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

COINS Technical Report 88-43

May 1988

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, Redondo Beach, California[4]

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

Arcadia Document Number UCI-88-07

[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 Defense Advanced Research Projects Agency, through ARPA Order 6100, Program Code 7E20, which was funded through grant CCR-8705162 from the National Science Foundation, with additional funding through National Science Foundation grant DCR-0403341 and U.S. Department of Energy 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.

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 consortium's joint research is focused on the construction of software environments that are highly integrated, yet flexible and extensible enough to support experimentation with alternative software processes. This has led us to view an environment as being composed of two distinct, cooperating parts. One is a *variant* part, consisting of process definitions plus the tools and objects specific to the defined processes. The other is a fixed part, or *infrastructure*, providing an invariant basis on which various process definitions, and their associated tools and objects, can be supported. The major components of the infrastructure are a process programming language and interpreter, user interface facilities, and an object management subsystem. The process programming facility allows precise definition and automated support of software development and maintenance activities. The object management subsystem provides persistent storage for highly structured, typed objects, and supports the type system of the process programming language. The user interface management system makes the system hospitable for human users, who carry out the creative parts of process programs.

Keywords: environment, software process, process programming, object management, user interface management system

Contents

1 A Characterization of Software Environments 1

2 Software Process Definition and Interpretation 3

3 Object Management 12

4 Interface with the Human User 20

5 Some Details of the Architecture of Arcadia 26

6 Summary and Conclusion 29

References 30

1 A Characterization of Software Environments

A common thread that runs through the literature on software environments is that the purpose of environments is to *support the user* in some software development or maintenance activity. Sometimes this activity is highly constrained and well defined, such as constructing syntactically correct source code. Other environments have broader scope, but are highly restrictive in the order of events that are permitted. Still other environments are simply collections of tools and data management facilities that are believed to be helpful in a broad arena of software evolution activities.

Thoughtful consideration of this notion of "supporting the user" yields some important insights. First, if the activities that an environment supports are not precisely and unambiguously described, it is difficult for potential users to assess whether their needs will be met. Second, the facilities provided by an environment may be so loosely structured that even though they could support a variety of activities, if all structuring and composition is solely the end user's responsibility, for which no automated support is provided, then undue burden is placed upon the user. It is likely that such an environment will be used to support only the simplest and smallest activities. Finally, change to virtually any software development or maintenance activity is inevitable. Users will wish to incorporate new tools and development methodologies. Thus, if the environment's structure is closely entwined with the original activity, then accommodating change may be difficult and costly, or not possible at all.

In our estimation, therefore, a useful environment will

- support clearly and precisely defined activities,
- mechanize the structuring and composition of support functions, and
- accommodate changes and personal preferences.

To enable an Arcadia environment to meet these goals, we chose to divide the environment into two parts: a fixed part and a variant part. The aspects of an environment that are prone to change, such as the tool set, belong to the variant part. The unchanging mechanisms necessary to ensure the integrity, extensibility, flexibility, and integration of the environment are encapsulated in the fixed part. We believe this division is a critical separation of concerns.

More specifically, the variant part consists of the evolving set of data objects (such as specifications, programs, and test data) along with rigorous, detailed descriptions of software development or maintenance activities, which we term "process programs" [34] [35]. These activities are defined in terms of individual specific operators, which correspond to the classical notion of tools. These operators can themselves be modeled as process programs (if not actually implemented as such), and are correspondingly in the variant part too.

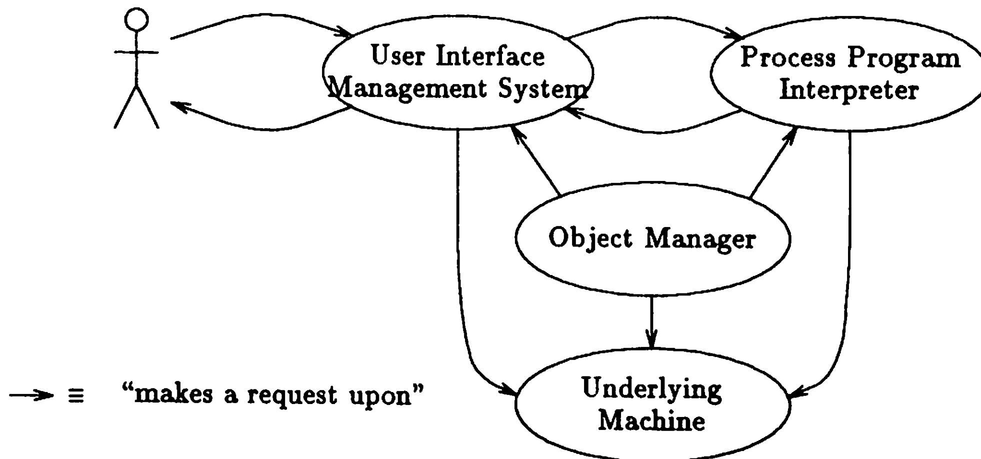

Figure 1: The active entities of an environment infrastructure with the "makes a request upon" relationship shown between them. The variant parts of an environment are not shown.

The fixed part, which can also be termed the *environment infrastructure*, consists of all the mechanisms necessary for the automated interpretation of process programs. Specifically, it consists of a language for writing process programs, an agent that enables interpretation of process programs, including mechanisms for managing persistent objects, and facilities for providing interfaces to the human user. The fixed part also encapsulates assumptions regarding its underlying machine. That is, it may make use of an operating system, a storage manager, and so forth. These assumptions are also components of the infrastructure. The components of the infrastructure that are active entities are shown in Figure 1.

The user may initiate action by making a request for support of some activity. The request is passed along, through the user interface management system, to the process program interpreter. Assuming that the request is well formed, the interpreter initiates a series of actions, executing the process specified by the user. This execution will involve invoking operators upon operands — both of which are managed by the "object manager" component. Examples of operators include lexers, parsers, code generators, debuggers, test data generators, and specification and design language processors. Examples of operands include source code, executable modules, test data, specifications, designs, management data, symbol tables, various internal representations of programs, designs, system generation directives (e.g., make files), and intermediate analysis results.

Many of the operators will themselves be processes that cause additional actions to occur. Note that the entities belonging to the variant part of the environment, such as tools and data objects, are *managed* by the infrastructure — they are called

into action at appropriate times and do their work — but they are not themselves components of the infrastructure.

There are, of course, operations in a software process that can only be performed by people. In essence, the mundane aspects of processes are automated in this view of environments, while the creative aspects are performed by creative agents — people. Accordingly the interpreter may issue a request for an operation to be performed by the user, passing the request through the user interface management system.

