Arcadia: A Software Development Environment Research Project

Richard N. Taylor, Lori Clarke*, Leon J. Osterweil**,
Jack C. Wileden*, Michal Young

Technical Report #85-31

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

November 25, 1985

[1] This work was supported in part by DARPA under contract (ONR) N00039-83-C-0567, by National Science Foundation grants DCR-8451421 and DCR-8502558, the Alcoa Foundation, and Hughes Aircraft (MICRO program).

[2] This work was supported in part by the following grants: Rome Air Development Corporation, No.SCEEE-PDP/85-0037; National Science Foundation, grant No.DCR-8404217 and No.DCR-8408143; and Control Data Corporation, No.84M103.

Contents

Abstract

The research objectives of the Arcadia project are two-fold: discovery and development of *environment architecture principles* and creation of novel *software development tools*, particularly powerful analysis tools, which will function within an environment built upon these architectural principles.

Work in the architecture area is concerned with providing the framework to support *integration* while also supporting the often conflicting goal of *extensibility*. Thus, this area of research is directed toward achieving *external* integration by providing a consistent, uniform user interface, while still admitting customization and addition of new tools and interface functions. In an effort to also attain *internal* integration, research is aimed at developing mechanisms for structuring and managing the tools and data objects that populate a software development environment, while facilitating the insertion of new kinds of tools and new classes of objects.

The unifying theme of work in the tools area is support for effective *analysis* at every stage of a software development project. Research is directed toward tools suitable for analyzing pre-implementation descriptions of software, software itself, and towards the production of testing and debugging tools. In many cases, these tools are specifically tailored for applicability to concurrent, distributed, or real-time software systems.

The initial focus of Arcadia research is on creating a prototype environment, embodying the architectural principles, which supports Ada[1] software development. This prototype environment is itself being developed in Ada.

Arcadia is being developed by a consortium of researchers from the University of California at Irvine, the University of Colorado at Boulder, the University of Massachusetts at Amherst, TRW, Incremental Systems Corporation, and The Aerospace Corporation. This paper delineates the research objectives and describes the approaches being taken, the organization of the research endeavor, and current status of the work.

[1] Ada is a trademark of the U.S. Department of Defense.

1 Introduction

Early research in software development environments was seldom referred to in those terms. The primary focus of environment-related research was usually on specific tools, programming languages, databases, or operating system concepts. Interlisp, for example, focused heavily on developing functions that helped in the production of Lisp programs [TM81]. Only after many of these functions were developed was it recognized that such a suite of capabilities constituted an *environment* in which code production could take place. Unix[2] is fundamentally an operating system, but its design encourages and even enforces a particular way of binding tool capabilities together; it is arguably an environment, though very different in structure from Interlisp.

Subsequent developments have aided in refining the notion of a software development environment. For instance, it is now recognized that a development environment must contain much more than just programming tools: design and analysis methods, the tools to support them, and management techniques must all be carefully integrated to form an effective environment.

More recent research has delineated the notion of an environment *architecture*. This term is used to denote the set of rules and support infrastructure which characterize, bind together, and enable utilization of the software development support tools existing within an environment. Object managers, user-interface tools, and tool activation managers may all be elements of an environment architecture.

Environments and their underlying architectures can be classified according to the qualities they possess. The qualities dominating the development of Arcadia are extensibility and integration. Extensibility refers to the ease with which it is possible to add new capabilities to an environment. Incorporation of new capabilities in an extensible environment will not involve changes to the fundamental architecture, unless a basic property of the new capability is in direct conflict with the fundamental principles of the environment. Note that "extensible" as used here is not equivalent to "open", as that term is often interpreted: open often designates the ability to incorporate foreign programs *without change*. Integration in the context of Arcadia denotes several properties: (1) consistent user interfaces, (2) easy context switching with restoration of state, and (3) efficient communication between tools through the sharing of data structures. The first of these contributes to external integration, while the latter two are aspects of internal integration. Secondary objectives for Arcadia are to effectively support creation and evolution of software by a team of developers and to support development

[2] Unix is a trademark of AT&T.

that occurs over a network of machines.

In addition to environment architecture issues, the Arcadia project is also investigating specific techniques to support all aspects of the software development process. Specifically, the project is working on creating tools, and on defining the objects that they manipulate, in three broad categories. First are the *basic components* of a software development environment. Emphases here include the development of an internal representation for programs, called IRIS, that is suitable not only for compilation purposes but also for interpretation, analysis and program transformation. Second are *tool-building tools.* This class of tools is of particular value to software developers who, like the members of the Arcadia consortium, are constructing software development environments. Since they include such things as lexer and parser generator tools, however, they are also of potential interest to many other software developers. Finally, there are *analysis tools*. These include testing and debugging tools as well as design analysis tools and other tools applicable in pre-implementation stages of software development. An important goal in this area is addressing the special concerns related to embedded systems software, such as distributed system design analysis and host-target debugging.

The Arcadia project is, therefore, a research activity exploring issues in two domains. An environment architecture is being investigated and created, as are a number of software development support tools. These tools and the objects they manipulate are the components acted upon by the environment architecture. Together, the architecture and the tools constitute an experimental Ada environment, which is being implemented in Ada. The environment is being designed to assist in a wide range of development activities and to support a variety of models of the software process; our focus is on fundamental issues concerning automated support for software development. Subsequent sections of this paper examine these aspects separately. Some aspects of our development strategy are also briefly described.

2 Aspects of the Architecture

Environment architectures are similar to operating systems in that they deal principally with the management of processes, objects, and interfaces. Moreover environment architectures provide a level of virtual machine above that of conventional operating systems. Specific environments provide levels above that, since they provide powerful tools. Our working premise has been that any program or interface definition which determines how tools interact, how they must be built, or how objects behave in the environment, must be a part of the environment architecture.

Environment architectures are since
they constitute a level of virtual uated
with respect to traditional progra bility,
uniformity, and completeness. Ea they
apply to the three fundamental ent n the
domain of environments extensibil and
modification of the entities, such as ifor-
mity is particularly an issue with re rd veen
the user and the environment. Co ar-
chitectures includes assessment of th hat
architecture must be methodology-sp ure
may be forced by that architecture t del,
programming methodology, or testin ay
not be supportable due to a lack of ar-
chitecture. For example, the archite for
concurrency in the environment and i

Lastly, environment architectures il-
ities with regard to the level of abstra g
with processes, objects, and interfaces.

