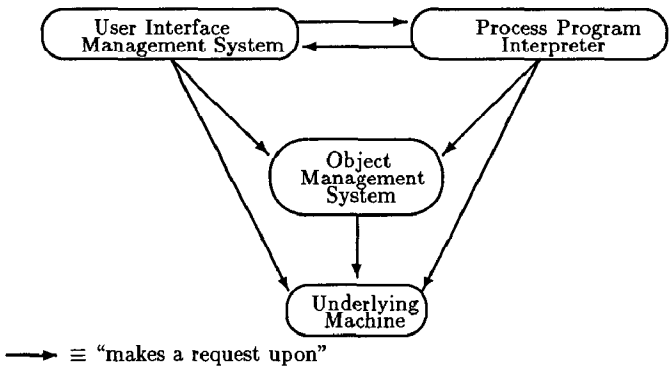

Figure 1: Invariant Components of the Arcadia-1 Environment

Program Interpreter (PPI) component is responsible for carrying out the instructions of the process programs. This component communicates with users through the User Interface Management System (UIMS) and accesses software objects via the Object Manager. All three of these components must interact with an underlying virtual machine for basic operating system support. In our early work we are assuming this machine to be the Berkeley UNIX [1] version 4.3 operating system running on a network of Sun workstations.

The architecture is designed so that the capabilities of each important subarea can be defined separately, through carefully specified interfaces that provide the appropriate functionality while hiding implementation details. This separation of concerns, captured by the system modularity, facilitates orderly evolution based on continuing research activity in each subarea. Clearly each key component must still interact with other key architectural components in supporting the overall architecture. For example, research into using process programming to achieve flexibility and extensibility is constrained by and coordinated with parallel research into using object managers and user interface management systems to achieve tight internal and external integration.

Figure 1 depicts the infrastructure, or *fixed* part, of an environment and thus does not show any of the process programs, tools, and software objects that populate an environment. It is anticipated that these components will change frequently, so they are referred to as the *variant* part of the architecture. The Arcadia architecture encourages change to the variant part, and discourages change to the fixed part. Of course, there are bound to be circumstances under which the fixed part will require change as well, but presumably such changes should occur infrequently. One of the contributions of the Arcadia project is the recognition of the need to separate the fixed and variant parts of an architecture. We intend to experimentally explore where the boundary between these two parts lies and the ramifications of this distinction.

2 SOFTWARE PROCESS PROGRAMS

2.1 Process Program Requirements and Related Work

Software development and maintenance organizations carry out their jobs by following carefully thought out processes. Difficulties arise because specifying and modifying these processes is a far harder task than expected and is therefore not done effectively. Modelling software processes is a useful way to approach these difficulties, and a number of approaches to modelling software processes have been proposed [47,73,20,57,6,44].

As the previous discussion on environment flexibility and extensibility reveals, environments must not only support user processes but they must also support change to those processes. Different users employ different processes. At different times a user may wish to change processes, perhaps because a better way of doing work suggests itself, because the overall project process has changed, or because the user may wish to begin using a new tool. In all of these cases, the needed changes seem to be accurately viewed as changes to the model describing the user's process. In effect we are suggesting that environment flexibility and extensibility are to be achieved by carrying out maintenance changes to process programs.

Process programming differs from earlier work in that it hypothesizes the need for a full programming language. At a minimum, process programming languages need to provide powerful typing mechanisms and a variety of powerful flow of control mechanisms such as concurrency, exception handling, and complex looping constructs. An important part of our research involves determining the necessary characteristics of such languages and defining at least one such language and developing an interpreter for it.

A process program indicates how the various software tools and objects would be coordinated to support a process. It would also be used to indicate to a process program interpreter which operators are to be carried out by humans, rather than tools or hardware. These operations would appear as subprocesses, which would not be further elaborated, but would be bound to human execution agents. The Arcadia research project is providing a framework for investigation of this premise.

Operating system control languages have attempted to support the specification of software processes, and we believe that these are primitive antecedents of process programming. In this context, system utilities are process programming language operators and the files managed by the operating system's file system are their operands. Command files or scripts are primitive process programs, using the operating system command language as a process programming language. These languages, however, do not usually enable users to assign types to data objects, specify complex control sequences, access operands smaller than files, or invoke operators smaller than executable programs and tools.

The lack of a rich type system in these languages is a serious deficiency. Key software objects, such as parse trees, requirements specifications, testcases, and bug reports, should be viewed as instances of types. These types seem readily definable using typing mechanisms in modern programming languages. Object typing is a powerful vehicle for organizing the objects managed in a software project, and for defining and organizing the operators — both human and machine executable — to be employed by the project.[2] Object typing is discussed further in Section 3.

Another deficiency is the lack of sufficiently powerful flow of control capabilities. Many operating system command languages incorporate some flow of control operators, but these are usually quite primitive, often consisting only of basic looping and alternation constructs (*csh* is an exception here). These restrictions are unacceptably constraining in trying to program real software processes, as some early work has shown. Perhaps the most no-

[1] UNIX is a registered trademark of AT&T

[2] In either case, the semantics of operations can be formally defined using, for example, pre- and post- conditions.

table of this early work was the diagrammatic software lifecycle modelling, such as the "Waterfall Model" [53]. In this work, principal software processes were represented by boxes, and the only flow of control that was shown was that which could be represented using arrows. These models were attacked as being incomplete and naive. Their weak flow of control capabilities made it impossible, for example, to depict concurrent activities, which are essential to any real, large-scale software activity.

Another deficiency in operating system command languages is their inability to show relations such as data flow and process hierarchy. Software process models have attempted to represent key relations among a variety of types of software objects. For these models notations have been developed in which, for example, data objects are differentiated from process objects by using different shaped boxes. Different relations are usually indicated by differently shaped lines and arrowheads and by different colors. Some examples of advanced techniques and notations for diagrammatic representations of processes are ETVX boxes [48], SADT diagrams [51] and Software Development Graphs [5]. These notations and representations are limited because there are so many valid relations among the software objects in a real software project, and different users may at different times wish to study various combinations of them. Creating a single diagram showing all of these relations is hardly a solution, as such a diagram is so complicated as to confound all understanding. Creating a single internal representation capturing all the relations among the objects and subprocesses of a software process, and then using tools to draw specific diagrams ("views") upon request, is a more promising approach. The PMDB project moved in this direction by proposing a model of the life-cycle process [44] in terms of its objects and relationship types and by performing a prototyping exercise exploring views based on user roles [43]. It did not, however, model explicitly the activities that produce such objects and relationships.

Some earlier software environments can be viewed as providing process programming support for limited software development activities. For instance, systems such as *PAISLey* [76], *RSL/REVS* [3], *SARA* [21], Data Flow Diagram Designs [70], Jackson System Development [12], and *USE* [71], support the creation of certain fixed types of specifications and designs. Code development is supported by programming environments, such as *Interlisp* [65], *Arcturus* [58], and *Cedar* [64], which incorporate tools for editing, parsing, debugging, and documentation. Similarly, intelligent editors such as the *Cornell Program Synthesizer* [63], *Integral-C* [52], *Gandalf* [24], and *Mentor* [19] effectively integrate user activities around a parsed representation of code. As long as the user's activities fall within the domain of these environments, they provide strong support. If the user seeks to model software products and processes outside this domain, support from these environments falters.

Support for limited, pre-determined processes has taught us a great deal about those processes but has also demonstrated how quickly users want to stray beyond those boundaries. Products and processes must be expected to vary from user to user, from location to location, and from time to time. Thus no fixed predetermined process will be adequate for a wide range of software activities, or even for a single software activity over a period of time. Thus it seems imperative that the full power of a general purpose programming language be available to users attempting to express the full range of processes to be supported.