The user interface management system may directly request services of the object manager for storage of windows and depictions of objects. Finally, the underlying machine provides services for each of the other three automated components of the infrastructure.

In short, the infrastructure is a virtual machine for the interpretation of process programs. (It is for this reason that we prefer to view the interpreter as part of the infrastructure, rather than a tool belonging to the variant part.)

Clearly we are burying many of the critical and interesting technical issues inside the process interpretation system and the object manager. The subsequent sections of this paper, which are organized around the entities shown in Figure 1, clarify and elucidate the ideas suggested here. The notions of software processes, process programming languages, and the interpretation of process programs are considered first, in Section 2, as they are central to our view of environments. These are followed by discussions of object management in Section 3 and the user interface management system in Section 4. Some details of the architecture of an Arcadia environment are introduced in Section 5. Additional details of these issues, plus discussion of the necessary capabilities of the underlying machine, measurement and evaluation of environments, and development and technology transfer can be found in [53].

2 Software Process Definition and Interpretation

Software Processes

Perhaps the most striking feature of the environment architecture described here is that it empowers users to rigorously specify their software products and product types *and* rigorously and explicitly specify alterable process programs to guide in the development and maintenance of these products. Previous environment architectures have exploited only primitive notions of explicitly specified products and processes. They have supported relatively fixed processes and products, often specified only implicitly. Moreover, the user's freedom to specify the process supported or the type of product produced by the environment was generally sharply restricted.

All software projects have as their goal the creation and/or modification of soft-

ware products. They work towards this goal by carrying out software operations on software operands. Thus, at the most basic level, any mechanism for supporting the specification of the coordination of such operations must be viewed as a process programming mechanism. Hence even operating system control languages can be viewed as primitive process programming languages. In that context, application language processors and system utilities are process programming language operators and the files managed by the operating system's file system are their operands. When a user issues commands to the operating system, it effects the requested operations. Thus, command files or scripts are primitive process programs, using the operating system command language as a process programming language.

It is significant that these primitive process programs are used to indicate the ways in which human operations are to be coordinated with machine executable operations. For example, users employ operators incorporated into tools, such as browsers, to help them carry out (human) selection operations. Selected objects are then used as operands to subsequent software operators, such as edit and compile. Users, moreover, often carry out standard sequences of operations at certain fixed times during software projects. They may invoke scripts to automatically compile new code, or automatically check consistency of new code with support libraries. These scripts orchestrate the interaction between machine operations (e.g., compiling and checking) and human operations (e.g., creating code). In this way, the scripts are small but particularly good examples of process programs.

Scripts are also used to automatically create new objects and maintain certain types of consistency between new and existing objects. For example, scripts are used to automatically recompile source code when support libraries have been changed, or to recreate executables when source code has been changed. The *Make* system in the *UNIX*[1] operating system [19] is an example of a capability whose goal is to facilitate the creation of powerful scripts of just this sort, through the use of a terse and precise notation. Clearly this notation is a process programming language, albeit a limited one.

Operating system command languages and *Make* lack the power needed to effectively program large and complex processes. One of their most basic deficiencies is their weak facilities for defining software objects as instances of types. Our ideas about the need for, and implementation of, an object typing facility are elaborated more fully in Section 3. Our basic position is that typing offers a powerful vehicle for organizing the basic objects to be managed in a software project, and for defining and organizing the operators — both human and machine executable — to be employed by the project[2].

---

[1] *UNIX* is a trademark of AT&T.

[2] In either case, the semantics of operations can be formally defined using, for example, pre- and post- conditions. These conditions, in turn, utilise the accessing primitives that participate in defining the object types.

Going further, we believe that a process programming language must support specification of the order in which operators are to be applied to operands. 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). In fact it seems that paucity of control flow expressive power may well be the weakest aspect of most operating system command languages.

Interestingly, other early attempts at rigorously expressing software process focused directly on these aspects. Most notable among these attempts have been efforts to use diagrammatic representations to depict the major features of large-scale software processes. In this work, principal software processes were represented by boxes, and flow of control relations among them were represented by arrows. The "Waterfall Model" of software development relied upon this device in an early attempt to represent an overall software development process [43]. Almost immediately, software process modelers attempted to use these pictorial representations to also show other relations such as data flow or process hierarchy. Even later work attempted to simultaneously show diagrammatic representations of many key relations among a variety of types of software objects. In order to do this, data objects were differentiated from process objects by making distinctions between the shapes of the boxes representing them. Distinctions among relations were made by defining different shapes of arrowheads and different colors and shapes of arrows to represent these different relations. Some examples of advanced diagrammatic process representations are ETVX boxes [37], SADT diagrams [41] and Software Development Graphs [5].

Inevitably these efforts are limited by the fact that there are arbitrarily many valid relations among the large number of software objects required to adequately describe software processes, and that different users may at different times wish to study various combinations of them. Creating one single diagram containing all of these relations is hardly a solution, as such a diagram is so complicated as to confound all understanding. Creation of a single internal representation capturing all of the complex relations among the objects and subprocesses of a software process, and then relying upon tools to draw specific diagrams ("views") upon request, seems to be a plausible solution to this dilemma. We believe that a process program, written in a suitable language, is the appropriate device for representing this set of objects, subprocesses and their interrelations.

In our view, earlier software environments can also be viewed as primitive attempts to address the desirability of being able to precisely program key software processes by means of code written in a powerful and well defined language. For instance, systems such as *PAISLey* [67], *RSL/REVS* [3], *SARA* [18], Data Flow Diagram Designs [61], Jackson System Development [9], and *USE* [62], provide strong support for the creation of certain fixed types of requirements specifications and

designs. They guide users to the development of design or specification objects in a particular fixed discipline and format, which is usually pictorial or graphical. For example, *RSL/REVS* aids users in creating, analyzing, and maintaining designs as instances of software objects types which are essentially hierarchies of heavily annotated graph structures. In some of these systems the user is able to tailor and adapt these software object types. Invariably, however, these adaptations can be made only within a narrow range. For example, users of such environments may be able to select the specific fields to be incorporated into a design node, but only from a given fixed list of fields and types.

In addition, these design and specification support environments attempt to lead the user through specific procedural processes that are intended to expedite the creation of designs and improve the chance that designs are well formed and in compliance with the supported methodology's guidelines. Accordingly, such environments are often either indifferent or overtly hostile towards attempts to create design objects of new or different types, or to follow development procedures that have been devised by the user. From our perspective, these environments contain "hard-coded" object specifications and processes (which they effectively support). They are not hospitable to user attempts to make significant alterations to such processes or design object specifications.