The following subsections detail o approach in these categories. Our end goal is the formulation of a clean conceptual architecture model upon which useful systems can be built. We begin with a capsule sketch of the environment.

The User's View and Tool Structure The toolset of Arcadia will be made up of very small, modular tool fragments (section 2.2); any substantial task will involve potentially complex interaction between fragments. It is imperative that users be shielded from this complexity. This will be achieved partially by allowing users to simply describe an object which can be derived from existing objects in the data base, and leaving it to the tool and object managers to plan and coordinate the application of tool fragments to produce the desired object. Users must also be shielded from the complexity of tool fragment interactions during highly interactive tasks, such as editing. The users of interactive tools in Arcadia will not be faced with the complexity of dozens of tool fragments interacting in potentially complex ways, but rather will deal with a high-level user interface. They will see a set of (graphically) depicted objects, upon which certain operations are possible. Different operations on the same object may be carried out by different tool fragments, and a single operation may involve the coordination of several tool fragments, but the users need not be aware of this.

What, then, is a "tool" in Arcadia? In a conventional programming environment, a tool is a collection of capabilities which execute together in an integrated manner, and which are generally developed and maintained together as a unit. That is, the usual notion of a tool is more or less equivalent to the notion of a program. In Arcadia, a tool is just a collection of tool fragments temporarily allied for the purpose of providing the desired functionality to complete some activity. These fragments may be loosely connected as in a Unix pipeline, but often they may be more closely coordinated. In particular, an interactive tool may consist of several fragments cooperating in the modification of a single object or collection of objects, including the display.

2.1 Object Management

A central premise of this project is that Arcadia should appear to the user to be a system for creating and managing numerous and diverse software objects such as source and object code, design elements, test data, and graph representations which arise during the process of developing software. Accordingly a view of software development and maintenance which Arcadia attempts to support is that software is a large and intricate collection of long lived (persistent) objects and aggregates of information, and that the process of developing software successfully is dependent upon (and perhaps tantamount to) the process of successfully creating, organizing, augmenting, and exploiting these objects and aggregates. Accordingly, Arcadia users will be encouraged to think of their work in terms of the need to create, aggregate, alter, and view the objects. (A trivial example: rather than requesting the execution of a program, the user will request display of the output of a program as applied to a specified set of data.) Thus the object management function is one of the most important in Arcadia. This philosophical approach to environment organization and the architecture of our object manager are both based upon similar approaches taken in the Toolpack and Odin projects [Ost83] [CO85].

2.1.1 Objects in Arcadia

Arcadia objects may range widely in size and character. Objects may be small—such as tool option specifications and test data, or may be large—such as entire executable programs or the results of executing large test suites. Objects may be as diverse as text, object code, test data, symbol tables or bitmap display frames.

To help in the management of such diversity, each object is annotated and each object may be related to a potentially wide range of other objects.

The most important annotation which Arcadia objects carry is type. In Arcadia source text is one type, object code is another, test data, parse trees, attribute tables, and so forth are yet others. From this perspective the Arcadia objects can be viewed as instances of abstract data types. The types are defined within Arcadia in terms of clusters of accessing functions. Arcadia will offer facilities for creating and integrating new accessing function clusters and thereby augmenting its collection of types. This is perhaps the most important sense in which Arcadia is extensible.

Newly created objects are said to have been derived from their predecessors, in which case Arcadia notes this derivation relation and uses it to store the new objects, placing them as descendents of their predecessors in a structure known as the Object Derivation Graph (ODG). Each object can be viewed as the root of two subgraphs of the ODG which are trees. An object's Ancestor Derivation Tree indicates which ancestor objects were used to derive the object; it's Descendent Deriviation Tree indicates which objects depend upon the object for their derivation. Objects which are created either by importation from outside Arcadia or by user creation through a tool such as an editor will initially have no ancestor or descendent subtrees. As tool functions are applied to these objects and, in turn, to objects derived from them, the Arcadia object manager will automatically organize all of these objects as descendents. As any Arcadia object may be persistent, these object derivation trees are potentially arbitrarily deep.

The notion of organizing objects into trees in which descendants have been created by derivation from ancestors has already been exploited in certain version control systems such as RCS [Tic82]. In such systems, all derivations are created by the action of a single tool, generally a text editor. It is assumed that there is a single root version, and that this version has one or more successors, each of which can be derived from the root version by the action of the deriving tool (e.g. the text editor). These descendents can then be further transformed by successive actions of the deriving tool to create further subtrees of versions. By moving up and down this tree, a software maintenance team can make changes having either greater or lesser impact, as their needs and wishes may require.

Arcadia will effect a similar derivational structuring of the object store, but in Arcadia this structure is developed, not by successive derivations by means of a single tool, but by applications of any Arcadia tool. Thus a user may use Arcadia to create a Descendent Derivation Tree for versions of source code as in RCS, or may create a complex tree in which objects of various types, the results of different sequences of tool applications, are all stored in a single Descendent Derivation Tree.

Arcadia will also incorporate algorithms for determining when derived objects at lower levels of a Descendent Derivation Tree have become obsolete because of alterations or deletions of objects higher up in the tree. Whenever an object at a higher level of the tree is changed, the Arcadia object manager will recognize that all descendents of that object must be viewed with suspicion. Arcadia will not take immediate steps to rederive these objects, however. Instead Arcadia will employ a demand rederivation strategy under which new versions of an object are not created until they have been requested either directly or indirectly by the user. At that time, all the objects in the requested object's Ancestor Derivation Tree are examined. If it was in fact derived from ancestor objects which have subsequently been altered, the Arcadia object manager will begin the process of recomputing it from the current version of the ancestor objects. Objects between the altered ancestors and the desired object are rederived. They will also be compared to their previous versions. If at any point the rederived objects match their previous versions, the rederivation process will stop and the equivalence of old and new versions of objects lower in the tree will be noted.