We believe that progress towards understanding what constitutes adequate products and effective processes can only follow from experimentation with alternatives. The best way to facilitate such experimentation seems to be to enable users to describe software products and processes in ways that are convenient and effective and to support the rapid interpretation of processes in terms of software tools and procedures. This seems tantamount to creating environments in which the product specifications, process descriptions, and set of operators is specifiable by the user, and in which the environment exploits this specification to fashion its support.

2.2 Arcadia Process Programming Research

Arcadia research in this area involves two interrelated activities, process research and process programming research. The first has as its goal to gain a better understanding of the process of developing software, and the second to develop means of representing the process in terms of a process programming language and its execution support.

We are starting to understand the requirements for a process programming language. Unfortunately, the specification of such a language depends upon the accumulation of experience in writing process programs. Thus there is a "research deadlock" between the need for a process programming language and the need to write process programs. We are breaking this deadlock through some prototyping projects. These projects are aimed at studying specific language features that seem important to process programming by adding those features to existing general purpose languages and then using these enhanced languages to write process programs. These process programs then serve as vehicles for studying the suitability of the new features. By proceeding in this way we avoid the cost of creating a language from scratch while enabling the evaluation of the new feature.

One such prototyping activity involves a language called *Appl/A* [27], and the writing of process programs making use of *Appl/A*. *Appl/A* is a superset of Ada that enables the definition of relations among software objects. Ada was chosen as the base language for *Appl/A* because it seems to be sufficiently powerful to support many sorts of capabilities needed in process programming and also because it is the language targetted for support by the early versions of Arcadia. Thus we hope that Arcadia may be useful at an early date in providing support for the maintenance of some of its own critical modules. We chose to enhance Ada with a relation capability because we believe that contemporary languages have failed to provide appropriate mechanisms to represent explicitly interconnections among complex objects, a capability that seems to underlie most realistic software processes.

In *Appl/A* we are experimenting with a relation management capability in an attempt to see just what features that capability should offer. *Appl/A* supports the definition of relations as sets of arbitrary tuples of software objects. It enables users to specify just how the various components of these tuples should be related to each other, and how the consistency of these components can be verified and maintained. As such, we believe that *Appl/A* provides a very basic capability that is needed in any process programming language.

The primary vehicle for studying process programming and for the evaluation of process programming language features, such as those supported by *Appl/A*, has been the development of realistic process programs. In one such experiment we are using process programming to describe process models such as Boehm's spiral model [6,7]. In another experiment, the PMDB+ project, we are extending the PMDB model to include process descriptions emphasizing selected aspects of the life-cycle process, such as change control, configuration management and project management and

control. In this project, we will focus on typing issues, mechanisms in support of change, and architectural interface concerns.

In the *Bopeep* (But One Prototype End-to-End Process) project we are developing process programs covering key software development and maintenance phases such as requirements specification, design, testing, and maintenance. For example, in this project we have written a series of requirements specification process programs that treat requirement development as an activity aimed at creating a directed graph of requirements elements, where each element is viewed as essentially a template. The fields of the template specify such types of requirements as functionality, robustness, efficiency, and accuracy. It is the job of the requirements analyst to specify which elements contain which fields and then to put values into the appropriate fields. The requirements analyst must also indicate any relationships among the various requirements elements and between requirements elements and other software objects.

For the design phase, we are writing process programs that attempt to codify such design methodologies as Object Oriented Design [8] and Parnas' Rational Design Methodology [42]. Not surprisingly, We are finding that many of the structures and mechanisms used in developing the requirements processes are also helpful in developing the design process programs.

Our experimentation is leading to the creation of a library of prototype process programs that are helping us understand these processes. It also helps us to evaluate the features incorporated into *Appl/A* as well as understand what other process programming language features are needed.

Although *Appl/A* has proven to be quite useful, our experiments have indicated that there are important process programming features that are not supplied by *Appl/A*, nor indeed by any language built atop Ada. One obvious feature is a type hierarchy. While different nodes of a requirement element may be of different types, it is clear that they must all share some common features (e.g., author, date, parent, and children attributes), and that they are probably beneficially modelled as being subtypes of a common parent type.

It is also clear that certain types of dynamism are important. For example, often it is useful to be able to design a new requirements element "on the fly," as the need for a somewhat different requirements element description becomes apparent. To do this, a new element type must be created dynamically, then instantiated, and then filled with attribute values. In fact, the way in which a new type is filled with values might also be expected to vary with the type, indicating that subprocesses should also be dynamically defined and instantiated. Test planning provides another example of the need for dynamism. During test planning, the test plan is created as a software object. This may entail such subactivities as development of test cases, encoding of algorithmic strategies for the systematic execution of the test cases, and development of procedures for capturing test results. Much later in the development process, after code has been developed, this test plan object must be executed. This entails treating the test plan object as a process, rather than as an operand. We believe that such dynamic capabilities are important to a wide range of software processes. The passive/active nature of some software processes points to the desirability of a language in which code and data can be freely interchanged. (Lisp is an example of a language having this property.) Thus we are investigating how this property might be blended with other desirable process programming language features.

It should also be noted that this work is leading to the impression that a strictly imperative, algorithmic language is not likely to be suitable as a process programming language. Although many aspects of many kinds of software process seem to be inherently procedural and algorithmic, there are other software activities that seem best described with a declarative or rule based paradigm. The processing that systems such as Make [22] and Odin [17] carry out is guided by rules that specify how software objects should be kept consistent with each other. In our prototyping we have found a number of applications for this sort of capability. We have found, for example, that it is far easier to specify that certain attributes of certain requirements elements must satisfy a certain logical relation to each other and that there are certain well-defined activities (such as the invocation of a tool) that need to be carried out to restore needed consistency. These sorts of specifications are far easier to make than imbedding active code to restore needed consistency in a variety of places distributed throughout the algorithmic part of process programs.

On a more ambitious level, we believe that design creation processes might also be easier to code if it were possible to specify certain parts of the process by means of rules. In design creation the goal is to create a design specification. Often (e.g., in the case of the Software Cost Reduction methodology [42]), it is quite possible to specify the goal object — namely a complex structure of carefully prescribed design elements — but it is not clear how to give complete procedural details on how to construct it. In such cases it is often reasonable to create rules that guide and constrain activities, such as the selection of good candidates for design elaboration, or that can intelligently raise issues about apparent inconsistencies among design elements. Thus some aspects of design seem to be rule-based. Other aspects, such as the orderly elaboration of details of design elements and their correlation with each other, are more procedural. This suggests that a process programming language might ideally be a language that combines procedural and rule-based paradigms. *Appl/A* takes a cautious step in that direction by enabling the specification of certain fields of relations — e.g., consistency conditions among software objects — as rules. This ability to mix procedural coding with rule coding has proven useful.

The process programs we have built to date have been pieces of software of significant size and complexity. The code for these processes has spanned dozens of pages, and in many cases, lowest level details have still not been specified. Thus these process programs still leave a great deal of initiative and creativity to humans. We expect that lower levels of detail will be inserted over time, thereby making the process programs more complete and more highly dependent upon tools. It is interesting to note that, although our original intent was to simply produce code, we found that it was necessary to develop requirements specifications and designs first. In retrospect, this is totally appropriate as process programs are complex software, and we have been trained to approach the development of complex software by starting with the creation of precode artifacts. In particular the design has proven to be more useful in understanding the nature of the processes we have written than the code itself.

We expect that firm understandings of the requirements for a process programming language and of the key software development and maintenance processes will continue to develop in parallel over a period of years.