There are also other environments aimed at supporting the development of code. Environments such as *Interlisp* [56], *Arcturus* [47], and *Cedar* [55] integrate facilities to support the creation of code in specific languages. They support such common activities as editing, parsing, debugging, and documentation. These environments assume that user activities can be uniformly and smoothly integrated by viewing them as examination and transformation of a single uniform representation of one product — code or a representation of the code. In *Interlisp* all software products, as well as the procedures and tools used to create them, are considered to be instances of lists. In *Arcturus*, software products and the commands used to manipulate them are all instances of Ada code. As long as the user's activities are effectively modeled in these ways, these environments provide strong support. As the user seeks to model software products and processes as objects of different types, support from these environments falters and becomes awkward.

Similarly, intelligent editors such as the *Cornell Program Synthesizer* [54], *Integral-C* [42], *Gandalf* [21], and *Mentor* [16] are all effective in integrating user activities, but only over a restricted range. Here, the integration rationale is that user activities all revolve around a parsed representation of code in a specific language. Experience has shown that this representation supports many user activities more effectively than a textual representation of the code. Shifting focus from text to an internal representation, however, does not solve the problems posed by restricting users from being able to create and manipulate software objects of types of their own creation using explicit processes of their own creation. Structure editors implicitly

assume that users are concerned with a few fixed types of objects.

In all of these cases the effectiveness of the support tools is drastically reduced when the process that the user wishes to carry out is not anticipated by the environment. In the case of code synthesis environments such as *Mentor* or *Interlisp*, there is only weak support for user attempts to operate on object types not related to code. In the case of a design environment, when the user attempts to stray beyond the supported methodology, or attempts to carry out such processes as coding or testing, support similarly is weak.

Support for only limited, pre-determined processes is particularly disturbing to us because we note that that there is currently little consensus about what constitutes adequate software products and effective software processes. Thus products and processes must therefore be expected to vary from user to user, from location to location, and from time to time. The most effective process architecture for a spreadsheet application, for example, will be different from the most effective process architecture for developing a complex command-control-communications system. Similarly, project schedule, budget, personnel, reliability, or portability constraints will strongly condition the most effective choice and sequencing of major process activities. Thus no fixed predetermined process is likely to be adequate for a wide range of software activities, or even for a single software activity over a period of time. For these reasons we believe it is imperative that software processes be programmable in order to assure the needed flexibility, and modifiability. Moreover, just as the programming of a software product is more effective when preceded by product requirements analysis, architecture definition, and design activities, so will be the programming of software processes.

Many observers believe that progress towards understanding what constitutes adequate products and effective processes can only follow from experimentation with alternatives. We believe that the best way to facilitate such experimentation is to enable users to easily yet rigorously 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. From our perspective it is clear that this is tantamount to the creation of environments in which the variant part — i.e., the product specifications, process descriptions and set of operators — is specifiable by the user, and in which the environment exploits this specification to fashion user support, utilizing components from the fixed part.

Process Programming Languages

In this section we discuss some key details of the Arcadia project's approach to enabling users to employ process programs to flexibly specify how they wish to have machine resources applied to the support of their activities. Figure 2 shows the relationship of process programs to the projects they support, to the language(s) in which they are written, and to the operations and operands that implement

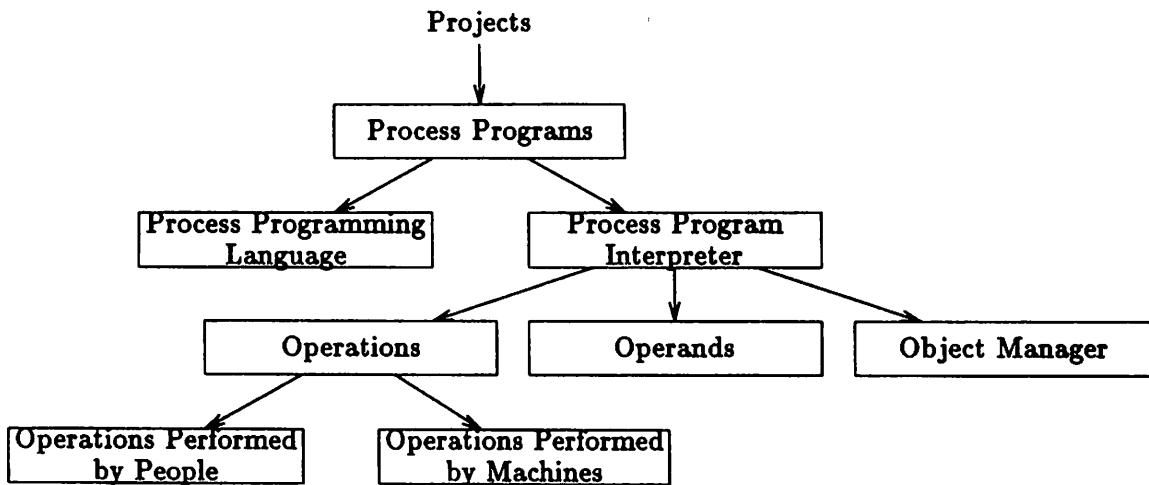

Figure 2: The relationship of process programs to the projects they support and to the mechanisms that implement them.

them. We see that projects are to be directly supported by specifications of how they are to be carried out, where these specifications are to be captured by process programs, written in a process programming language and carried out by a process program interpreter. The interpreter is responsible for translating the specifications embodied in the process program into operations on operands. As indicated in Figure 2, some of the operators upon which the interpreter relies are executed by computing devices, but others are executed by humans. We will now summarize the significant research activities which have been carried out in these areas.

It is important to observe that while progress in specifying software processes clearly depends upon the existence of a process programming language in which they are to be written, that the specification of such a language must also depend upon the accumulation of experience in writing process programs. Thus, in an important sense, there is a "research deadlock" between the need for a process programming language and the need to write effective process programs. Our approach to breaking this deadlock has been to engage in a variety of prototyping projects. These projects include the development of a prototype process programming language, called *Appl/A*, and the writing of prototype process programs making use of *Appl/A*.

