A REPORT ON ARCADIA

A Presentation to the ACM-SIGAda Future APSE '86 Workshop

Alexander L. Wolf

COINS Technical Report 86–59
December 1986

Software Development Laboratory
Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

A version of this report to appear in
Proceedings ACM-SIGAda Future APSE '86 Workshop

This work supported in part by the following grants: Rome Air Development Corporation, No. SCEEE–PDP/85–0037; National Science Foundation, No. DCR–84–04217 and No. DCR–84–08143; and Control Data Corporation, No. 84–M103.

ABSTRACT

Arcadia is a research project aimed at the discovery and development of both *architectural principles* and *software tools* for software development environments. The principles are intended to enable the construction of environments that are both integrated and extensible, while the tools are intended to support the description and analysis of software systems throughout their lifetimes, from initial conception through maintenance. A major goal of the Arcadia project is to create a research platform that can be used to experiment with new principles and new tools. This platform will initially be used to build Arcadia-1, a first prototype of the sort of next-generation environment that we believe can better support the development and maintenance of large, complex software systems.

The Arcadia project is organized as a consortium of academic and industrial researchers. The principal members are from the University of California at Irvine, the University of Colorado at Boulder, the University of Massachusetts at Amherst, Stanford University, Incremental Systems Corporation, TRW, and The Aerospace Corporation. In addition to their research contributions, the industrial members are expected to act as conduits of the technology that emerges from the Arcadia project.

This paper provides a brief report on the research directions being explored by members of the Arcadia project in the areas of architectural principles and software tools, and describes our plans for Arcadia-1. The report was originally presented by the author at the ACM-SIGAda Future APSE '86 Workshop.

1. Introduction

Arcadia is a research project aimed at the discovery and development of both *architectural principles* and *software tools* for software development environments. The principles are intended to enable the construction of environments that are integrated and extensible—integrated in that there is synergy in the underlying operation of the environment's tools as well as uniformity in the user interfaces to those tools, and extensible in that it is relatively easy to absorb new capabilities into the environment without changes to the fundamental architecture. The emphasis in the tools area is on supporting the description and analysis of software systems throughout their lifetimes, from initial conception through maintenance. This emphasis should benefit both engineers and managers, providing them with meaningful information about, and control over, the evolution of software systems.

A major goal of the Arcadia project is to create a research platform that can be used to experiment with new principles and new tools. This platform will initially be used to build Arcadia-1, a first prototype of the sort of next-generation environment that we believe can better support the development and maintenance of large, complex software systems. Some of the more significant features of Arcadia-1 will be:

- mechanisms to formally describe and automatically support/enforce multiple software process models;
- mechanisms to organize and manage the diverse set of objects associated with a software system, such as requirements documents, management reports, source code, and test data;
- mechanisms to organize, manage, and integrate the diverse set of tools populating the environment;
- mechanisms to manage and integrate user interfaces;
- various analysis tools; and
- various tool-generation tools ("meta-tools").

Included as a component of Arcadia-1 will be a set of tools appropriately viewed as a software development environment for Ada. In fact, Arcadia-1

is being implemented in Ada; the Ada-oriented tools in Arcadia-1 will be used to aid in the prototype's own, incremental development.

The Arcadia project is organized as a consortium of academic and industrial researchers. The principal members are from the University of California at Irvine, the University of Colorado at Boulder, the University of Massachusetts at Amherst, Stanford University, Incremental Systems Corporation, TRW, and The Aerospace Corporation. In addition to their research contributions, the industrial members are expected to act as conduits of the technology that emerges from the Arcadia project. They will be responsible for producing and maintaining production-quality versions of architectural components and tools. Other organizations, whose primary charter is technology transfer, may also become involved in this process.

The remainder of this paper provides a brief report on the research directions being explored by members of the Arcadia project in the areas of architectural principles and software tools. A more detailed overview of the Arcadia project can be found in [14] and [15].