3 OBJECT MANAGEMENT

3.1 Object Management Requirements and Related Work

An environment user's primary objective is to create and/or maintain a *software product.* No matter what process program might be used in creating and maintaining it, a software product typically will be a very complex and highly interrelated collection of objects. Those objects will be of widely different kinds, ranging from source code and executable modules to documentation and test plans. Each kind of object will have an associated set of applicable operations, but operations applicable to one kind of object will generally not be appropriate for use with other kinds. This suggests that an environment's infrastructure should provide support for managing typed objects and a rich set of relationships among them.

Most environment builders have had to rely on a traditional file or database system for managing the objects associated with their environment. It is now widely believed, however, that a much richer set of capabilities for controlling object creation, access, and organization is essential to a software environment. In particular, a suitably powerful object management system will enhance the environment's support for change, integration, software reuse, and cooperative work by multiple developers.

As Figure 1 indicates, the object management system will be a major component of the Arcadia environment infrastructure. It will be responsible for managing two distinct categories of objects: the *components* of the software products being produced by users of the environment, and the *tools and information structures* that constitute the environment itself. From the process programming perspective, the former can be viewed as the (input and output) data manipulated by a process program while the latter are the operators and internal data structures of the process program.

Thus, the object management system will provide the underlying mechanism upon which the data management capabilities of a process programming language can be implemented by its interpreter. A particular process programming language might present its users exactly the same object management capabilities that the environment's object management system provides, as an assembly language presents its users exactly the same data types provided by the underlying machine. It seems likely, however, that a process programming language might offer a different view of objects than that provided by the environment's object manager. In either case, the properties of the object management system will influence the data management aspects of an environment's process programming languages.

Work on environments during the last decade has elucidated some of the important requirements for an object management system. In particular it seems clear that an object management system for a software environment should provide support for:

- types,
- relationships,
- persistence, and
- concurrency and distribution.

Each requirement poses interesting problems. The capabilities sought for each of these areas and the problems we foresee are discussed below.

Type systems. We view a type system as the primary mechanism for describing and maintaining objects. The object manager should be able to enforce the type system, hiding the internal structure of typed objects behind well-defined interfaces and strictly controlling the operations that can be performed on those objects. If all objects are instances of abstract data types, it is easier to share objects or to change their implementations. Thus, basing the object management system on a typing system that fully supports data abstraction will contribute to environment flexibility and software reuse.

Current approaches to object management in environments fall far short of providing full support for typed objects. Typically, the components of a product are treated simply as files and tools are viewed as operators applicable to the contents of those files. Usually in such systems, only a predetermined and limited number of different kinds of components (e.g., source file, object file) and operations (e.g., compiler, linker) are available. *Make* and to an even greater extent *Odin* [17] improved on this simple view by using file name extensions as a weak form of typing mechanism for files. It also allowed users to define which tools could operate on or produce files of various types. The *System Modeller*, developed as part of the *Cedar* system [32] used the term "object" for referring to the files containing product components, but did not treat the objects as instances of abstract data types. The Common APSE Interface Set (CAIS) [10] defines a system model with three kinds of nodes—file, structural, and process—but does not treat those nodes as typed objects. Recent revisions to the CAIS model [11] add a rich form of typing that specifically addresses the issue of tool evolution and environment interoperability. *Gandalf*'s SVCE mechanism [25] employs strong type checking to determine consistency of syntactic units during version control. While clearly improving on the simple use of files, all of these systems provide only partial support for typed objects. Meanwhile, work on support for typed objects within the traditional database community [61,13,77], while encouraging, is still in its primitive stages and far from providing the flexibility and power needed for object management in a software environment [4]. Recent work on rich type systems, particularly in the context of object-oriented languages, is also encouraging, but also still in its infancy. No consensus has yet emerged on a desirable and appropriate set of features for such a type system.

Thus, the kind of type system needed to describe the objects populating a software environment is one of the major object management research issues. The type system needs to be flexible and powerful enough to capture the relevant properties of environment objects. Tools, processes, and perhaps even types themselves need to be treated as typed objects. Once the capabilities of the type system are clearly delineated, suitable mechanisms for realizing those capabilities must be found. While there are many intriguing proposals for type mechanisms, it is not clear which of these (e.g., single vs. multiple inheritance, specification vs. representation inheritance, generics, static vs. dynamic binding) form a compatible set providing the capabilities needed for environments.

Relationship systems. Closely related to the ability to precisely define and maintain the typed objects in the environment is the ability to capture and maintain the relationships among those objects. Much environment work in the last ten years has focused on mechanisms for describing, reasoning about, or exploiting relationships among objects. Examples of relationships include those connecting various versions of a module, or those between the modules constituting a configuration, or those be-

tween a module and all the others that it calls, or those joining activities in a work breakdown structure. Examples of tools that reason about or exploit relationships among objects include version control systems [67,46], automated system building tools [22], call graph analyzers, and work activity management systems [23]. Explicitly indicating the relationships among an environment's tools and information structures should make it easier to modify the environment since the effect of changes can be determined. Moreover, capabilities that rely on relationships, such as inference and derivation, will enhance environment integration by allowing users to interact with the environment at a high level, leaving the intermediate steps to be automatically determined. Generic relationship capabilities will also enhance integration by providing a uniform set of capabilities across different kinds of relationships.

A weakness in previous work is that there has been no systematic treatment of the numerous and complex relationships that exist among environment objects. The CAIS notions of primary and secondary relationships (also found in the node structure of the *ALS* [66]), *Odin*'s derivation graphs, and the system models of *Cedar* represent important starting points. The concept of configuration threads found in *DSEE* [33] and the relationship capabilities for module interconnection languages provided by *Intercol* [68], *Inscape* [45], and *PIC* [74] are additional examples of partial treatments specialized to one class of relationships.

Thus, determining suitable primitives and constructors for defining the relationships needed in environments is another important object management research issue. It is not clear whether the diverse relationships needed in software development and maintenance can be captured in a single model or not. Moreover, how should the relationship structure and the type system interact? Associated with the relationship system is a set of capabilities, such as consistency checking, derivation tracking, and inferencing. Work needs to be done on identifying these capabilities and in exploring how generic such capabilities can be. For example, can generic consistency checking tools applicable to the relationship structure subsume the specialized consistency analyses associated with interface control or configuration management? Another important concern is when and how such capabilities are initiated. Some must be requested by the environment user, either directly or via an executing process. Others can be more effective if triggered by resulting events. Thus, support for "active" objects or daemons that are triggered by process or user-specified events in the environment is needed.

Persistence. The object manager must be able to preserve the components of software products and the constituents of software environments for arbitrary periods of time. Moreover, it should preserve both the structure and the restrictions imposed by the type system on how these objects can be manipulated. Under such a scheme, the traditional distinction between primary and secondary storage representations of objects is hidden within the typed object abstraction. This can free both environment users and environment builders from concern about distinctions between internal and external representations of objects and conversions between those representations. Thus, the object manager should support *persistence*, enabling objects to continue to exist beyond the lifetime of any of the tools or process programs that manipulate them and preserving the integrity of their types and relationships to other objects.

Current approaches to persistence, based on files or databases, require explicit action by the tools. Using a file system, a tool must take responsibility for converting the internal form of an object to an acceptable (e.g., linear) external form and, when needed, converting it back. There are few restrictions to assure that the type of an object is not violated (e.g., that its contents are not altered using an editor while it resides in the file) or changed (e.g., that a stack is not read back as an array). Using a database system, the tool must make calls on the database to explicitly store and retrieve information. Current databases provide support for only a limited number of types, so once again the tool must provide the conversion algorithms and there is no guarantee of type integrity. There has been some interesting work on merging database support into programming languages [2,18, 38], although implemented prototypes have been very restrictive about the supported types [2] or the underlying program model [18].