Appl/A is a language designed as a superset of Ada which enables the definition of relations among software objects. Arcadia believes that software products should be viewed as large aggregates of software objects which are interconnected by very large numbers of very complex relations. As contemporary languages do not support the direct definition and manipulation of such relations, we believe that these languages lack a key component needed by any successful process programming language. 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 must be maintained. As such, we believe that *Appl/A* provides a very basic capability that is needed in any process programming language. *Appl/A* is described in more detail in [23]. It is also described in the context of the wider Arcadia object management effort in Section 3.

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 *Bopeep* (But One Prototype End-to-End Process) project. In *Bopeep* we are developing process programs covering key software development and maintenance phases such as requirements specification, design, and maintenance. We have written a series of requirements specification process programs in the *Rebus* (REquirements BUilding System) project. All *Rebus* process programs treat requirements development as an activity aimed at creating a DAG 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 to then put values into the appropriate fields. The requirements analyst must also indicate any relations among the various requirements elements and between requirements elements and other software objects. *Appl/A* has proven to be quite useful in supporting some of these capabilities. The *Rebus* project has also indicated that there are other important process programming features which cannot be supplied by *Appl/A*, and indeed by any language built atop Ada. These features include a type hierarchy, and various dynamic features, such as dynamic type creation, and dynamic process creation.

We are studying design development through the *Debus* (DEsign BUilding System) project. In *Debus* we are writing process programs that attempt to codify such design methodologies as Object Oriented Design [6], and Parnas' Rational Design Methodology [36]. *Appl/A* is being used in *Debus* and we are finding that many of the structures and mechanisms used in *Rebus* are also helpful in developing *Debus* process programs. We are also writing maintenance process programs as part of the *Orbit* project, and will shortly begin writing Testplan Building Systems as part of the *Tebus* project. All of this work is leading to the creation of a library of prototype process programs which are helping us to understand these processes. It also helps us to evaluate the features incorporated into *Appl/A* as well as other process programming language features which are needed.

It should 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 defy simple algorithmic description and suggest that the declarative paradigm is much more appropriate. Design creation is an example of such an activity. In design creation the goal is to create a design specification. Often (e.g., in the case of the Software Cost Reduction methodology [36]), 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

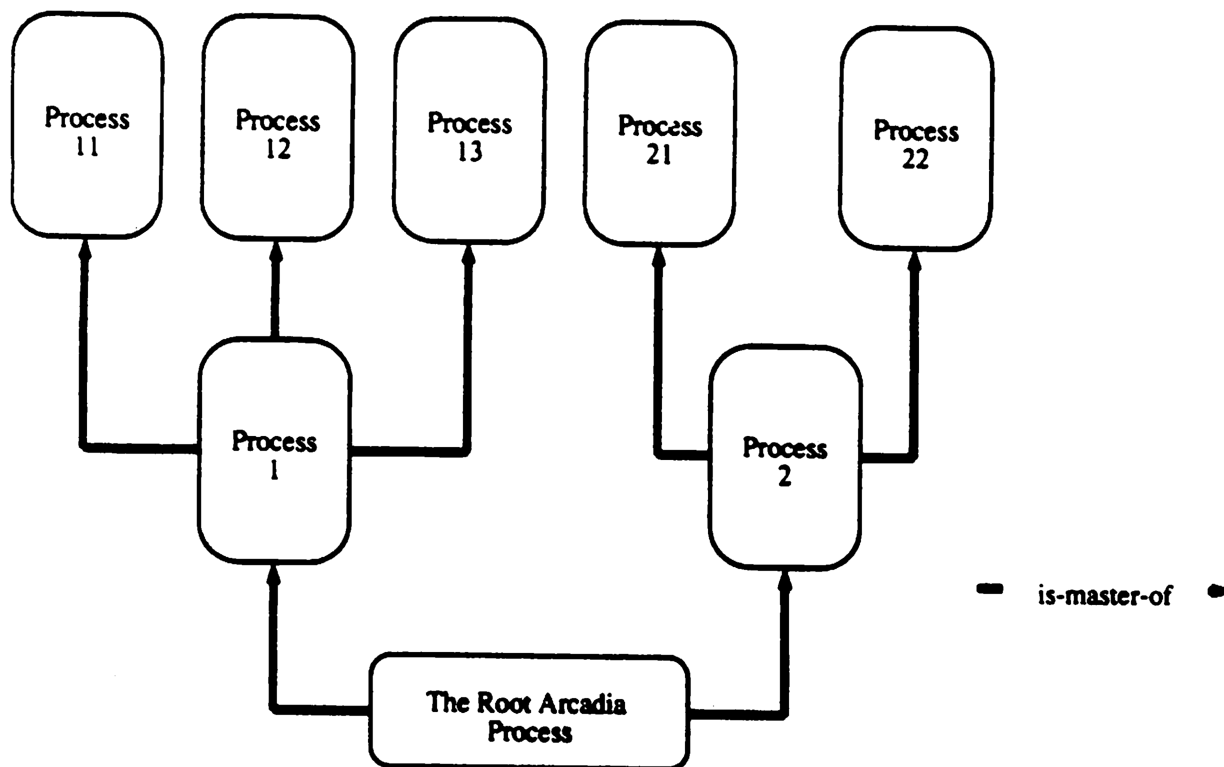

Figure 3: Arcadia as an Interacting Hierarchy of Processes

ability to mix procedural coding with rule coding has proven useful.

From this early work with the *Bopeep* prototypes we have learned that Arcadia must be built as an interacting hierarchy of processes, as indicated in Figure 3. Each of the indicated processes should be thought of as having been written as a process program, although some may have been intended to be executed by humans and others by computer hardware. The processes in the indicated hierarchy are related by the relation *is–master–of*. A process is created to accomplish some software development or maintenance activity; a process may create additional processes to aid in achieving this objective. The *is–master–of* relation denotes this creation relationship and serves as the basis for defining other relations, particularly those expressing communication.

It is important to stress that this relation is only one of the many different sorts

of relations among subprocesses which an Arcadia process will require. In particular it is important to distinguish between the Arcadia process architecture and the structure of such more primitive systems as *ISTAR* [17], in which only a strict hierarchy of processes is possible. In *ISTAR* processes rely upon subprocesses in order to get their work done. Furthermore, all communication between processes must be between parent and child processes. This has proven to be a serious weakness. In Arcadia, this is only one of the possible interprocess relations. In fact the exact relations between processes in Arcadia are to be programmed by process programmers as necessary. Thus, the *is–master–of* relation shown corresponds to a procedure invocation relation, and as such is clearly only one of many that might be programmed according to the needs of the process.