2. Architecture Issues

The Arcadia project is addressing the needs of an extremely difficult application domain: the development and maintenance of long-lived, large-scale, possibly concurrent/distributed/real-time software systems by teams of engineers and managers working on a network of distributed workstations. This domain forces us to rethink many long-held beliefs about software and its construction.

"Software" in this domain is much more than simply the code that runs the hardware. It includes requirements, specification, and design documents, test plans, test data, and test results, management guidelines and management reports, user documentation and user reports; in sum, it is the entire collection of artifacts or *objects*, whether manually or automatically produced, pertaining to the development, maintenance, management, and use of a software system.

Correspondingly, "software development and maintenance" are much more than simply the writing or altering of the code that runs the hardware. They are the production of all the diverse objects mentioned above as

well as the establishment of relationships among those objects, such as, for example, consistency between test data and test results or between requirements and design documents. It is the responsibility of the environment to provide and integrate the *tools* that automatically support these activities. The tools can be seen as transformers that operate upon the objects, producing new objects from old objects, such as a compiler that transforms source objects into target objects or a productivity monitor that transforms source objects into productivity-report objects.

Our approach to supporting this domain is further affected by a recognition that the development, maintenance, and management of a large software system involves complex combinations of various kinds of activities or *software processes.*[1] Researchers have sought to understand and characterize software processes using so-called *software process models* [16]. An example of a software process is regression testing, while an example of a software process model is the Spiral Model [4]. From the perspective of an environment, a given software process can be viewed as the orderly (although sometimes nondeterministic) application of a certain set of tools to a certain set of objects. The environment should support the description of software processes and, to the extent that they can be automated, support their execution. Some software processes, such as "recompilation of source code", are fairly well understood and are easily automated. Others, such as "update of specification documents to reflect changes in design documents", while not as easily comprehended or described, are just as important to the successful production or enhancement of a large software system and thus deserve equal treatment by the environment.

In sum, the issues facing environment architects fall into three broad categories: object management, tool management, and software process management. The next three sections outline the approaches being taken in the Arcadia project to resolve the issues in each of these categories.

[1] The concept of a software process should not be confused with the operating-system concept of a process; we are not concerned here with such things as process control blocks, program counters, and registers, but rather with the activities of engineers and managers, such as editing, testing, and project planning.

2.1 Object Management

The number and variety of objects associated with a large software system and the complexity of the interactions among those objects provide strong impetus for intimately involving the environment in the production and management of the objects. We are developing the notion of an *object base* as the foundation of object management. The object base reflects the following characteristics of objects.

- Objects are instances of (abstract data) types. Type information is used both to control which tools can operate on which objects (e.g., to prevent a text editor from operating on an executable image) and how those tools can manipulate those objects (e.g., to prevent a Petri-net graph node from being placed into an abstract syntax tree).

- Objects may be persistent—that is, the lifetime of an object may extend beyond activation of tools that manipulate the object.

- Objects may be arbitrarily large or small. Examples of objects include source code, object code, symbol tables, lexical tokens, graph nodes, graph edges, test data, test results, text (e.g., documentation), and bit-map display frames.

- Software process descriptions are objects, as are the types (i.e., descriptions) of all objects and the input/output specifications (i.e., descriptions) of all tools. The significance of maintaining these as objects in an environment is explained in Section 2.3.

The object base is further characterized by the fact that it is extensible—that is, new types of objects can be incorporated into the object base to account for new tools in the environment—and that it is distributed. Moreover, the object base contains explicit representations of tool and object relationships. Examples of some classes of relationships include hierarchy (e.g., tool activation sequences, composite objects), consistency (e.g., results of analyses), and derivation (e.g., subtyping, versioning).

2.2 Tool Management

The view of tools being taken in the Arcadia project emphasizes small, reusable building blocks. In particular, tools may be composed of a set of *tool fragments* allied for the purpose of performing some task; these alliances may possibly involve concurrency and may possibly be dynamic and short-lived. The tool-fragment structure, however, is kept invisible to users of a tool. An example of a tool seen as a set of tool fragments is a pretty printer composed from a scanner, a parser, and a formatter. Notice that the same scanner and the same parser can also be fragments of other tools, such as a compiler.