Hierarchy will also be used to organize the Arcadia object store. Users may define arrays or structures of objects of heterogeneous types. These array and structure definitions are also Arcadia objects, and they may, in turn, be organized into higher level structures. This enables the Arcadia user to create structural hierarchies which are useful in, for example, modelling the inherent structure of systems being developed or maintained. In Arcadia these structures may overlap. This will make it possible for the user to create an object corresponding to a procedure library and then include it in several other objects, presumably corresponding to higher level functional sections of code. This inclusion is logical rather than physical and thus does not cause duplication of storage or difficulties in properly reflecting updates of shared objects.

It is important to point out that Arcadia software objects are not necessarily always created in response to direct user requests. Arcadia will support the notion of active tools as well as passive tools. Active tools will commence execution without direct invocation by users. They will carry out their activities by accessing and processing objects according to plans which will have been designed in advance, and perhaps by users other than the current user. These active tools will be invoked by Arcadia in accordance with such control mechanisms as timers or daemons whose job it will be to detect relevant changes in the object store.

Finally, it should be noted that Arcadia and all of its constituent tools, including the object manager, are themselves objects which are managed. Components of the environment architecture are in this sense indistinguishable from components of the software being developed by the environment user. This structure will aid in the ongoing development of Arcadia. Details of how tools may be considered as objects and what configuration management must be performed can be found in Section 2.2 on page 13.

2.1.2 Sharing Objects

Although Arcadia encourages the user to view software development and maintenance activities as the creation and management of a central store of persistent software objects, there is nothing in this view that requires this store to be physically centralized. In fact Arcadia will feature a distributed object store in which sharing of objects will be allowed and mediated. Arcadia is designed to support the cooperative activities of teams of software developers and maintainers, each of which is working on a workstation having significant self-contained disk storage facilities. Thus, each worker will have a separate store of persistent software objects. In addition, the workstations are assumed to be connected to each other by high speed data links. Thus each worker will also have potential access to software objects stored at other workstations.

The sharing of an object in Arcadia will be entered into carefully, monitored and supervised closely, and withdrawn from safely, all under the supervision of the Arcadia Federation Policy Manager. This software device is based upon the notion of a Software Federation [HM85]. In a federation, all objects initially belong to only one user, but shared access to these objects can be arranged as well. The sharing arrangement is negotiated by the party requesting the object and the party (or parties) currently controlling the object. The sharing arrangement may specify who is able to access the object, who is able to alter the object and the circumstances under which access to the object can be passed on to others. When one or more of the sharing parties wishes to withdraw from the sharing arrangement this too is negotiated under control of the Federation. Withdrawal may entail the making of copies, the creation of automatic updating relations among former sharing partners, or other devices for assuring an amicable and equitable parting of the ways.

The Federation mechanism assures that users retain autonomy when necessary, but are able to share when sharing is necessary or profitable. It also offers users such benefits of distribution as robustness in the face of hardware difficulties, and easy access to popular objects through the creation of multiple copies.

2.2 Tool Management And Invocation In Arcadia

Tool Structure Arcadia is intended to be a vehicle for furnishing extensive and growing tool capabilities. These capabilities will be furnished primarily by collections of small tools — tool fragments — rather than by a few, large, monolithic tools. For example, in Arcadia the task of prettyprinting could be carried out by a collection of tool fragments including a lexical analyzer, a parser, and a formatter—operating in concert. Some sophisticated prettyprinting functions would also require the invocation of a static semantic analyzer. An instrumented test execution could be carried out by a dozen or more tool fragments, operating at times in sequence and at times in parallel.

Creating larger tool capabilities out of smaller, more general tool fragments is a powerful and effective approach for a number of reasons. One of the main reasons is that, if the tool fragments are well chosen, they will prove to be usable as components of a variety of larger tools, thereby enabling the creation of these larger tools at lower cost. For example, both the prettyprinter and dynamic instrumentation tool just mentioned require lexical analysis and parsing in order to begin their work. Both tools incorporate these fragments, thereby saving the creators of these tools the effort of having to recreate these functional capabilities. In fact, the tool and object management functions of Arcadia will permit either the prettyprinter or dynamic instrumentation tool to re-use the intermediate results of the other.

Another benefit of a tool fragment architecture is that it encourages toolmakers to think in terms of good modular decomposition for their tools and good organization for the data and software objects which their tools use. Thus, we believe that the writer of a prettyprinter, for example, will create a better tool because correspondingly more time can be spent contemplating the nuances of the problem of prettyprinting, having been spared the problems of contemplating the nuances of parsing. In addition, because the writer of the prettyprinter is given access to the output of a proven parser, it is likely that the prettyprinter will be of better quality for its reliance on a more robust parser and its ability to exploit a richer range of information than would likely have been created from scratch. In short, we contend that the prettyprinter will be a better tool because it will be based conceptually upon such data and software constructs as lexical tokens, parse trees, and symbol tables and will be based physically upon major bodies of robust proven code.

Monolithic tools are not without their advantages, however, as experience with Arcturus [TS85] and Interlisp has shown. Accordingly there is nothing in the architecture of Arcadia which prohibits inclusion of monoliths. With suitable hardware support such tools may outperform more modular varieties. As the population of tools within Arcadia increases, "fly-offs" between tools of the two styles will be interesting and important studies.

Planning Tool Activations Although we have only mentioned two straightforward tool capabilities, Arcadia can incorporate a rich variety of tools, some of which may require complex configurations of tool fragments for their implementation. In many cases it will be possible to determine in advance just which tool fragments will have to be invoked to transform existing objects into other needed objects. There are some tools, however, for which this knowledge may not be completely available in advance. For example, a data flow analyzer (eg. [OF76]) must analyze the compilation units of a program in two passes, where the order of analysis during the second pass is computed during the first pass. In this case it is impossible to predetermine the exact order of invocation of tool fragments on the separate software objects.

Arcadia will support the synthesis of such tools as well. The mechanism needed here is a planning tool fragment whose job is to dynamically create tool invocation sequences that are tailored and adjusted in accordance with the current state of the object store and preprogrammed conditions. Planner tools in Arcadia will be able to create tool invocation sequences in which tools are scheduled for future invocation and in which software objects are taken and produced in unexpected or changed orders. In addition, planners will be able to schedule the invocation of other planners at projected future critical points.