For example, our early work indicates that an important aspect of Arcadia software processes is that they often create other processes that are executed later on. For example, test planning is a process whose goal is the creation of the description of a process that is to be executed at some future point in the execution of the software development process. 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 an operand. This passive/active nature of some software processes points to the desirability of a language in which code and data are freely interchangeable, as in Lisp.

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. At present we have come to some important basic understandings through the development of prototype languages and processes.

3 Object Management

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. We believe, however, that a much richer set of capabilities for controlling object creation, access, and organization is essential to an 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. As mentioned in Section 2, an object can move between categories during its lifetime.

Thus, the object management system will provide the underlying mechanism upon which the data management capabilities of a process programming language and a process program interpreter can be constructed. 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 equally 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 revealed that those properties are determined by how the object management system addresses issues in four important areas:

- type systems;
- relationship systems;
- object persistence; and
- concurrent and distributed object management.

Each poses interesting problems. The capabilities sought in 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 [19] 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. The *Odin* subsystem of *Toolpack* [14] improved on this simple view by using file name extensions as a weak form of typing mechanism for files (going beyond the capabilities of *Make*). 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 [27] used the term "object" for referring to the files containing product components, but did not treat the objects as instances of abstract types. The Common APSE Interface Set (CAIS) [8] defines a system model with three kinds of nodes—file, structural, and process—but does not treat those nodes as typed objects. 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 [50,10,68], while encouraging, is still in its primitive stages and far from providing the flexibility and power needed for object management in a software development environment [4]. Recent work on rich type systems, particularly in the context of object-oriented languages [30], 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, one major object management research issue is: What kind of type system is needed to describe the objects populating a software development environment? 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 between 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 revision control systems [58], automated system building tools [19], call graph analyzers, and work activity management systems [20]. 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* [57]), *Odin*'s derivation graphs, and the system models of *Cedar* represent important starting points. The concept of configuration threads found in *DSEE* [28] and the relationship capabilities for module interconnection languages provided by *Intercol* [59] and *PIC* [64] are additional examples of partial treatments specialized to one class of relationships.

Thus, another important object management research question is: What are suitable primitives and constructors for defining the relationships needed in environments? It is not clear whether the diverse relationships needed in software development 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.

Object persistence. The object manager must be able to preserve the components of software products and the constituents of software environments for ar-

bitrary 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 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 [2,15,33], although implemented prototypes have been very restrictive about the supported types [2] or the underlying program model [15].

Thus, an important issue that must be addressed by the object manager is: How should persistence be provided for arbitrarily complex, typed objects? 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.

Concurrent and distributed object management. 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] [24] [29] and file system and database research [22] [60] [44]. 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 (e.g., update a source program) 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, the major questions facing object management include: What are appropriate constructs for expressing distribution and concurrency constraints and what underlying mechanisms must be provided to support these constraints? 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?

Object management within an Arcadia environment. 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.

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, change, and distribution.

It is clear that some new solutions are required to satisfy the special needs of software development environments. To sharpen our understanding of those needs, we are examining process programs for a wide variety of activities, examining a wide variety of tools that would make use of the object management system, and reflecting on our experience building *Odin*, *Keystone* [13], and *Graphite* [12], which

can be viewed as primitive object management systems. We are also developing formal models, as we have previously done for module interface relationships [65], 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 [40] 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 they relate to 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 translator 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* [63] are two prototypes 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 a separate 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 [25] 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 [45,4,46,10] 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 [31]. 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 Interface with the Human User

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. We begin by discussing a set of goals, i.e., characteristics of good human interfaces, and a set of problems that must be addressed to achieve those goals in the context of a broad, extensible environment. Then we discuss the approach that has been taken in *Chiron*, the user interface subsystem of an Arcadia environment. A prototype implementation of *Chiron* has been distributed within the Arcadia consortium; additional detail on the design and implementation may be found in [66].

Characteristics of good interfaces. Broad consensus exists on the qualities which 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, mostly by imposing informal standards (like the *Macintosh* user interface guidelines [26]) and providing a toolkit of reusable components (scrollbars, menus, and the like).

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 represen-

tations of state. Pioneering work in the application of graphics to programming and software engineering include the *Incense* debugging system [32], the *Balsa* algorithm animation system [7], and the *Pecan* programming environment [39].

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* [39] and the *Pi* debugger [11] hint at the richer interaction possible when each tool may coordinate several threads of control.

Outstanding problems. Uniformity becomes both more important and harder to maintain as the scope of an environment grows. A large, extensible environment will contain tools contributed by a diverse community of developers and users. 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 [51], 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 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 becomes apparent when one notes that other tools, as well as human users, may use a tool component. A good human interface is generally not a good tool interface. An all-purpose interface, like the UNIX system character streams, is unlikely to be satisfactory in either role. Thus, in current systems based on the UNIX system, the set of tool-usable tools is quite disjoint from the set of interactive tools. It is difficult, for instance, to use a screen-oriented editor or a spreadsheet program as part of a pipe or shell script.

Techniques and approaches. User interface management systems (UIMS) is an active area of research, outside the context of software environments research as well as within it. The *Chiron* subsystem of an Arcadia environment adapts and extends some key ideas from current UIMS research to address the particular demands of large, extensible software environments. The following paragraphs discuss our ap-

proaches 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 an Arcadia environment, tool components which manipulate model objects can be largely freed of concern with view objects. The *Chiron* subsystem is used to build encapsulated tool components which 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 an Arcadia 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 [32] in the *Incense* symbolic debugging system. *Loops* [48,49] 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 4).

The essential characteristic of annotation as a mechanism for binding artists

[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.