The application of tools to objects is realized through (possibly remote) procedure calls and may be statically or dynamically determined. Dynamic applications may be data driven or event driven. Moreover, some tools may be "self-motivated"—that is, they essentially determine their own activations.

Conceptually, all communication among tools is through the object base; the output objects of one tool serve as the input objects to other tools. In some cases, this conceptual view may be abandoned at lower levels of implementation to improve efficiency. Nonetheless, the tools—and in particular the tool writers—should be unaware of whether communication is "loose" (i.e., objects are shared through the object base) or "tight" (i.e., objects are shared through primary memory). In fact, different approaches may be used with a tool during different activations of that tool.

While we have chosen to consider tool management as a separate category of architecture issues, the tools themselves are actually treated as objects; these objects and their relationships happen to be of particular interest to developers of environments. For instance, the tool-fragment structure is describable as a set of relationships (hierarchy) over a set of objects (tool fragments). This unified view of objects and tools has significant advantages for software process management.

2.3 Software Process Management

Software process management is the glue that holds a project together. At a high level, it organizes and coordinates the activities of the project's

engineers and managers. This is reflected at a lower level in the organization and coordination of the application of tools to objects. Unfortunately, software process management is today enforced only by convention and so can easily be circumvented, whether intentionally or unintentionally. Moreover, the models under which software processes operate are not well understood, nor can they easily be experimented with.

A major reason for the current shortcomings in software process management is the lack of a satisfactory medium for the rigorous description of software processes and software process models. Thus, we have begun the development of a *software process programming language* [12]. The (primitive) operators in this language are an environment's tools and the operands are an environment's objects. Programs in this language realize software processes. Execution of these programs amounts to an automation and concomitant enforcement of software process models. Although the definition of this language is in its early stages, certain properties of the language can already be identified.

- The language will have a type structure rich enough to describe any object that can reside in the object base.
- The language will allow the definition and maintenance of relationships among the objects in the object base.
- The language will allow the specification of the input/output behavior of tools.
- The language will have facilities for expressing control flow and procedural abstraction, including facilities for expressing concurrent execution.

Given the existence of machine-readable descriptions of software processes, software process management takes on a broader meaning. In particular, software process programs, just like application programs, have associated with them development and maintenance activities (i.e., software processes) and so can profit from being treated as genuine objects in the environment. Moreover, software process management must account

for changes in the software processes and the software process models under its purview, propagating the effects of those changes throughout the system.

2.4 User Interfaces

While the Arcadia project's approaches to object, tool, and software process management are the primary contributions to environment-architecture research, there is another important area, namely user interfaces, in which Arcadia research is providing interesting results [18]. The remainder of this section briefly sketches this work.

The Arcadia project assumes that next-generation environments will be hosted primarily on powerful workstations that provide bit-mapped graphics and pointing devices, making extensive use of multiple windows and multiple, simultaneous (operating system) processes. Within this setting, any object is given the opportunity of being "viewed", but maintenance of a depiction of an object is logically separated from manipulation of that object by tools. This separation is realized by drawing a distinction between an object and its abstract depiction, and between an abstract depiction and a concrete depiction on a particular graphics device. Tools manipulate only the object, while so-called *artists* are responsible for mapping the object into an abstract depiction. The final step, mapping the abstract depiction into a concrete depiction, is handled by yet another environment component, hiding the low-level details of managing a device from all other components. To maintain the integrity of the type structure of the environment's object base, operations that are used to "graphically" manipulate an object form part of the type of that object. In fact, an artist can be viewed simply as an extension of the operations associated with an object.