Thus, providing persistence for arbitrarily complex, typed objects is an important research issue. To permit maximum flexibility in the creation of objects and their relationships, the persistence of an object should be a property orthogonal to all other object properties. It is not clear how persistence should be recognized in a program (e.g., declared as part of the type or explicitly requested with the instantiation of an object) or how invisible persistence can be (e.g., no need to explicitly "commit" or "linearize" objects). Supporting a rich type system and providing an invisible line between memory and secondary storage raise challenging problems.

Concurrency and distribution. To allow multiple users to work conveniently on the same software development project requires support for concurrent and distributed object management. Assuming a network of workstations, different members of a development project may simultaneously invoke the same or different tools to operate on one or more of the same objects. Thus, the object manager must be able to mediate concurrent use of objects and to maintain consistency of both the objects and their relationships. Ideally, the object manager should make the distributed nature of the object base and the concurrent access to its objects invisible to users and tools in the environment.

A variety of approaches for handling distribution and concurrency have emerged from programming language [1,29,34] and file system and database research [26,69,54]. Unfortunately, no single model for dealing with these issues is universally accepted within one of these domains, let alone for objects that move between them. Moreover, some of the more popular approaches are ill-suited for use in an environment object management system. Locking schemes, for example, typically apply to entire objects and do not permit concurrent access to disjoint subsets of an object's components, which may be a frequent occurrence in an environment. Transaction schemes generally presuppose relatively short duration transactions, while a software developer's transactions may last for days or weeks. The rollback approach to conflict resolution is also of questionable value in an environment, where a rollback could discard considerable human effort.

Thus, determining appropriate constructs for expressing distribution and concurrency constraints and the underlying mechanisms needed to support these constraints is yet another major object management research issue. It is not clear what storage management primitives need to be provided to adequately capture the distribution and concurrency needs of an environment. As with types, relationships, and support for persistence, the appropriate descriptive notations must be developed as well. Also, where should the desired concurrent/distributed behavior be described—in the tools that create the actual instances of the objects, in the abstract data types that define the objects,

or in the process programs that describe how the objects are to co-exist within the environment?

3.2 Arcadia Object Management Research

As indicated above, much work has previously been done on problems related to object management. That work, however, has generally been directed toward solving individual problems, leading in some cases to incompatible solutions, and has not yet resulted in consensus on the appropriateness of those solutions. Moreover, much of the work has been oriented toward domains with needs other than those of software development and maintenance.

The approach to developing an object management system that is being taken in the Arcadia project is therefore one of synthesis and extension. In particular, we are initially looking to programming language technology for guidance in the design of a type system and the expression of distribution and concurrency constraints and initially looking to database technology for guidance in the design of mechanisms for persistence, relationships, and distribution.

It is clear that some new solutions are required to satisfy the special needs of software environments. To sharpen our understanding of those needs, in addition to examining process programs for a wide variety of activities, we are also examining a wide variety of tools that would make use of the object management system, and reflecting on our experience building *Odin*, *Keystone* [16], and *Graphite* [15], which can be viewed as primitive object management systems. We are also developing formal models for describing and evaluating the various capabilities intended for inclusion into the object management system. Finally, we are building a number of prototypes that allow us to gain direct experience with proposed capabilities.

Our conceptual view of the object management system is that it consists of three basic levels. At the top level are descriptive capabilities for specifying the types of objects, the relationships among objects, and the persistence characteristics of objects. At the middle level are capabilities for managing actual object instances and relationships, such as those to guarantee type consistency and to automatically trigger inferencing over relationships. The bottom level provides facilities for such things as storage management, concurrency control, and transaction management. We are using the models and prototypes to perform experiments within and across these levels.

Oros [50] is one model that we have developed as part of our investigation of the typing and relationship issues at the top level. It is being used to describe and evaluate the type system that we believe is appropriate for object management. *Oros* has a number of innovative characteristics. One is that objects, relationships, and operations are all treated as having co-equal importance, which reflects situations that we have encountered in trying to describe actual environment data types. Such equality is not found, for example, in the type systems of so-called object-oriented languages, where operations are unavoidably subservient to objects and relationships are not dealt with at all. Another characteristic of *Oros* is that relationships can be used as integral parts of a type definition. In other words, *Oros* allows the definition of types in terms of how their instances are related to instances of other types, not just the usual description in terms of the operations appropriate to instances of the type. As a simple example of the utility of this, consider the definition of a type for source-code modules. In a traditional definition, the fact that the instances of this type are related to instances of another type for target-code modules (thus the use of the terms "source" and "target") is only expressible implicitly. *Oros* permits this implicit aspect of the definition to be made explicit. A third characteristic of *Oros* is that it allows a distinction to be made between operations and relationships that are truly *definitional* of a type and those that are merely *auxiliary*. For example, if we view the translation of a source to a target as an operation, then that translation is to a great extent a definitional operation of the source (and, indeed, target) type. On the other hand, a pretty-printer viewed as an operation of the source type might be more appropriately considered auxiliary. We have found this distinction useful in a number of ways, but especially so in helping us address the problem of changing types in an environment, where a change to an auxiliary operation or relationship can be made to have a different impact than a change to a definitional operation or relationship.

Appl/A, which was mentioned in Section 2, and *PGraphite* [72] are two prototypes also at the top level of our conceptual layering of the object management system. *Appl/A* is exercising our ideas concerning relationships. It is intended as a vehicle for exploring the suitability of various automated constraint-satisfaction and inferencing techniques in the domain of process programming. Specifically, it provides a general framework for specifying goals in terms of "active" relationships over objects and provides mechanisms, such as backward and forward inferencing, for satisfying those goals. *PGraphite* is helping us to explore the interaction between typing and persistence, and thus complements the work on *Appl/A*. It concentrates on one kind of object, the directed graph, which is an extremely common data structure in environments. *PGraphite* provides a mechanism for the specification of types for directed graphs and automates the generation of implementations for those types in Ada. It also provides a means to indicate the persistence of particular objects as an orthogonal property of those objects. Finally, *PGraphite* provides a mechanism for specifying transactions against a persistent store as a "hook" for utilizing lower-level concurrency control and transaction management systems (see below).

Cactis [30] is a prototype at the middle level of our conceptual layering. It is a manager of object instances and relationships cognizant of the fact that the values contained in some objects and the relationships among those objects may depend upon the values in other objects and the relationships among those other objects. *Cactis* emphasizes efficient, automatic updating of values and relationships in response to changes to the values and relationships upon which they depend. One early client of *Cactis*'s services is *Appl/A*, which uses *Cactis* to manage relationships.

In addition to its role as a top-level prototype, *PGraphite* is providing insights at the middle level into the kinds of information about an object's type that must be available at run time to realize a general persistence mechanism. A version of *Appl/A* built upon *PGraphite*, where *PGraphite* would manage the persistence of *Appl/A* relationships, is planned for the near future.

Much work has already been done by others on the storage management, concurrency control, and transaction management capabilities of the bottom level of our conceptual layering [55,4,56,13] and we plan to make as much use of those results as possible. The problem we face is how to connect to those varied and evolving systems in such a way that we can easily experiment with the higher-level capabilities. Thus, our primary challenge at this level is to develop an appropriate interface mechanism. Our prototype of that mechanism is called Mneme [36]. Mneme offers a simple and efficient abstraction of low-level objects, supporting flexibility in three ways: the high-end languages/systems that can map down to that abstraction, the low-end managers/

servers that can help implement that abstraction, and the specific management policies (such as clustering of objects to optimize access) that can be specified by the Mneme user. A version of *PGraphite* is being built on top of Mneme so that we can experiment with a variety of realizations for *PGraphite*'s mechanism of persistent-store transactions.