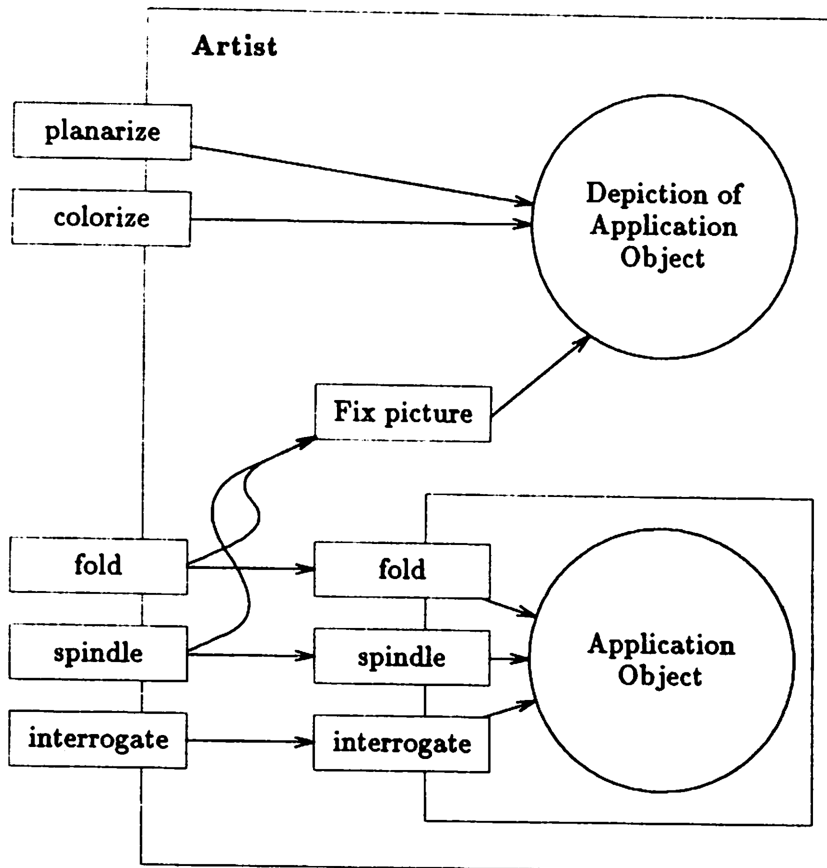

Figure 4: An artist is logically "wrapped around" an abstract data type.

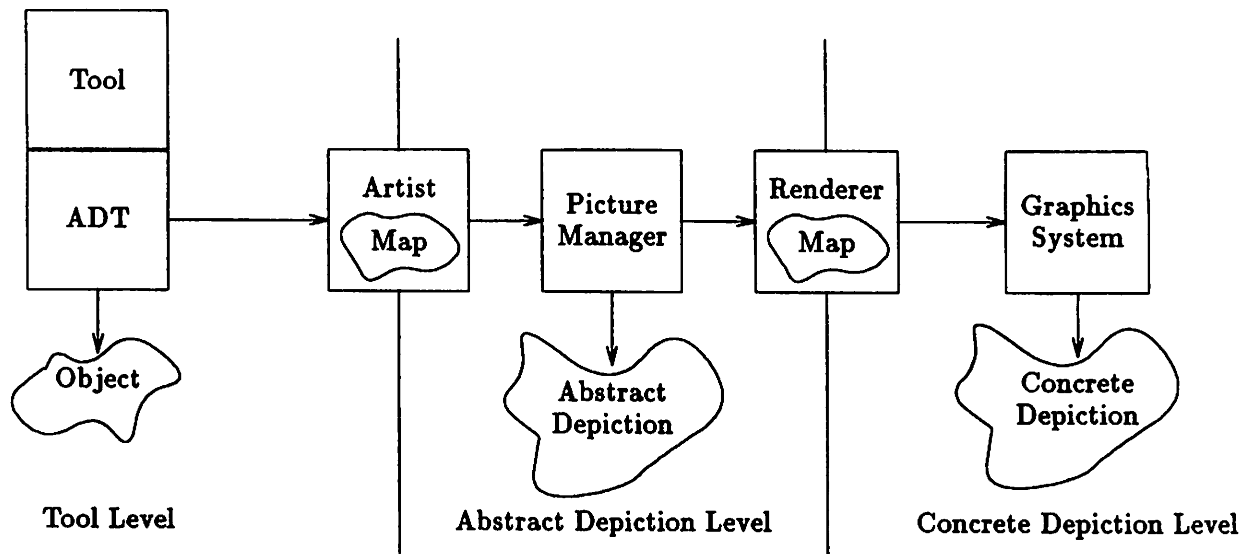

Figure 5: Artists manipulate an abstract description of the display, and a rendering agent maps the abstract depiction to bitmap images.

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 5).

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. Window systems provide input correlation down to the window level, but not within windows. This is sufficient for menus and scrollbars, which can be designed so that each choice lies in its own window, but not

sufficient for general diagrammatic depictions of software objects. An abstract depiction level managed by the environment can perform input correlation to the level of individual polygons, lines, and so forth.

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 scope and extensibility in a software environment. The most that can be done by the user interface subsystem 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 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 an Arcadia 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 Some Details of the Architecture of Arcadia

Having described the underlying concepts and the basic technical approaches being taken in the Arcadia environment project, we now briefly consider some particular design decisions. One of our primary objectives in the design has been to maintain fidelity to the conceptual approach while designing to accommodate anticipated change.

As was discussed in Section 2 and shown in Figure 3, an Arcadia environment as a dynamic entity can best be viewed as an interacting hierarchy of *Arcadia processes.* In addition to keeping the notion of "process" at the forefront of Arcadia and enabling change to the processes that the environment supports, this design enables encapsulation of some environment components that will remain in flux while experience is obtained and alternatives explored. For example, the notion of Arcadia process is defined such that each may employ its own process programming language. Thus to "start-up" an Arcadia environment one only requires a small language adequate for programming the root process. Subsidiary processes may employ more powerful and experimental languages. (Of course communication of objects between processes is dependent on the processes involved sharing a common type system.)

Encapsulation of other aspects of Arcadia that are subject to change can be seen by taking a more detailed look at this notion of Arcadia process. Figure 6 shows a more robust version of this entity, while Figure 7 indicates how this notion is related to other, simpler, notions of process. In addition to supporting alternative process programming languages, the following entities, for example, are also encapsulated, supporting their anticipated change:

- the object management system's storage manager: within the OM servers
- resources of the underlying machines: within the resource manager
- protocols for inter-process communication: within the communications manager
- *Chiron*'s abstract depiction language: within the UIMS server.

Change to a variety of other components is similarly anticipated, but space does not permit their consideration here. Note that many of these components are logically parallel activities.