Given the diversity of tools we envision will populate an environment, a particularly difficult challenge is formulating a framework for user interfaces that promotes a feeling of integration among those tools. We believe that such a feeling enhances the usability of an environment. Our approach is based on separating out the two, orthogonal components of a tool's user interface, namely the inherent *conceptual* or *semantic model* of how a user interacts with the tool and the specific *syntax* of commands. User input is mapped into abstract commands appropriate to a conceptual model; ab-

stract commands are independent of the particular command syntax visible to the user. Uniformity of user interfaces is then achieved through identification of *classes* of tools, such as the class "editor" or the class "report generator", within which there is a significant sharing of some fundamental concepts. Presentation of shared concepts in the form of commands should be made consistent across the tools in a class. Of course, classes are not necessarily disjoint. Rather, they form a hierarchy in which classes at lower levels in the hierarchy inherit commands from higher levels, which is similar in some respects to the way methods are inherited in a Smalltalk class structure. So, for example, it would make sense for the tool class at the highest level of the hierarchy to have a "terminate session" command associated with it and for that command to be shared uniformly (i.e., inherited) by tools in all other classes.

3. Software Tools

As mentioned in the introduction, the Arcadia project is committed to advancing the capabilities of software tools as much as it is involved in providing a suitable infrastructure into which those tools can be placed. Our work in this area, which to date has been somewhat focused by the needs of the Ada-oriented component of Arcadia-1, is aimed at developing three kinds of tools: meta-tools to support the building of tools; basic capabilities to support the primitive programming needs of Ada projects; and extended capabilities to support sophisticated description and analysis techniques throughout the lifetime of a software system. This section briefly outlines our work on these tools.

3.1 Meta-Tools

It is not uncommon to find subsets of tools within an environment that have similar underlying control and/or data structures. For example, the various languages used in an environment, such as specification, design, and implementation languages, all require some sort of language processor. It is also not uncommon to find subsets of tools that share particular kinds of objects, such as several analysis tools that examine the same representation of a program. Furthermore, as tools and objects undergo development,

they undergo change. Meta-tools are intended to ease the effort involved in developing an environment's tools and objects by automating the (re)coding of major components of those tools and the (re)coding of interfaces to those objects. To date, the Arcadia project has produced three such meta-tools: ALEX, AYACC, and GRAPHITE, all written in Ada.

ALEX and AYACC are, respectively, a scanner generator and a parser generator for language processors. While the specifications that are the inputs to these tools bear strong resemblance to the inputs to the Unix tools LEX and YACC, ALEX and AYACC differ from their Unix counterparts in two significant ways. First, they are based on somewhat different algorithms and second, they produce scanners and parsers that are coded in Ada.

GRAPHITE is intended to foster the development of objects that are attributed graphs [8]. It is becoming clear that such graphs are major building blocks of an environment. For example, Arcadia-1 uses attributed graphs to represent programs written in Ada and Ada-like languages (see Section 3.2). GRAPHITE consists of a language, called GDL, for specifying classes of attributed graphs and a processor for automatically generating abstract data types in Ada that are implementations of the specified classes. One of the significant contributions of GRAPHITE is its innovative method for supporting rapid prototyping within a statically-typed language such as Ada. In particular, GRAPHITE makes it possible for different engineers to experiment with definitions of attributed-graph objects without forcing the recoding or, in most cases, even the recompilation of tools that access the redefined object but do not explicitly use the changed information.

3.2 Basic Capabilities

Because Arcadia-1 is to contain tools suitable for an Ada software development environment, we have spent some effort in designing and building a basic set of capabilities for handling programs written in Ada and Ada-like languages. (An example of an Ada-like language is PIC/ADL, which is a design language based on Ada and intended for use in Ada environments [17].) These basic capabilities include compiler/interpreter front

ends, static semantic analyzers, and pretty printers.[2] Moreover, they include a unifying internal representation for programs written in Ada and Ada-like languages, called IRIS [13].