Though it is anticipated that substantial tool activation will occur as the result of users requesting display of various objects, direct tool invocation will, of course, be supported. There is no gain in power here, it is merely a convenience and acquiescence to what seems natural in certain circumstances. Furthermore many specfic interactive tools are command driven and requiring that the environment be directed another way is incompatible with our goal of uniformity of interface. The topic of user interfaces is treated in section 2.3.

Object Usage In Arcadia, all software objects which are created by the tool fragment components of larger tools have the potential for being automatically stored by the object manager. There are at least two good reasons for doing this. First, by storing them, the object manager can later make them available as needed by other tools subsequently invoked by the user. This will result in an execution time saving. Second, these objects represent increased knowledge about the state of the software being developed or maintained and thus might well be of interest to the user. On the assumption that the user might at some future point wish to access this knowledge, it is stored by the object manager.

As observed earlier, these objects all have types which are assigned by the tool fragments which create them. Thus, another perspective on the outcome of executing a tool is that it effects the derivation of a collection of typed software objects by a set of tool fragments, some of which may not have been directly invoked by the user. These objects are persistent and are kept available for possible future reuse by subsequently invoked tools or for perusal by the user. In either case, the possible magnitude and diversity of these objects makes it imperative that this process be transparent to users, most of whom will not be interested in, or equal to, the task of correctly and efficiently orchestrating their creation and exploitation.

Automatic storage of all objects is not required, of course, as it may be undesirable under certain circumstances. For instance, to provide rapid response to user interaction, a suite of tool fragments may be employed which pass on information from fragment to fragment though a series of objects. Though these objects could be profitably retained, as described in the preceding paragraph, it may be expedient to not retain them as a performance penalty may be involved.

Incorporation of New Tools The internal composition of tools will be of limited interest to tool users, but will be of great interest to tool developers. Tool developers are expected to be able to consult Arcadia objects which store the specifications of current Arcadia tool fragments and object types and use those specifications in creating new tools. In Arcadia all such information about existing tool fragments and the abstract data types that they manipulate will be stored in centrally maintained objects. This will facilitate the process of altering and augmenting the type structure and set of tools.

The chief Arcadia object which stores information about existing fragments and types is the Type Derivation Graph. This graph contains as its nodes all of the abstract data types maintained by Arcadia, and has as its edges annotated designations of the various tools. New Arcadia types are incorporated by creating new nodes in the Type Derivation Graph. In order for the user to be able to create instances of this type, arguments to the appropriate creation/access routines may have to be derived from instances of existing types. Thus, loosely speaking, in order for the new type to be effectively incorporated into Arcadia, at least one tool which creates instances of the new type by calling appropriate routines must be written, and that tool must be represented in the type derivation graph by a set of edges connecting the new node to nodes representing the types of the objects which the new tool must take as input.

Again, this new tool will access capabilities for the creation of instances of the new type. In Arcadia, these capabilities must be clearly separated out into functional primitives belonging to the new type. These primitives—for the creation of new instances—must be accompanied by other functional primitives for describing and manipulating objects of the new type. In short, the new type will be integrated into Arcadia by the augmentation of the type derivation graph, by the creation of at least one new tool, and by the creation of a set of accessing primitives which is sufficiently complete to seal the implementation of the new type and make it a data abstraction in the usual sense of the term.

Although the preceding discussion described the way in which new tool fragments can be entered into Arcadia, it also indicates how existing tool fragments and object types can be altered. If the implementation of an existing object type is to be altered transparently to the tool fragments using it, this alteration can be done simply by replacing the bodies of the accessing functions defining the type (and of course converting any instances of that type). Alternatively, the new implementation can be viewed as implementing a new type, and as being able to create instances of the new type. If a new tool fragment, capable of creating objects of existing types, is to be incorporated into Arcadia, this will be done simply by incorporating into the type derivation graph a new edge or set of edges.

Tools are Objects, Too The preceding paragraph suggests that in Arcadia tools themselves are also viewed as objects. This is an important, though hardly novel, perspective on the architecture of our system, reflecting our desire to treat tools uniformly with other objects. (This view has been exploited successfully in many environments, such as Interlisp and Cedar [Tei84].) Tool fragments are themselves bodies of executable code which are created from source text by a derivation process which will be captured within Arcadia. When these tool fragments are incorporated into Arcadia as described above, they become capable of deriving new types of objects from existing types. They should not be viewed as inherently different from the objects and types upon which they operate, however. Clearly tools themselves demand careful version control, and Arcadia furnishes this. When the source text for a tool fragment is altered, the object code derived from it is presumably altered as well. Thus, objects derived by use of the new tool fragment cannot be assumed to be the same as objects derived from identical ancestor objects by the predecessor version of the tool fragment. Arcadia's version control and derivation tracking features will identify these differences and take proper account of them in managing the Arcadia object store. This will enable Arcadia system maintainers to maintain Arcadia with itself, to incorporate new tool fragments in parallel with older versions, to monitor the new fragments, and to replace older fragments when warranted.

Inter-tool Communication The model of communication between tools in Arcadia is the (possibly remote) procedure call, as all objects are modelled as instances of abstract data types. This is in contrast to, for example, sequential input/output of ASCII text, as is the model in Unix. With this model it should be possible to transfer information efficiently between tools. When "tight integration" is desired, multiple tools must be able to share data structures (i.e. concurrently access the same ADTs which are implemented as monitors). This will be especially important when the internal form of an object is not naturally linear. Tools, of course, will be written not knowing whether tight or loose integration will be employed, where "loose" means data is transferred through the file system. For instance, it should be possible, at the underlying implementation level, to pass a tree from tool to tool by "passing a pointer" to the root of the tree, without transforming the tree to a stream of text. A linearizing transformation of a data structure should ideally be necessary only when an environment object is moved to secondary storage by the object manager; practically, linearization may be necessary for moving between tools in different address spaces.

A related issue is flexibility in environment configuration. We expect that the hardware upon which the environment is hosted will vary: some users will be able to afford more resource-consumptive tools than others, or will be able to afford more expensive styles of interaction. For instance a user having a machine with a large physical memory may desire that several tools execute concurrently, improving efficiency by communicating through shared reference to objects. Moreover many tools may be concurrently active. Other users may need an environment which makes extensive use of a file system for communication between tools and parsimoniously uses memory. Flexibility in environment configuration is therefore called for, to suit varying performance requirements and resource availability.