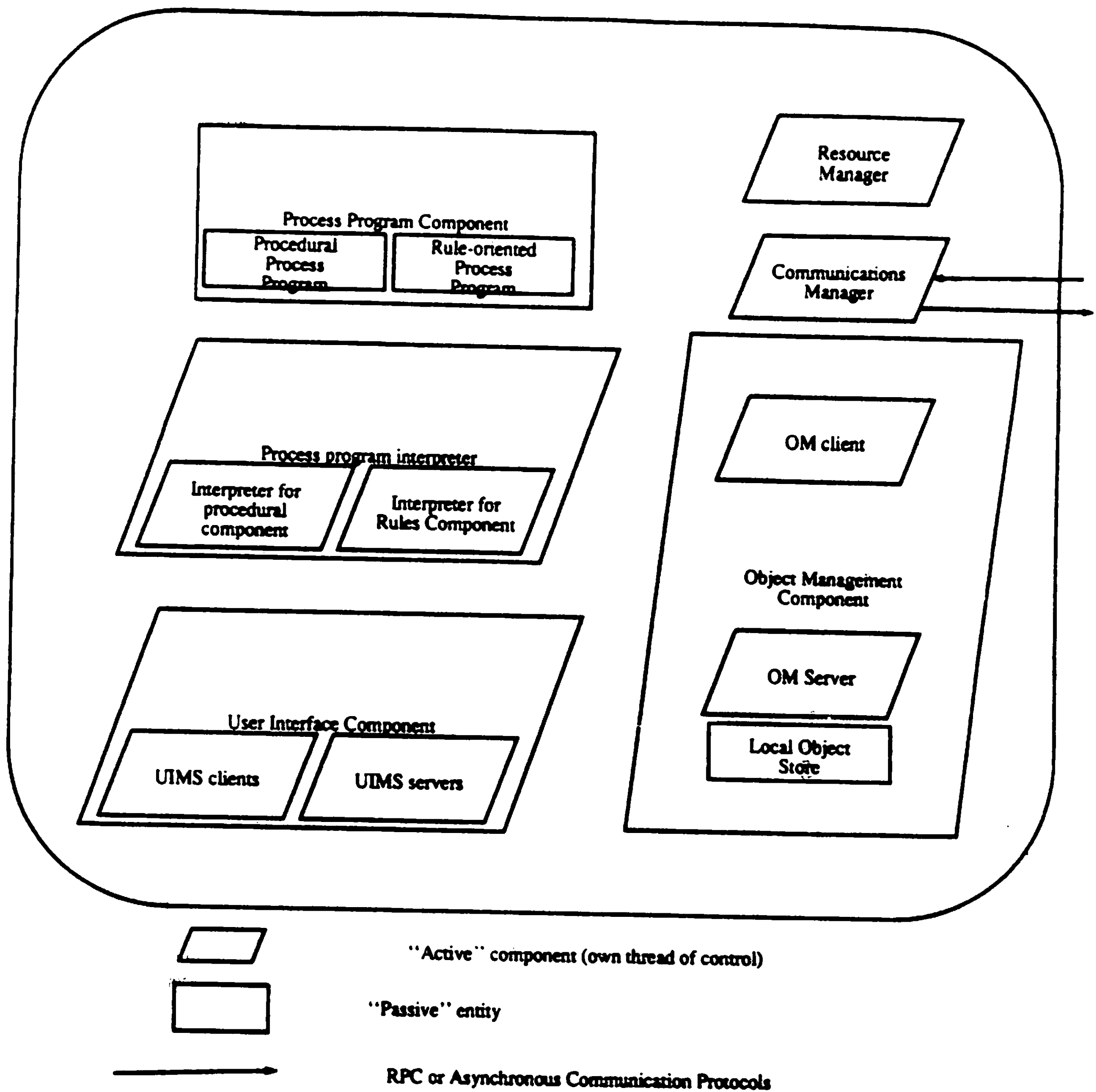

Figure 6: The Anatomy of an Arcadia Process

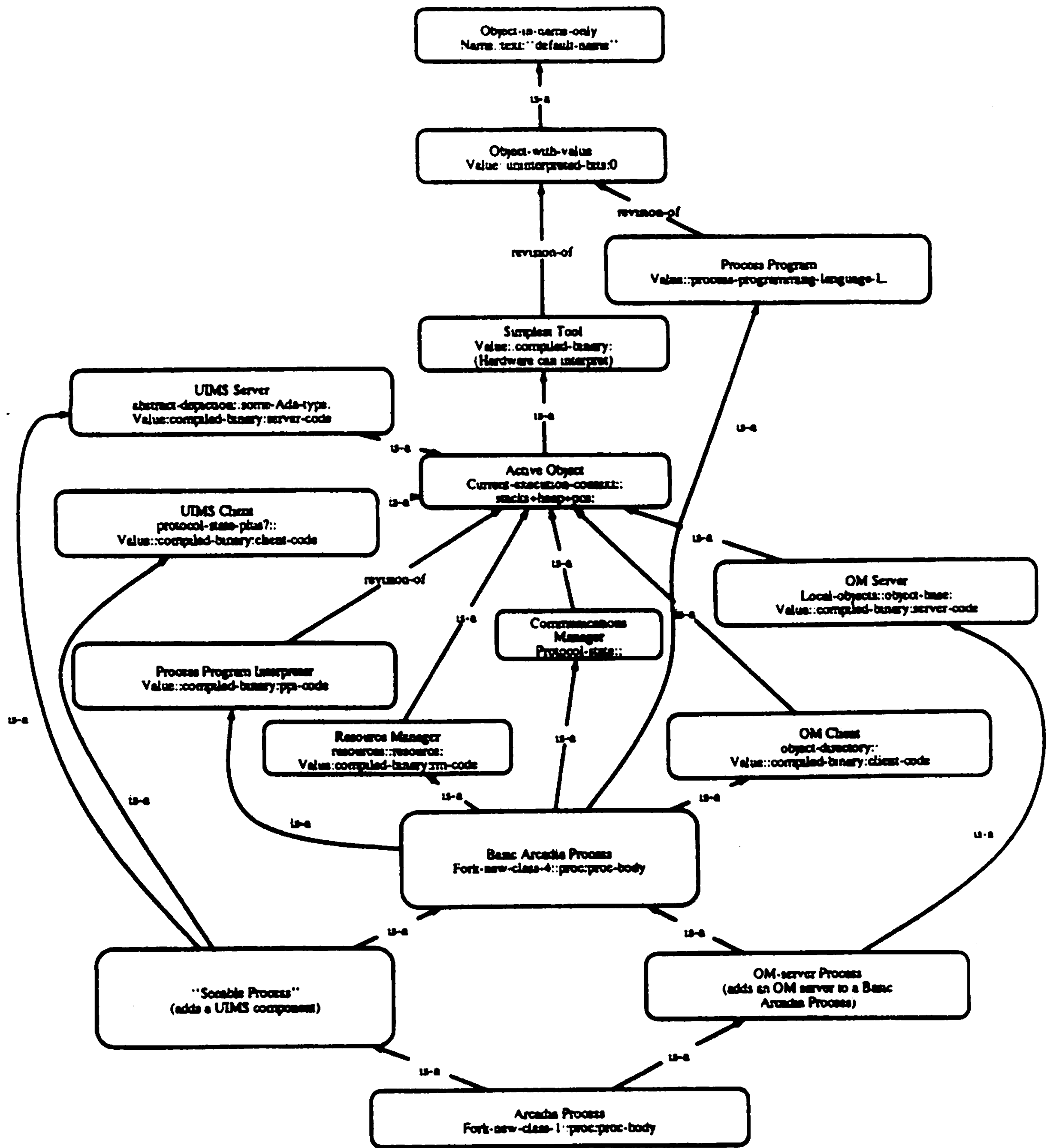

Figure 7: Rounded rectangles denote classes of objects. Each class contains some number of slots. Each slot has a name as indicated, and may have a type and a value. The class of objects at the tail of an *is-a* arc inherit all the slots (names, types, and values, if any) present in the class at the head of the arc. New slots can be added to the class at the tail of the arc. Inheritance is by aggregation only. *revision-of* links are an orthogonal notion to *is-a*. The value of a slot-type field or a slot-value field is altered from that shown in the object at the head of the arc to that shown in the tail of the arc.

This multiple-Arcadia process model, with multiple threads within each process, maps cleanly to some modern operating systems, notably Mach [38]. Details of this mapping, as well as further consideration of the design, can be found in [52]. It remains to be seen whether the emerging design and its realization in Mach will exhibit adequate performance.

6 Summary and Conclusion

The current flurry of activity in environments and in software process specification is exciting. A proper focus for environments — supporting the user's multiple, complex activities — is being reemphasized at a time when some pertinent sub-technologies are maturing. This paper has presented a view that is useful in categorizing and assessing developments in environments, and has attempted to separate some key concerns. In so doing, a number of emerging principles and important open problems have been identified and some promising research directions described.

One key distinction is between an environment's fixed infrastructure and its variant part. As part of the infrastructure, a user interface management system provides communication between humans and executing software processes. These processes are described in a formal process programming language and are interpreted by a process program interpreter. Mundane, automatable activities are handled directly; creative activities are performed by creative agents: people. A key component of the automated interpreter is an object management subsystem, whose typing system, relationship system, persistence scheme, and facilities for distributed and concurrent object management, support the constructs of the process programming language. Having process programming as a key part of the concept makes the environment an active agent, rather than a purely reactive one.

In our estimation, progress on the various fronts of environment research is now tied to realistic prototype development, empirical evaluation, and technology transfer. Prototypes are needed to validate concepts, generate feedback, and provide demonstrations that new environment technologies are useful to large development teams tackling large development activities. To be fully convincing, and to generate as much insight as possible, realistic prototypes must be subjected to well-designed empirical evaluation. Carefully planned technology transfer activities are then needed to ensure that the sought-after benefits are fully realized.

The Arcadia consortium has been formed to do research in environment architectures. We are attempting to make major strides in the development of the fundamental technologies, develop prototypes, conduct careful empirical studies, and move the technology to industrial practice. Joint funding to support the activities of the consortium began in August 1987.

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, 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, January 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 Transactions on Software Engineering*, SE-3(1):49–60, February 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] G. Booch. Object-oriented development. *IEEE Transactions on Software Engineering*, SE-12(2):211–221, February 1986.

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

[8] *Military Standard Common APSE Interface Set (CAIS), Proposed MIL-STD-CAIS*. Department of Defense, January 1985.

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

[10] 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 DMBS: A Preliminary Report*. Technical Report CS-644, Computer Science Department, University of Wisconsin - Madison, Madison, May 1986.

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

[12] 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, April 1986.

[13] 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.

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

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

[16] 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.