IRIS is a class of attributed graphs (specified using GDL) that contains information gained from static semantic analysis—static semantic analyzers transform the output of the front ends into IRIS graphs. IRIS is characterized by a consistent conceptual model, which results in a remarkable minimization of special-case processing by tools. The conceptual model is that of applying operators to operands. All user-defined program entities are described in this manner, as are all the primitive features of Ada. For example, Ada's if-statement is treated as an operator on four operands; the operands represent the conditional expression, the then-part statement, a list of else-if branches, and the else-part statement. Each operand is in turn an operator applied to a set of operands. The nodes in an IRIS graph reflect the consistency of the conceptual model by having a uniform structure; they all have an attribute representing an operator and some number of attributes representing operands. These attributes are actually references to other nodes; the nodes represent the declarations of the operator and operands, providing the semantics for the program entity being represented. IRIS graphs "bottom out" at so-called literal nodes, which represent such things as identifiers and string literals. It should be noted that the consistency of this model is in strong contrast to that of DIANA, which is another proposed internal representation for Ada programs. DIANA graphs are built from a large number of special-purpose nodes, which leads to significant complications in the code of tools that manipulate those graphs. The approach taken in IRIS avoids such complications by effectively reducing the kinds of nodes needed to represent an Ada program to two.

In formulating these basic capabilities for Ada, we have learned some important lessons about how to build flexible representations that admit to a variety of analyses and flexible tools that can be applied to programs written in a variety of languages. This experience will be put to good use as Arcadia prototypes are extended into other domains.

---

[2] We do not plan to develop our own editors and code generators, but rather to import and integrate existing ones into the environment.

3.3 Extended Capabilities

From the perspective of the engineers and managers that will use the results of the Arcadia project in building actual systems, the most important tools are those that constitute the extended capabilities. These tools are centered around the notion that the key to successful development and maintenance of large, complex software systems is the ability to obtain accurate, detailed, timely, and meaningful information about the state of the project—the correctness of its software and the productivity of its work force—throughout the lifetime of that project.

The members of the Arcadia consortium are working on a wide variety of such analysis tools. Following is only a partial list of current research efforts; detailed information about them can be found in the cited papers.

- Formal specification techniques [10]
- Formal design techniques for concurrent systems [1]
- Precise module-interface control [17]
- Static analysis of sequencing constraints [11]
- Static analysis of concurrent programs [19]
- Rigorous and systematic testing [7]
- Flexible interpretation [9]
- High-level debugging of distributed systems [3]
- Debugging of concurrent programs [6]
- Software metrics [2]
- Software productivity [5]

It is important to reiterate that a major goal of the Arcadia project is to develop a research platform flexible enough to "capture" new and diverse environment capabilities. These capabilities include not only the tools resulting from our own research efforts, but those formulated by others

as well. Once this platform reaches a reasonably stable state, we plan to actively solicit contributions from outside the consortium. We believe that the diversity of tools already being put forth by members of the Arcadia project will lead us to build a platform that can truly satisfy that goal.

Acknowledgements

The principal academic and industrial members of the Arcadia project are Richard Taylor (UCI), Leon Osterweil (CU), Lori Clarke and Jack Wileden (UMass), David Luckham (SU), David Fisher (ISC), Frank Belz (TRW), and Fredrick Cowan (TAC).

Many people associated with the consortium's institutions have contributed considerably to the Arcadia project. From UCI: R. Selby, C. Snider, C. Kelly, I. Shy, S. Sykes, R. Schmalz, T. Nguyen. From CU: D. Heimbigner, K. Olender, S. Sutton. From UMass: S. Zeil, D. Richardson, G. Barbanis, M. Burdick, E. Epp, P. Tarr. From SU: W. Tracz. From ISC: D. Baker. From TRW: B. Boehm, M. Penedo. From TAC: A. Brindle, C. LeDoux, D. Martin.

The Arcadia project has also benefited from the ongoing guidance and encouragement of Stephen Squires.

REFERENCES

[1] 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 Trans. on Software Engineering**, SE-12, no. 2, February 1986, pp. 278–292.

[2] V.R. Basili, R.W. Selby, and D.H. Hutchens, *Experimentation in Software Engineering*, **IEEE Trans. on Software Engineering** (to appear), 1986.

[3] P. Bates and J.C. Wileden, *High-level Debugging of Distributed Systems: The Behavioral Abstraction Approach*, **Journal of Systems and Software**, 3, 1983, pp. 255–264.