2.3 User Interface

A succession of systems from Xerox [TM81] [Tei84] [GR83] [SIK*82] and from Xerox émigrés [Ins85] have refined a style of user interface characterized by the illusion of directly manipulating a set of objects depicted on the screen. Several properties are necessary to maintain this illusion, including rich visual representation of objects and immediate reaction to user input. To support this approach, Arcadia will be hosted primarily on powerful workstations with bitmapped graphics and pointing devices. The usual amenities will be provided, including nested and overlapping (or tiled) windows and selection of objects and operations with a pointing device.

Managing the User Interface Graphical user interfaces with windows and mice are now common, but most of these systems are based on toolkits which must be incorporated piecemeal into an application program. Details of the user interface are woven into the application, complicating both.

Recently, some researchers have described an approach in which the user interface is completely separated from the functional aspects of an application. Manipulation of objects is logically separated from maintenance of a depiction of those objects, and user input is mapped into abstract commands which are independent of the particular command syntax visible to the user. This separation of concerns has several advantages. Tool fragments are simplified because they need not contain user interface code, and the same fragments can be incorporated in interactive and non-interactive tools. Klefstad [Kle85] has shown how this approach can maintain uniformity of the interface across tools while allowing customization of the interface to suit each user and adapt to the available terminal device. Coutaz [Cou85] argues that such a separation also allows the user interface and tool functionality to evolve independently. An important contribution of her work is identifying the object to be displayed, the abstract depiction of that object as maintained by the user interface system, and the concrete depiction of the object on the display device, as three objects subject to independent but coordinated manipulation.

The Arcadia display manager will be a separate collection of tool fragments which manage the relation between objects, their depictions, and their displays. The input manager, in coordination with the display manager, will provide tools with a sequence of commands in exactly the same manner as those commands would be received if the tool were running in a non-interactive activity. If we view the environment for a moment as a processor for a language in which the objects are all the objects of the environment and the operations are provided by tool fragments, then we can view this organization of the user interface as separating the syntax of a command language from its semantics.

Command Language The approach taken to developing a command language or languages for Arcadia reflects an attempt to satisfy two potentially conflicting requirements. On the one hand uniformity of user interface, regardless of the function being performed, is held as greatly desirable. This uniformity can be achieved most readily by choosing a single command language which is used in all contexts. If the command language is sufficiently rich, such as Ada or an extension thereto, this will probably be an acceptable choice in many environments. Arcturus successfully adopts this approach for an Ada programming environment. The work of Klefstad even demonstrates that a respectable amount of extensibility can be retained with this approach [Kle85].

On the other hand, study of the plethora of activities that occur in the broad spectrum of system development reveals that many different *conceptual models* are used in the various activities. For instance, very different sets of concepts underlie the graphical manipulation of Petri nets, typesetting of documents, verification of an algorithm, and assessment of the progress of a project. The interface to automated aids which support these activities should, therefore, be consistent with these varying models. In a given context, such as Petri net editing, the user should have available the exact set of semantic commands needed for that activity, phrased in appropriate terminology, reflecting the appropriate conceptual model. As the context changes, so does the conceptual model, and therefore so changes the set of commands. This, apparently, argues for non-uniform user interfaces.

We are attempting a synthesis governed by the following propositions. First, there exist *classes* of tools which share some fundamental concepts. "Editors" is an example class. Second, the *syntax* of commands is an orthogonal issue to that of the conceptual model.

Our approach to commanding Arcadia is, therefore, as follows. Ignoring for the moment the issue of syntax, at any moment when the user is in contact with Arcadia there are a few conceptual commands always available. This set is not agreed upon yet, but may well include "terminate this activity" and "help!". Then, depending on the class of activity in which the user is engaged, there exist commands which are universal to this class. There may be several levels of classes. Finally, the user will be performing activities in a specialized context; within it exist commands peculiar to that activity. The set of commands which may be entered at any time includes, therefore, locally defined commands, commands provided by the parent class, and on up the chain until the outermost level is reached. Commands at a given level are not able to hide commands at higher levels [3] [4]. Thus

[3] This approach bears a top-level relation to Smalltalk's organization. Arcadia will differ, however, in the implementation and in the inheritance mechanism. We seek an inheritance mechanism which will exhibit some of the desirable properties of strong typing and information hiding.

[4] Note that classes of tools mix with classes of objects. That is, there may be a set of operations

the command language available to the user, in terms of semantic concepts, varies from context to context, but exhibits uniformity of concepts within a class. The overall principle, therefore, is the meta-notion of selecting semantic constructs.

At each level in this hierarchy of semantic commands different syntax for invoking the commands could be used. This seems undesirable (though that is not without debate) and is not necessary in Arcadia; uniform syntax is an achievable goal. Specifically, at all levels icons could be used, or menu picks (including the picking of points in a graphical region), or textual function signatures. [5] The syntax may be device dependent. The Arcadia user interface facilities will therefore enable the development of substantial sets of tools which are uniform syntactically and, more important, uniform semantically.

Graphical Depiction of Objects Interactive manipulation of an object involves the coordinated management of three distinct representations: an internal representation suited for manipulation by the involved tool fragments, a representation which captures the additional information necessary to graphically depict the object, and a device-oriented representation (e.g., bitmap) actually presented to the user. A component of the Arcadia framework will map the abstract graphical depiction into the concrete depiction, hiding from all other components the details of managing a particular device. Artists, written by tool builders, will be responsible for mapping the internal representation of an object into the abstract depiction accepted by Arcadia.

Artists, introduced by Myers for an interactive debugging system [Mye83], contribute to the modularity goal of Arcadia by separating the functionality of tool fragments from display. Artists also contribute to the portability of Arcadia by adapting a tool to different classes of devices. The graphical interface component of the Arcadia framework can hide the details of managing a particular device, but it can't hide such gross characteristics as whether the user is seated at a bitmap workstation or a character-oriented terminal. A petri net editing tool, for instance, might use a graphically oriented artist when communicating through a workstation and a textual petri net artist when connected to a character terminal. 'Plug-compatible' artists can also be used on the same device when the desirable

universal to editors, and another set of operations universal to petri nets. A petri net editor consists of a set of tool fragments providing an appropriate combination of those two sets of operations. Moreover, objects mix with objects. A textual editor for Ada programs should provide editor operations, textual operations, and (some) program operations. This is a sort of inheritance– class 'editable petri net' may be created by inheriting capabilities from 'petri net' and 'editing'.