[17] M. Dowson. ISTAR — an integrated project support environment. *ACM SIGPLAN Notices*, 2(1), January 1987. Proceedings of the Second ACM SIGPLAN/SIGSOFT Software Engineering Symposium on Practical Development Support Environments.

[18] 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 Transactions on Software Engineering*, SE-12(2):293–311, February 1986.

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

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

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

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

[23] 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.

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

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

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

[27] B. W. Lampson and E. E. Schmidt. Organising 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.

[28] 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).

[29] 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.

[30] N. Meyrowitz, editor. *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, ACM SIGPLAN, Portland, Oregon, September 1986. (Appeared as SIGPLAN Notices, 21(11), November 1986.).

[31] 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, April 1988.

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

[33] 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.

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

[35] L. J. Osterweil. Software processes are software too. In *Proceedings of the Ninth International Conference on Software Engineering*, pages 2–13, Monterey, CA, March 30 – April 2, 1987.

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

[37] 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.

[38] R. F. Rashid. From RIG to Accent to Mach: the evolution of a network operating system. In *Proceedings of the Fall Joint Computer Conference*, pages 1128–1137, Nov. 1986.

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

[40] B. Rosenblatt, J. Wileden, and A. Wolf. *Requirements for an Object Management Type System in Software Development Environments*. Technical Report 88–37, COINS, University of Massachusetts, Amherst, MA, April 1988.

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

[42] 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, December 1986.

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

[44] 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. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 35–50, 1985.

[45] 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.

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

[47] 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).

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

[49] 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.

[50] 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.

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

[52] R. N. Taylor. *The Architecture and Implementation of Arcadia Environments: The Top Levels*. Arcadia document UCI-88-03, Department of Computer Science, University of California, Irvine, California, 1988.

[53] R. N. Taylor, D. A. Baker, F. C. Bels, B. W. Boehm, L. Clarke, D. A. Fisher, L. Osterweil, R. W. Selby, J. C. Wileden, A. L. Wolf, and M. Young. *Next Generation Software Environments: Principles, Problems, and Research Directions*. Technical Report 87-16, Department of Information and Computer Science, University of California, Irvine, July 1987. Also issued as Arcadia Document Number UCI-87-10, University of Colorado Technical Report Number CU-CS-370-87, University of Massachusetts, Amherst Technical Report Number 87-63, and Incremental Systems Corporation Technical Report Number 87-7-1.

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

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

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

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

[58] 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.

[59] 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.

[60] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS distributed operating system. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 49–70, 1983.

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

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

[63] J. Wileden, A. Wolf, C. Fisher, and P. Tarr. *PGraphite: An Experiment in Persistent Typed Object Management.* Technical Report 88–35, COINS, University of Massachusetts, Amherst, MA, April 1988.

[64] 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.* (to appear).

[65] A. Wolf, L. Clarke, and J. Wileden. A model of visibility control. *IEEE Transactions on Software Engineering,* SE-14(4), 1988.

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

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

[68] 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.