4 USER INTERFACE MANAGEMENT

4.1 User Interface Management Requirements and Related Work

The third major component of the environment infrastructure provides the human user pleasant and efficient access to the functions supported by both the fixed and variant parts of the environment. Broad consensus exists on the qualities that distinguish good user interfaces for software environments. *Uniformity* (or *consistency*) reduces the difficulty of learning new activities and moving between activities. The *direct manipulation* interaction paradigm, using graphics and pointing devices, increases the communication bandwidth between tool and user. *Permissive* (or *non-preemptive*) interfaces allow the user to interleave activities in a natural way.

Uniformity reduces the number of details a human user must remember, and increases skill transfer between activities. A uniform interface makes the same set of operations available everywhere they make sense, and allows the user to specify an operation in the same manner wherever it is available. Interpreter-based programming environments made significant early progress toward uniformity by unifying the command language and programming language of the environment. More recently, editor-based programming environments have provided a uniform set of commands for manipulating program source code, blurring the distinction between editing, compiling, and debugging. Limited progress has been made in providing a uniform interface across a wider variety of activities. This progress has been made, for the most part, by imposing informal standards (like the *Macintosh* user interface guidelines [31]) and providing libraries of reusable components (scrollbars, menus, and the like).

Uniformity becomes both more important and harder to maintain as the scope of an environment grows. A flexible, extensible environment will contain tools contributed by a diverse community of developers and users. Moreover, both the toolset and interaction techniques can be expected to evolve during the lifetime of the environment.

A critical problem, then, is decoupling the human interface from tools so that each may evolve independently. Providing a set of reusable components is helpful, but may not be enough by itself. The *SunView* facilities [62], for instance, encourage similar visual appearance across tools, but they are not much help in establishing consistent interpretations of mouse and keyboard actions within windows managed by tools. The X Toolkit translation manager[35] goes further, allowing input translation to be decoupled from individual components by providing a uniform input translation scheme for new "widgets" as well as for those supplied with the basic library. These graphical toolkits can effectively encapsulate application-independent interaction, but they fall short of decoupling the human interface from the functionality within particular application domains.

The interface between interaction and tool functionality (in the application domain) is the most troublesome interface in modularizing interactive graphics programs. Because graphics toolkits deal entirely with the graphical domain, they do not help clean up this interface. The problem of the interface between interaction and application domain functionality becomes apparent when one notes that other tools, as well as human users, may use a tool component. As many have noted, a good human-tool interface is generally not a good tool-tool interface. It is as difficult for a tool to make use of the text manipulation facilities of a screen-oriented editor or the calculation capabilities of a spreadsheet as it would be for the human user to directly access a library of text and formula manipulation procedures. An all-purpose interface, like UNIX character streams, is unlikely to be satisfactory in either role. In current UNIX-based systems, the set of tool-usable tools is quite disjoint from the set of interactive tools.

Direct manipulation, or more precisely the illusion of directly manipulating a set of objects, requires a rich visual representation of state. This visual representation unburdens the users' short-term memory, replacing recall tasks with easier recognition tasks. (Menus serve a similar purpose with respect to remembering commands.) Objects are referred to with a pointing device and through implicit pointing (e.g., cursor position.) Changes in the representation provide immediate confirmation of user actions. The basic principles of direct manipulation are applicable to character displays, but modern bitmapped workstations are capable of richer visual representations of state. Pioneering work in the application of graphics to programming and software engineering include the *Incense* debugging system [37], the *Balsa* algorithm animation system [9], and the *Pecan* programming environment [49].

Permissiveness is an essential aspect of direct manipulation, too seldom achieved in current systems. A permissive interface allows the user to choose the next action, arbitrarily interleaving interactions with each object depicted on the screen. The converse of permissiveness is *preemption*. A preemptive interface imposes an order on user actions. The prompt/input paradigm of gathering input is a classic example of preemption. Window systems are primarily a means of limiting preemption. Windows grafted onto a conventional system in the form of multiple virtual terminals provide a minimal degree of permissiveness, sufficient for the user to temporarily escape from the control of a single application. The multiple views of *Pecan* [49] and the *Pi* debugger [14] hint at the richer interaction possible when each tool may coordinate several threads of control.

4.2 Arcadia User Interface Management Research

User interface management is an active area of research, outside the context of software environments research as well as within it. In the Arcadia project, the *Chiron* system [75] is being developed as a prototype UIMS component to demonstrate our research approach. *Chiron* adapts and extends some key ideas from current UIMS research to address the particular demands of flexible, extensible software environments.

This subsection discusses our approaches to separating application functionality from interaction facilities, managing the display, and establishing a uniform interface to all the functions supported by an environment.

Separating functionality and interaction. Several current approaches to direct manipulation interfaces carefully separate the application domain (or model) from the presentation domain (or view). This separation is especially appropriate in software environments, since few software objects are inherently graphical. Even in the case of diagrams (e.g., structure charts, data flow diagrams, SADT diagrams), the meaning of the diagram can be distinguished from its representations in terms of boxes,

lines, and text. In the Arcadia-1 prototype environment, tool components that manipulate model objects can be largely freed of concern with view objects. The *Chiron* system is used to build encapsulated tool components that maintain consistency between objects in the model and view domains, so that view objects accurately reflect the state of model objects and model objects properly respond to direct manipulation of view objects.

In "editor" environments supporting a narrow set of objects and functions, a central tool component typically maps the application data structure (usually a parse tree) to a visual representation. Separation of concerns between application domain and presentation domain is achieved, but at the cost of requiring all environment facilities to operate on a single shared data structure rather than a variety of data structures suited to different applications. Environments of wide scope require a more flexible scheme.

In the Arcadia-1 prototype environment, each abstract data type in an application domain may have an associated *artist* for maintaining a corresponding view object. An artist encapsulates decisions about how each particular data type is depicted; there is no requirement for all tools to share a single data type or data model, beyond the requirement that objects be cleanly encapsulated as abstract data types.

Artists for data structures were introduced by Myers [37] in the *Incense* symbolic debugging system. *Loops* [59,60] binds the equivalent of artists to objects using a specialized form of inheritance called *annotation*. *Chiron* adopts a more formal version of annotation for binding artists to abstract data types. An annotation on an abstract data type may add new operations, add local state (*instance variables,* in the nomenclature of object-oriented programming), and extend existing operations. New operations and extensions to existing operations may modify only new state.[3] An artist adds new state to keep track of the depiction of an object, and extends existing operations to update the depiction when the object is modified (Figure 2).

The essential characteristic of annotation as a mechanism for binding artists to abstract data types is that neither the semantics nor the syntax (signatures of operations) of an object are changed. A tool component need not be modified just because the object it is manipulating is depicted on the screen; the interface to tools is not corrupted by the interface to human users.

Managing the display. The view objects created by artists could be actual bitmaps, but it is generally better to interpose an intermediate level of representation between application objects and their concrete depiction on the screen. *Chiron* provides a diagram-oriented $2\frac{1}{2}$D hierarchical structure for describing displays, including nested and overlapping windows. Artists manipulate this *abstract depiction*. A separate rendering agent maps it into actual bitmap images (Figure 3).

Operating on the abstract depiction has several advantages over purely procedural abstractions for operating on bitmaps. An artist may modify a display by making small changes to the abstract depiction, without concern for the extent of changes to the bitmap image (e.g., if moving a circle causes a previously obscured rectangle to become visible). More importantly, an abstract depiction can be used as a basis for input correlation, relating an input action (e.g., mouse click) with a particular application object. Whereas window systems typically provide input correlation only down to the level of individual windows, *Chiron* provides correlation to the level of individual polygons, lines, and so forth.