[5] Arcturus uses the latter approach exclusively. The "ideal" implementation of Arcturus can thus be viewed as using Ada as the implementation language and as the sole command language.

depiction is a matter of user preference.

Display Mapping Artists and the graphical interface component of Arcadia map, in two steps, the internal representation of objects into concrete depictions. Furthermore, an inverse mapping, from physical screen locations to objects, must also be maintained to support input with a pointing device. First, a two-way mapping between *objects* and components of the *abstract depiction* will be maintained by the operations on the abstract depiction data type supplied as part of the Arcadia framework. Second, a two-way mapping between parts of the *abstract depiction* and parts of the *concrete depiction* will be maintained by the graphics subsystem. When the user points to a concrete depiction of an object, Arcadia will associate the screen position with an object and inform the tool of the object selection.

Binding of Tools and Artists A tool, as we have described it above, is not a static entity but rather a temporary alliance of tool fragments cooperating in the manipulation of one or more objects. It is not reasonable, therefore, to statically bind artists to particular tool fragments. It is better to think of an artist as an extension of the operations supported by an object, adding some operations (e.g., *select*) and extending some others (i.e., causing each change to the state of an object to result in a corresponding change in the graphical depiction). To some degree this view of artists is consonant with our view of all objects in Arcadia as instances of abstract data types. It points out, however, some important requirements for object management in Arcadia.

First, since the abstract depiction constructed by an artist includes references to the depicted object (the two-way mapping discussed above), it cannot be managed completely independently from the depicted object. For instance, the user of a petri net editor may move places and transitions around on the screen. After an editing session the abstract depiction will contain useful layout information that ought to be preserved for future sessions. Another, non-interactive tool may modify the internal representation of the same net in other ways. The references from the abstract depiction to the depicted object must be managed in such a way that they do not become invalid when this happens.

Second, we wish to minimize the effort required to build new objects and their artists. The Incense system [Mye83] provided default artists for data structures. Arcadia will also provide a scheme for supplying default artists and inheritance of artists from related object types. For instance, the default artist for a record containing fields of type *foo* and *bar* should be composed from the generic record artist, the artist for type *foo*, and the artist for type *bar*.

3 Aspects of the Ada Environment

The first use of the Arcadia environment architecture will be as the foundation for an advanced Ada software development environment. In the short term, construction of this advanced Ada environment will provide an opportunity to exercise and evaluate the integration and extensibility capabilities of the Arcadia architecture. A first version of this environment will be populated with a set of ***basic components*** needed to support programming and program validation activities. This version will also include a set of ***tool-building tools***, such as lexer and parser generators, that could prove valuable to a wide variety of software developers.

The longer term objective is to use the extensible Arcadia environment architecture as a research platform to develop and study prototype tools providing software developers with ***extended capabilities*** that go far beyond simply supporting coding. Tools supporting analysis, at every stage in the software development process, are of particular interest. This prototyping activity will build upon and use the basic components and tool-building tools contained in the first version of the Ada environment as well as exploit the extensibility of the Arcadia architecture.

This section outlines Arcadia activities in each of these areas.

3.1 Basic Capabilities

The basic capabilities needed in any Ada environment are primarily directed at composing, running and debugging Ada programs. A variety of such tools are currently in various stages of development. Already in existence is an Ada front-end that performs lexing, parsing (with decent error recovery), tree building, and pretty-printing. It is completely written in Ada. Current efforts are directed at fleshing out other basic components of a programming environment. For instance, work is underway on a static semantic analyzer that will transform the output of the front-end into the IRIS internal representation described below. We are also developing an interpreter and a debugger that uses IRIS. Among the basic components not being developed by Arcadia consortium members are editors and code generators. At present, these are two areas in which we prefer to import and integrate existing tools into the environment rather than expend our own development resources.

From an Ada perspective, perhaps the most interesting of the basic capabilities in the Arcadia Ada environment is the internal representation used for programs. Led by members from Incremental Systems Corporation and the University of California at Irvine, the consortium has explored a range of possible options. Our primary goals for the internal representation have been extensibility, simplicity, performance, and capacity. In particular we are seeking to minimize the amount of special case processing performed by tools, provide a general capability for adding arbitrary unanticipated fields as later required by individual tools, and requiring that only the attributes used by a given tool need be local when that tool is executing.

The internal form that has resulted from our deliberations is called IRIS (*I*nternal *R*epresentation *I*ncluding *S*emantics). IRIS is a graph structure that represents an Ada program including information derived from static semantic analysis (e.g., overload resolution).

The nodes in an IRIS attributed graph are of only two types: terminal and non-terminal. Non-terminal nodes contain an indication of the operator represented by the node, an indication of the number of operands required for this operator, and pointers to the appropriate number of operand subtrees. The IRIS operators have been chosen to reflect the semantics of Ada, and not its syntax. Terminal nodes contain an indication of the nature of the contents of the node, literal or non-literal, followed by the appropriate contents. Literals include integers, characters, strings, reals, and identifiers. If the node is a non-literal, then an operator field is present which has the same content and interpretation as the operator field of non-terminal nodes. This implies that the terminal node may contain a pointer to an object declared elsewhere in the IRIS graph [6]. The detailed specification of the contents of IRIS nodes is still in flux, as consequences of various choices are being examined.

3.2 Tool-building Tools

Tool-building tools will play a dual role in the Arcadia Ada environment. Their first use will be to facilitate the work of consortium members in developing prototypes of additional tools, primarily analysis tools, that will eventually be integrated into the environment. They are also, however, of potential value to end users of

[6] Note that this represents an unusual use of the phase "terminal node". It is terminal in the sense that there are no additional IRIS subtrees hanging off this node. This pointer may only point up in the IRIS tree, presumably to the declaration of an object for which this node represents a reference.

the environment, since many software systems employ components such as parsers or attributed graph structures. Hence they will remain among the tools available in later versions of the environment.