[4] B.W. Boehm, *A Spiral Model of Software Development and Enhancement, Proc. Inter. Workshop on the Software Process and Software Environments*, appearing in **ACM SIGSOFT Software Engineering Notes**, vol. 11, no. 4, August 1986, pp. 14–24.

[5] B.W. Boehm, M. Penedo, E. Stuckle, R. Williams, and A. Pyster, *A Software Development Environment for Improving Productivity*, **IEEE Computer**, vol 17, no. 6, June 1984, pp. 30–42.

[6] A.F. Brindle, R.N. Taylor, and D.F. Martin, *A Debugger for Ada Tasking*, **IEEE Trans. on Software Engineering** (to appear), 1987.

[7] L.A. Clarke, A. Podgurski, D.J. Richardson, and S.J. Zeil, *A Comparison of Data Flow Path Selection Criteria*, **Proc. Eighth Inter. Conf. on Software Engineering**, London, August 1985, pp. 244–251.

[8] L.A. Clarke, J.C. Wileden, and A.L. Wolf, GRAPHITE: *A Meta-tool for Ada Environment Development*, **Proc. IEEE Computer Society Second Inter. Conf. on Ada Applications and Environments**, Miami Beach, Florida, IEEE Computer Society Press, April 1986, pp. 81–90.

[9] E.C. Epp and S.J. Zeil, *Ada Interpretation in a Tool-fragment Environment*, **Technical Report 86–57**, Computer and Information Science

Department, University of Massachusetts, Amherst, Massachusetts (submitted for publication), November 1986.

[10] D.C. Luckham and F.W. von Henke, *An Overview of* ANNA, *A Specification Language for Ada*, **IEEE Software**, vol. 2, no. 2, March 1985, pp. 9–22.

[11] K.M. Olender and L.J. Osterweil, *Specification and Static Analysis of Sequencing Constraints in Software*, **Proc. Workshop on Software Testing**, Banff, Canada, IEEE Computer Society Press, July 1986, pp. 14–22.

[12] L.J. Osterweil, *Software Process Interpretation and Software Environments*, **Technical Report CU–CS–324–86**, Department of Computer Science, University of Colorado, Boulder, Colorado, April 1986.

[13] S.D. Sykes, I. Shy, D. Fisher, and R.N. Taylor, IRIS: *An Internal Form for Ada*, **Arcadia Research Report**, Department of Information and Computer Science, University of California, Irvine, California, 1986.

[14] R.N. Taylor, L.A. Clarke, L.J. Osterweil, J.C. Wileden, and M. Young, *Arcadia: A Software Development Environment Research Project*, **Proc. IEEE Computer Society Second Inter. Conf. on Ada Applications and Environments**, Miami Beach, Florida, IEEE Computer Society Press, April 1986.

[15] R.N. Taylor, L.A. Clarke, L.J. Osterweil, R.W. Selby, J.C. Wileden, A.L. Wolf, and M. Young, *Arcadia: A Software Development Environment Research Project*, **Arcadia Research Report**, Department of Information and Computer Science, University of California, Irvine, California (in preparation for journal submission), 1986.

[16] J.C. Wileden and M. Dowson (editors), *Proc. Inter. Workshop on the Software Process and Software Environments*, appearing in **ACM SIGSOFT Software Engineering Notes**, vol. 11, no. 4, August 1986.

[17] A.L. Wolf, L.A. Clarke, and J.C. Wileden, *The AdaPIC Toolset: Supporting Interface Control and Analysis Throughout the Software Development Process*, **IEEE Trans. on Software Engineering** (to appear), 1987.

[18] M. Young and R.N. Taylor, *User Interface Facilities for Software Environments*, **Arcadia Research Report**, Department of Information and Computer Science, University of California, Irvine, California, April 1986.

[19] M. Young and R.N. Taylor, *Combining Static Concurrency Analysis with Symbolic Execution*, **Proc. Workshop on Software Testing**, Banff, Canada, IEEE Computer Society Press, July 1986, pp. 170–178.