[3] Most current implementations of annotation-like mechanisms do not enforce this restriction, but it is essential for reasoning about annotated abstract data types. The property we desire is that proofs involving a type T remain valid when an annotation T' is substituted for T.

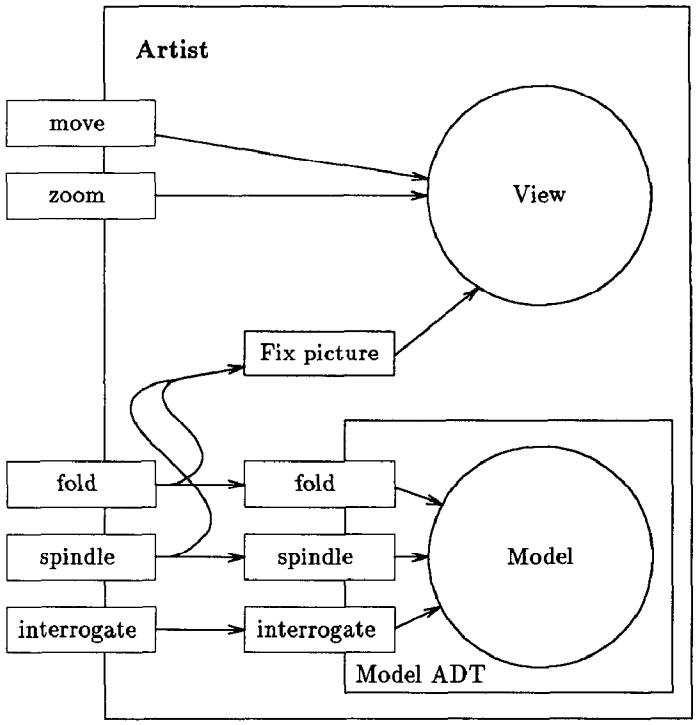

Figure 2: An artist is logically "wrapped around" an abstract data type.

Input model. Approaches to processing user input can be classified according to whether input routines appear as subroutines to the application (called the *prompting* or *internal control* model), or the application appears as a subroutine(s) to the input processor (*dispatching* or *external control* model), or the input routines and application are logically concurrent, cooperating processes. The prompting model is inferior from the user's point of view, because it is highly preemptive — the user has too little control over the program. The dispatch model, on the other hand, distorts the natural logic of some applications by forcing the programmer to "flatten" control structures.

Chiron supports a concurrent model of input processing. Each object type may be associated with an agent for handling events on objects of that type, and these agents may proceed concurrently with other processing in tools and the user interface. Avoiding preemption by supporting concurrency is especially important when interpreting process programs — when a process program calls for a human activity, the user still maintains control and may freely interleave the new activity with other current activities.

Approaches to uniformity. There is no complete technical fix to insure uniformity, if one is unwilling to sacrifice flexibility and extensibility in a software environment. The most that can be done by the user interface component is to promote uniformity by a variety of means.

Centralized interpretation of low-level input can be used to achieve a basic level of uniformity. For instance, if the lexeme *select* is bound to a single click of the leftmost mouse button, then the application will receive the event *select,* rather than a raw key click, when the button is pressed. Binding of lexemes to raw events should always be under control of the user, rather

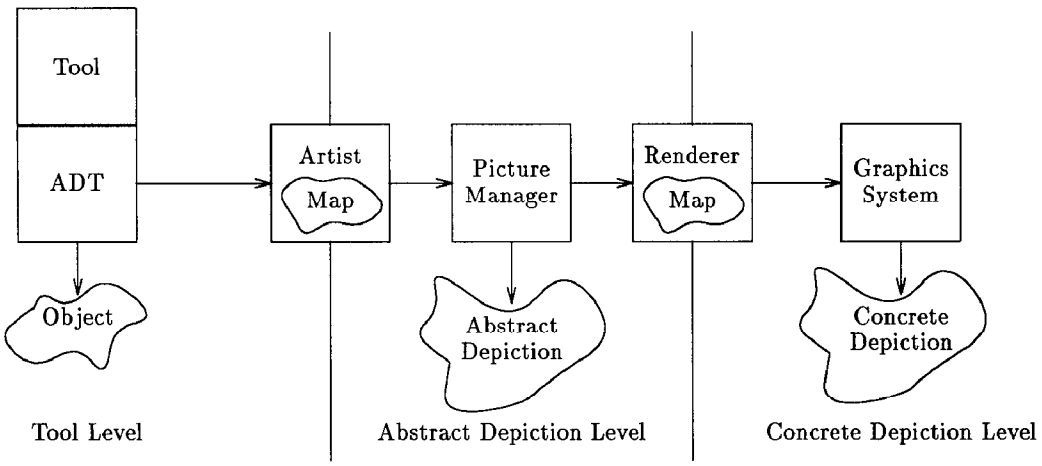

Figure 3: Artists manipulate an abstract description of the display, and a rendering agent maps the abstract depiction to bitmap images.

than the tool builder. Techniques adequate for administering this level of interpretation are well known (e.g., the TIP tables of *Cedar*).

Central administration can also guarantee consistent interpretation of a small set of "global" commands, for instance, terminating a tool. Anyone who has attempted to kill an unfamiliar program in the UNIX system with keyboard incantations will appreciate the importance of such guarantees.

Reusable components are a complementary approach to promoting uniformity. Application-independent components, such as scrollbars, are already in common use. In the Arcadia-1 prototype environment, clean encapsulation of interaction facilities makes it feasible to provide reusable components for data abstractions in a particular application domain (e.g., Petri nets), as well. Since artists in an Arcadia environment are associated with abstract data types, the path of least resistance for tool developers is to reuse an artist for all interactive tools dealing with a particular data abstraction.

5 CONCLUSION

The Arcadia consortium has been formed to explore a number of issues in software environments. We are attempting to make major strides in the development of fundamental technologies, develop prototypes, conduct careful empirical studies, and move the technology to industrial practice.

This paper presents the approach being pursued by Arcadia researchers investigating software environment architectures. This project is exploring an architecture to support flexibility and extensibility as well as tight internal and external integration. We feel these issues, while often in conflict with each other, are necessary ingredients for environments to fulfill their potential of assisting users in software development and maintenance activities. Our research approach involves simultaneously investigating several challenging research areas and synergistically striving to develop compatible solutions in each.

The environment architecture that we have proposed has separated the basic components of the infrastructure, or fixed part, from the process programs, tools, and objects that constitute the variant part. The overriding job of the fixed part is to facilitate tight integration over the flexible/extensible variant part. This clear separation of concerns has helped to modularize the problem and to identify some important open questions and promising research directions.

Our approach is based on the process programming paradigm, where software processes are formally captured by programs, which are then executed by the process program interpreter. Formal descriptions of software processes, presented in an expressive language along with tools for creating and modifying such programs, provide a basis for flexibility and extensibility. The user interface management system supports external integration by providing a uniform method of communication between humans and executing software processes. The object management system supports internal integration by providing typing, relationships, persistence, distribution, and concurrency capabilities upon which process programs can be interpreted. We believe that among the most important results of the Arcadia effort will be understanding the tradeoffs that are possible and desirable in achieving effective user interface and software object management support in the face of the high degree of flexibility and extensibility afforded by process programming.