Two important tool-building tools are currently in existence: alex and ayacc. Their input specifications are nearly identical to their Unix counterparts, lex and yacc. They produce, respectively, scanners and parsers that are written in Ada. Both alex and ayacc are written in Ada as well.

Work is presently progressing on another tool-building tool, called GRAPHITE, that supports the use of attributed graph representations by facilitating the encapsulation of attributed graph classes into abstract data types. This tool comprises a Graph Description Language (GDL), for succinct definition of classes of attributed graphs, and a processor that automatically creates an Ada implementation for a class of attributed graphs from its GDL description. Thus GRAPHITE allows its users to treat attributed graphs and their operations as primitive constructs in a higher-level programming language and to enjoy the benefits of information hiding. Perhaps the most interesting aspect of GRAPHITE is its support for prototyping activities such as those being undertaken by Arcadia consortium members. Using GRAPHITE, it is possible for different groups to experiment with definitions of attributed graph objects, without forcing recoding or even recompilation of those tools that access the redefined object but do not explicitly use the new information. Details of GRAPHITE may be found in [WCW86].

3.3 Extended Capabilities

Our plans for extended capabilities, to be developed within one to three years, focus particularly on analysis activities. Our interests lie in pre-implementation analysis, analysis applicable to distributed and real-time systems, testing and debugging, host-target analyses, analysis of inter-module relationships, and support for efficient analyses of large systems. We also plan to support language extensions, such as Anna [vHLKO85] [LvH85], project management activities, and some forms of automatically aided program generation. A brief overview of some of our plans is given here.

One class of extended capabilities will support rigorous and systematic testing of Ada programs. Work is underway to establish a suitable set of test coverage methods [CPRZ85] and to automate them. Other work is aimed at evaluating, integrating, and automating test data selection strategies. Both kinds of testing capabilities are specifically targeted for use in conjunction with Ada programs and for implementation in Ada.

Another class of extended capabilities will support debugging of Ada tasking programs. Work is underway on a powerful tasking debugger that supports breakpointing, value modification, and full control over task scheduling, yet retains consistency with key Ada semantics [BTM85]. This debugger is based on a uniprocessor interpreter for tasking programs. Its use in debugging host-target configurations is also being investigated [Tay84]. Also underway is work on a high-level debugger for distributed systems that supports monitoring of distributed programs at a level above that of the Ada language primitives. This approach allows users to specify the higher level events, typically consisting of complex patterns of event occurrences, that the debugger should notice, then wait for the debugger to announce when one has occurred [BW83]. Various distributed implementations of this debugging tool are being explored.

Yet another class of extended capabilities address pre-implementation analysis. Work on a static analysis method for concurrent Ada programs [Tay83] provides a basis for tools that will derive a complete concurrency history from an Ada program. Tools based on the constrained expressions approach [ADWR86] will permit designs of concurrent systems, expressed in an Ada-based design notation, to be analyzed. Such analyses will allow users of the Arcadia Ada environment to detect errors in their concurrent programs long before those programs are coded and ready for testing. Cooperative application strategies for these techniques are also being studied.

Support for programming-in-the-large will also be among the extended capabilities provided in the Arcadia Ada environment. The Precise Interface Control (PIC) language constructs have been tailored to support the description of interface relationships among the modules of an Ada software system [WCW85]. Tools to support analysis of those interface relationships, throughout the software development process, are currently being implemented.

Finally, management tools will also be among the extended capabilities of the environment. Capabilities pioneered in the TRW Software Productivity Project [BPS*84] will provide managerial control over the software development process as practiced by users of the Arcadia Ada environment.

4 Development Strategy

In this section we present an overview of the Arcadia development strategy. First, we describe our assumptions about the host environment on which the Arcadia software development environment will be implemented. Then we discuss our maturation and technology transfer plans. Finally, we describe the internal organization of the Arcadia consortium.

An important objective for the Arcadia environment is to achieve modest portability. We believe that a prototype demonstrating novel research directions is of limited value unless it can be widely disseminated among the research community. Although support for full portability would extract far too heavy a toll on our development efforts, modest portability can probably be achieved at reasonable cost. To accomplish this goal we have decided to base our host environment upon those resources that are widely available in current academic or industrial research and development organizations. To be specific, we will use the programming language Ada, the UNIX operating system, and VAX mainframes networked with SUN workstations.

Although it could be argued that Ada is not widely available, we believe that this situation is quickly being remedied with the increased introduction of validated compilers. An explicit goal of the Ada design was to support portability. We want to exploit this feature as well as take advantage of Ada's support for advanced programming language concepts, such as abstract data types and distributed processing.

Despite agreement on the host environment, we still intend to limit and isolate operating system and machine dependencies as much as reasonably possible. For example, we expect file manipulation and graphics capabilities to be potential portability problem areas. In both cases, we intend to select a relatively small set of primitive operations and use an Ada package to map these operations onto the host environment. These packages will most likely have to be recoded if the Arcadia environment is ever ported to other machines or operating systems.

As the prototype environment is developed we intend to carefully evaluate it. One of the advantages of the environment architecture, basic components, and tool-building tools is that together they provide a research platform for evaluating individual analysis tools or groups of tools. Moreover the addition of new tools to the environment is one way to evaluate how well the architecture supports integration and extensibility. In addition to subjective evaluations based on individual use and experience, we intend to conduct objective experiments. In this regard, we plan to monitor the performance and use of the overall environment as well as undertake tool-specific experiments.

This evaluation process, which will be an on-going activity, will lead to a succession of environment prototypes, each with more refined and extended capabilities than the previous version. At appropriate points in the development process, some of the architectural components or environment tools may become candidates for production-quality development. Recognizing that the transition from experimental to production quality poses several difficulties, we are trying to plan for technology transition activities. Most of the industrial consortium members are involved in this planning. Although producing and maintaining production quality versions is infeasible for the university members of the consortium, it is our expectation that industrial members may choose to fulfill this role. Other organizations, whose primary charter is technology transfer, may also choose to become involved in these activities.

The idea of forming a research consortium was under discussion among the Arcadia principals for some time before the first meeting was actually held and members agreed to work together on this project. Before that, all of the members had had previous research interactions with each other and some had worked together on projects in the past. Most notably, all the members continue to share a common world view. This view is founded on the belief that software development environment research needs to be expanded well beyond the domain of program development tools. In particular, consortium members emphatically agree on the crucial importance of integration, extensibility, and powerful analysis tools applicable throughout the software development process. This background has enabled the consortium to function effectively, with no external influence necessary.