The Arcadia project has just completed its first year of funding, and results are still very preliminary. Arcadia researchers realize that to be convincing and to gain as much insight as possible, realistic prototypes must be subjected to well-designed empirical evaluation. The Arcadia researchers are currently implementing prototypes of all the major components of the environment infrastructure. Analysis tools are also being developed as part of this project and their insertion into the variant part, along with suitable process programs, will provide a challenging test for the architecture. In addition, an evaluation framework is being developed to enable meaningful empirical studies to be undertaken. Finally, technology transfer activities are being explored so that realistic industrial feedback can be obtained. These combined activities should lead to valuable research results, significantly advancing our knowledge and capabilities in software environments.

ACKNOWLEDGEMENTS

Arcadia has benefited from the ongoing encouragement of William L. Scherlis and Stephen L. Squires.

We gratefully acknowledge the contributions of participants from each institution of the consortium, who have been instrumental in shaping virtually every aspect of Arcadia. During the past two years key contributions have come from D. Baker, B. Boehm, A. Brindle, D. Fisher, D. Heimbigner, C. LeDoux, M. Penedo, D. Richardson, I. Shy, S. Sykes, W. Tracz, and S. Zeil.

Additional contributions have been made by R. Adrion, G. Barbanis, G. Clemm, R. Cowan, J. Durand, E. Epp, S. Gamalel-Din, S. Graham, G. James, C. Kelly, S. Krane, R. King, D. Luckham, T. Nguyen, K. Nies, A. Porter, W. Rosenblatt, R. Schmalz, C. Snider, S. Sutton, P. Tarr, and D. Troup.

References

[1] *Military Standard Ada Programming Language (ANSI/MIL-STD-1815A-1983).* American National Standards Institute, Jan. 1983.

[2] M. Atkinson, P. Bailey, K. Chisholm, P. Cockshott, and R. Morrison. An approach to persistent programming. *The Computer Journal*, 26(4):360–365, 1983.

[3] T. E. Bell, D. C. Bixler, and M. E. Dyer. An extendable approach to computer-aided software requirements engineering. *IEEE Trans. on Software Engineering*, SE-3(1):49–60, Feb. 1977.

[4] P. A. Bernstein. Database system support for software engineering. In *Proceedings of the Ninth International Conference on Software Engineering*, pages 166–178, IEEE Computer Society Press, Monterey, California, March 1987.

[5] D. Bjorner. On the use of formal methods in software development. In *Proceedings of the Ninth International Conference on Software Engineering*, pages 17–29, IEEE Computer Society Press, Monterey, California, March 1987.

[6] B. Boehm. A spiral model of software development and enhancement. *Computer*, May 1983.

[7] B. Boehm and F. Belz. Applying process programming to the spiral model. In *Proceedings of the Fourth International Software Process Workshop*, pages 1–11, Moretonhampstead, Devon, United Kingdom, May 1988.

[8] G. Booch. Object-oriented development. *IEEE Trans. on Software Engineering*, SE-12(2):211–221, Feb. 1986.

[9] M. H. Brown and R. Sedgewick. A system for algorithm animation. *Computer Graphics*, 18(3):177–186, July 1984.

[10] *DOD-STD-1838, Common Ada Programming Support Environment (APSE) Interface Set (CAIS).* Department of Defense, Oct. 1986.

[11] *Common Ada Programming Support Environment (APSE) Interface Set (CAIS), Proposed Military Standard DOD-STD-1838A (Revision A).* Department of Defense, May 1988.

[12] J. R. Cameron. An overview of JSD. *IEEE Trans. on Software Engineering*, SE-12(2):222–240, Feb. 1986.

[13] M. J. Carey, D. J. DeWitt, D. Frank, G. Graefe, J. E. Richardson, E. J. Shekita, and M. Muralikrishna. *The Architecture of the EXODUS Extensible DBMS: A Preliminary Report.* Technical Report CS-644, Computer Sciences Department, University of Wisconsin – Madison, May 1986.

[14] T. A. Cargill. The feel of Pi. In *Winter 1986 USENIX Technical Conference*, pages 62–71, USENIX Association, Denver, Colorado, Jan. 1986.

[15] L. A. Clarke, J. C. Wileden, and A. L. Wolf. Graphite: A meta-tool for Ada environment development. In *Proceedings of the IEEE Computer Society Second International Conference on Ada Applications and Environments*, pages 81–90, IEEE Computer Society Press, Miami Beach, Florida, Apr. 1986.

[16] G. M. Clemm, D. Heimbigner, L. Osterweil, and L. G. Williams. Keystone: a federated software environment. In *Proceedings of the Workshop on Software Engineering Environments for Programming-in-the-Large*, pages 80–88, June 1985.

[17] G. M. Clemm and L. J. Osterweil. *A Mechanism for Environment Integration.* Technical Report CU-CS-323-86, University of Colorado, Boulder, Jan. 1986.

[18] *CLF Overview.* USC Information Sciences Institute, Marina del Rey, California, March 1986. Informal Report.

[19] V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang. Programming environments based on structured editors: the Mentor experience. In *Interactive Programming Environments*, pages 128–140, McGraw-Hill Book Co., New York, 1984.

[20] M. Dowson, editor. *Proceedings of the 3rd International Software Process Workshop*, IEEE Computer Society Press, Breckenridge, Colorado, Nov. 1986.

[21] G. Estrin, R. S. Fenchel, R. R. Razouk, and M. K. Vernon. SARA (System ARchitects Apprentice): modeling, analysis, and simulation support for design of concurrent systems. *IEEE Trans. on Software Engineering*, SE-12(2):293–311, Feb. 1986.

[22] S. I. Feldman. Make—a program for maintaining computer programs. *Software — Practice & Experience*, 9(4):255–265, Apr. 1979.

[23] C. Green, D. Luckham, R. Balzer, T. Cheatham, , and C. Rich. *Report on a Knowledge-Based Software Assistant.* Technical Report, Kestrel Institute, June 1983.

[24] A. N. Habermann and D. Notkin. Gandalf: software development environments. *IEEE Trans. on Software Engineering*, SE-12(12):1117–1127, Dec. 1986.

[25] A. N. Habermann and D. E. Perry. System composition and version control for ada. In H. Huenke, editor, *Software Engineering Environments*, pages 331–343, North-Holland, 1981.

[26] D. Heimbigner and D. McLeod. A federated architecture for information management. *ACM Transactions on Office Information Systems*, 3(3):253–278, July 1985.

[27] D. Heimbigner, L. Osterweil, and S. Sutton. *APPL/A: A Language for Managing Relations Among Software Objects and Processes.* Technical Report CU-CS-374-87, University of Colorado, Department of Computer Science, Boulder, Colorado, 1987.

[28] P. Henderson, editor. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, ACM SIGPLAN, Pittsburgh, Pennsylvania, Apr. 1984. (Appeared as SIGPLAN Notices, 19(51), May 1984.).

[29] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[30] S. Hudson and R. King. The Cactis project: database support for software environments. *IEEE Trans. on Software Engineering*, June 1988. (to appear).

[31] *Inside Macintosh.* Apple Computer, Inc., Cupertino, California, promotional edition, March 1985.

[32] B. W. Lampson and E. E. Schmidt. Organizing software in a distributed environment. In *Proceedings of the ACM SIGPLAN '83 Symposium on Programming Language Issues in Software Systems*, pages 1–13, June 1983.

[33] D. B. Leblang and J. Robert P. Chase. Computer-aided software engineering in a distributed workstation environment. *SIGPLAN Notices*, 19(5):104–112, May 1984. (Proceedings of the First ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments).

[34] B. Liskov and R. Scheifler. Guardians and actions: linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[35] J. McCormack. *X Toolkit Intrinsics — C Language X Interface.* Technical Report Review Version 2.1, Western Software Laboratory, Digital Equipment Corporation, Feb. 1988.