Members of the consortium meet about every three months to distribute plans and share ideas. These meetings lead to in-depth discussions and evaluations of the work that each group is doing. Members find that this immediate feedback, although often critical, provides new insights into the problems and positively affects their research. Members are committed to sharing software and expertise. There have even been some short-term exchanges of personnel between organizations.

Working in a consortium adds overhead to the development process; ideas must be discussed until there is some consensus and software must be developed to conform to the dictates of the architecture. On the other hand, members of the consortium can build upon each other's work. In fact, one of the primary motivations for forming a consortium was recognition of the huge software development effort that was involved in building even a prototype environment. Basically, because of the sharing of ideas, expertise, and software, we expect the results of the consortium effort to be greater than the sum of the contributions each organization could make separately.

5 Acknowledgements

There is hardly a single aspect of Arcadia which has not been shaped by participants from each institution in the Consortium. Over the past year the following people have been involved in the design of Arcadia. At UCI: R.W. Selby, C. Snider, C. Kelly, I. Shy, S. Sykes, R. Schmalz, T. Nguyen. At Colorado: D. Baker, D. Heimbigner, G. Clemm. At Massachusetts: A. Wolf. At Aerospace: A. Brindle, R. Cowan, D. Martin. At TRW: F.Belz, B. Boehm. At Incremental Systems Corporation: D. Fisher. We also readily acknowledge the contributions of S. Squires.

References

[ADWR86] G. S. Avrunin, L. K. Dillon, J. C. Wileden, and W. E. Riddle. Constrained expressions: adding analysis capabilities to design methods for concurrent software systems. *IEEE Transactions on Software Engineering*, SE-12(1):(to appear), January 1986.

[BPS*84] B. Boehm, M. Penedo, E. Stuckle, R. Williams, and A. Pyster. A software development environment for improving productivity. *IEEE Computer*, 30–42, June 1984.

[BTM85] Anne F. Brindle, Richard N. Taylor, and David F. Martin. *A Debugger for Ada Tasking.* Technical Report ATR-85(8033)-1, The Aerospace Corporation, El Segundo, CA, 1985.

[BW83] Peter Bates and Jack C. Wileden. High-level debugging of distributed systems: the behavioral abstraction approach. *Journal of Systems and Software*, 3:255–264, 1983.

[CO85] Geoffrey M. Clemm and Leon J. Osterweil. The integration of Toolpack/IST. In *IFIP Working Conference on Problem Solving Environments for Scientific Computing*, IFIP, Sophia-Antipolis, France, June 1985. To appear.

[Cou85] Joëlle Coutaz. Abstractions for user interface design. *Computer*, 18(9):21–34, September 1985.

[CPRZ85] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil. A comparison of data flow path selection criteria. In *Proceedings of the Eighth International Conference on Software Engineering*, pages 244–251, ACM Sigsoft, London, August 1985.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language And Its Implementation.* Addison Wesley, 1983.

[HM85] Dennis Heimbigner and Dennis McLeod. A federated architecture for information management. *ACM Transactions on Office Information Systems*, 3(3):253–278, July 1985.

[Ins85] *Inside Macintosh.* Apple Computer, Inc., Cupertino, California, promotional edition, March 1985.

[Kle85] Raymond O. Klefstad, II. *Uniform User Interface for a Programming Environment.* PhD thesis, University of California, Irvine, Department of Information and Computer Science, 1985. In preparation.

[LvH85] David C. Luckham and Friedrich W. von Henke. An overview of ANNA, a specification language for Ada. *IEEE Software*, 2(2):9–22, March 1985.

[Mye83] Brad A. Myers. Incense: a system for displaying data structures. *Computer Graphics*, 17(3):115–125, July 1983.

[OF76] Leon J. Osterweil and Lloyd D. Fosdick. DAVE – a validation, error detection, and documentation system for FORTRAN programs. *Software — Practice & Experience*, 6:473–486, 1976.

[Ost83] L. J. Osterweil. Toolpack— An experimental software development environment research project. *IEEE Transactions on Software Engineering*, SE–9(6):673–685, November 1983.

[SIK*82] David Canfield Smith, Charles Irby, Ralph Kimball, Bill Verplank, and Eric Harslem. Designing the Star user interface. *BYTE*, 7(4):242–282, April 1982.

[Tay83] Richard N. Taylor. A general-purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, 26(5):362–376, May 1983.

[Tay84] Richard N. Taylor. Debugging real-time software in a host-target environment. *Technique et Science Informatiques (Technology and Science of Informatics)*, 3(4):281–288, 1984.

[Tei84] Warren Teitelman. A tour through Cedar. In *Proceedings of the 7th International Conference on Software Engineering*, pages 181–195, Orlando, FL, March 1984.

[Tic82] Walter 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, September 1982.

[TM81] W. Teitelman and L. Masinter. The Interlisp programming environment. *Computer*, 14(4):25–33, April 1981.

[TS85] Richard N. Taylor and Thomas A. Standish. Steps to an advanced Ada programming environment. *IEEE Transactions on Software Engineering*, SE-11(3):302–310, March 1985. (Appeared earlier in Proceedings of the Seventh International Conference on Software Engineering, Orlando, March, 1984).

[vHLKO85] Friedrich W. von Henke, David C. Luckham, Bernd Krieg-Brueckner, and Olaf Owe. Semantic specification of Ada packages. In John G. P. Barnes and Gerald A. Fisher, Jr., editors, *Ada in Use: Proceedings of the Ada International Conference*, pages 185–196, Association for Computing Machinery and Ada-Europe, Cambridge University Press, Paris, May 1985. Also available from Cambridge University Press as a volume in the Ada Companion Series.

[WCW85] Alexander L. Wolf, Lori A. Clarke, and Jack C. Wileden. Ada-based support for programming-in-the-large. *IEEE Software*, 2(2):58–71, March 1985.

[WCW86] A. Wolf, L. Clarke, and J. Wileden. GRAPHITE: a meta-tool for Ada environment development. In *Second Ada Applications and Environments Conference (submitted)*, 1986.