[36] J. Moss and S. Sinofsky. *Managing Persistent Data with Mneme: Issues and Applications of a Reliable, Shared Object Interface.* Technical Report 88–30, COINS, University of Massachusetts, Amherst, MA, Apr. 1988.

[37] B. A. Myers. Incense: A system for displaying data structures. *Computer Graphics*, 17(3):115–125, July 1983.

[38] J. A. Orenstein, S. K. Sarin, and U. Dayal. *Managing Persistent Objects in Ada.* Technical Report CCA–86–03, Computer Corporation of America, Cambridge, Massachusetts, May 1986.

[39] L. J. Osterweil. *A Process-Object Centered View of Software Environment Architecture.* Technical Report CU-CS-332-86, University of Colorado, Boulder, May 1986.

[40] L. J. Osterweil. Software environment research: Directions for the next five years. *Computer*, 14(4):35–43, 1981.

[41] L. J. Osterweil. Software processes are software too. In *Proceedings of the Ninth International Conference on Software Engineering*, pages 2–13, Monterey, CA, March 1987.

[42] D. L. Parnas and P. C. Clements. A rational design process: how and why to fake it. *IEEE Trans. on Software Engineering*, SE-12(2):251–257, Feb. 1986.

[43] M. H. Penedo. Prototyping a project master data base for software engineering environments. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Development Environments*, Palo Alto, California, Dec. 1986.

[44] M. H. Penedo and E. D. Stuckle. PMDB — A project master database for software engineering environments. In *Proceedings of the 8th International Conference on Software Engineering*, pages 150–157, London, England, Aug. 1985.

[45] D. E. Perry. Software interconnection models. In *Proceedings of the Ninth International Conference on Software Engineering*, pages 61–69, March 1987.

[46] D. E. Perry. Version control in the Inscape environment. In *Proceedings of the Ninth International Conference on Software Engineering*, pages 142–149, March 1987.

[47] C. Potts, editor. *Proceedings of Software Process Workshop*, IEEE Computer Society Press, Egham, Surrey, United Kingdom, Feb. 1984.

[48] R. A. Radice, N. K. Roth, A. C. O. Jr., and W. A. Ciarfella. A programming process architecture. *IBM Systems Journal*, 24(2):79–90, 1985.

[49] S. P. Reiss. PECAN: program development systems that support multiple views. *IEEE Trans. on Software Engineering*, SE-11(3):276–285, 1985.

[50] W. Rosenblatt, J. Wileden, and A. Wolf. *Preliminary Report on the OROS Type Model.* Technical Report 88–70, COINS, University of Massachusetts, Amherst, MA, Aug. 1988.

[51] D. T. Ross and K. E. S. Jr. Structured analysis for requirements definition. *IEEE Trans. on Software Engineering*, SE-3(1):6–15, Feb. 1977.

[52] G. Ross. Integral C — A practical environment for c programming. In *Proceedings of the Second ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, pages 42–48, Dec. 1986.

[53] W. W. Royce. Managing the development of large software systems. In *Proceedings, IEEE WESCON*, pages 1–9, IEEE, Aug. 1970. Also reprinted in Proceedings 9th International Conference on Software Engineering, pp. 328–338.

[54] M. Satyanarayanan, J. H. Howard, D. A. Nichols, R. N. Sidebotham, A. Z. Spector, and M. J. West. The ITC distributed file system: principles and design. *Operating Systems Review*, 19(5):35–50, 1985.

[55] A. H. Skarra, S. B. Zdonik, and S. P. Reiss. An object server for an object-oriented database system. In *Proceedings of the Object-Oriented Database Systems Workshop*, pages 196–204, IEEE Computer Society Press, Sep. 1986.

[56] A. Z. Spector. *Distributed Transaction Processing in the Camelot System.* Technical Report CMU–CS–87–100, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, Jan. 1987.

[57] *Proceedings of 4th International Software Process Workshop*, Moretonhampstead, Devon, United Kingdom, May 1988.

[58] T. A. Standish and R. N. Taylor. Arcturus: A prototype advanced Ada programming environment. *Software Engineering Notes*, 9(3):57–64, May 1984. (Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments).

[59] M. Stefik and D. Bobrow. Object-oriented programming: Themes and variations. *AI Magazine*, 6(4):40–62, Winter 1986.

[60] M. J. Stefik, D. G. Bobrow, and K. M. Kahn. Integrating access-oriented programming into a multiparadigm environment. *IEEE Software*, 3(1):10–18, Jan. 1986.

[61] M. Stonebraker and L. A. Rowe. The design of POSTGRES. In *Proceedings of the ACM SIGMOD '86 International Conference on Management of Data*, pages 340–355, June 1986.

[62] *SunView Programmer's Guide.* Sun Microsystems, Inc., Mountain View, California, Feb. 1986.

[63] T. Teitelbaum and T. Reps. The Cornell Pprogram Synthesizer: A syntax directed programming environment. *Communications of the ACM*, 24(9):563–573, Sep. 1981.

[64] W. Teitelman. A tour through Cedar. *IEEE Trans. on Software Engineering*, SE-11(3):285–302, 1985.

[65] W. Teitelman and L. Masinter. The Interlisp programming environment. *Computer*, 14(4):25–33, Apr. 1981.

[66] R. M. Thall. The KAPSE for the Ada Language System. In *Proceedings of the AdaTEC Conference on Ada*, pages 31–47, ACM, Oct. 1982.

[67] W. F. Tichy. Design, implementation, and evaluation of a revision control system. In *Proceedings of the Sixth International Conference on Software Engineering*, pages 58–67, Tokyo, Japan, Sep. 1982.

[68] W. F. Tichy. Software development control based on module interconnection. In *Proceedings of the Fourth International Conference on Software Engineering*, pages 29–41, IEEE Computer Society Press, Munich, West Germany; Sep. 1979.

[69] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS distributed operating system. *Operating Systems Review*, 17(5):49–70, 1983.

[70] P. T. Ward. The transformation schema: an extension of the data flow diagram to represent control and timing. *IEEE Trans. on Software Engineering*, SE-12(2):198–210, Feb. 1986.

[71] A. I. Wasserman, P. A. Pircher, D. T. Shewmake, and M. L. Kersten. Developing interactive information systems with the user software engineering methodology. *IEEE Trans. on Software Engineering*, SE-12(2):326–345, Feb. 1986.

[72] J. Wileden, A. Wolf, C. Fisher, and P. Tarr. PGraphite: an experiment in persistent typed object management. In *Proceedings SIGSOFT '88: Third Symposium on Software Development Environments*, Dec. 1988.

[73] J. C. Wileden and M. Dowson, editors. *Proceedings of the International Workshop on the Software Process and Software Environments*, ACM SIGSOFT, Coto de Caza, Trabuco Canyon, California, March 1985. (Appeared in ACM Software Engineering Notes, Vol.11, No.4, Aug. 1986).

[74] A. Wolf, L. Clarke, and J. Wileden. The AdaPIC toolset: supporting interface control and analysis throughout the software development process. *IEEE Transactions on Software Engineering*, 1988. (To appear).

[75] M. Young, R. N. Taylor, and D. B. Troup. Software environment architectures and user interface facilities. *IEEE Trans. on Software Engineering*, June 1988.

[76] P. Zave and W. Schell. Salient features of an executable specification language and its environment. *IEEE Trans. on Software Engineering*, SE-12(2):312–325, Feb. 1986.

[77] S. B. Zdonik and P. Wegner. Language and methodology for object-oriented database environments. In *Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences*, pages 378–387, Jan. 1986.