Issues Encountered in Building a Flexible Software Development Environment

Lessons from the Arcadia Project

R. Kadia

Abstract

This paper presents some of the more significant technical lessons that the Arcadia project has learned about developing effective software development environments. The principal components of the Arcadia-1 architecture are capabilities for process definition and execution, object management, user interface development and management, measurement and evaluation, language processing, and analysis and testing. In simultaneously and cooperatively developing solutions in these areas we learned several key lessons. Among them: the need to combine and apply heterogenous componentry, multiple techniques for developing components, the pervasive need for rich type models, the need for supporting dynamism (and at what granularity), the role and value of concurrency, and the role and various forms of event-based control integration mechanisms. These lessons are explored in the paper.

1 Introduction

The Arcadia project goal has been to carry out validated research on software development environments. This research has stressed development of advanced prototypes to demonstrate concept feasibility and to demonstrate integration of these capabilities into an operational whole. Integrating the various Arcadia components has been an important forcing function, compelling consideration of how environment architecture issues and usage contexts impact the various individual components.

This material is based upon work sponsored by the Defense Advanced Research Projects Agency under Grants MDA972-91-J-1009, MDA972-91-J-1010 and MDA972-91-J-1012. The content does not necessarily reflect the position or the policy of the U.S. Government, and no official endorsement should be inferred.

ACM-SDE-12/92/VA, USA

© 1992 ACM 0-89791-555-0/92/0012/0169...$1.50

This paper presents some of the insights we have gained while building and experimenting with these components. We begin by briefly describing our goals and the principal components of Arcadia. In Sections 4 through 8 we discuss several key lessons that seem to have general applicability to a wide range of environment efforts. Many of these lessons were learned by repeatedly encountering similar problems and devising similar solutions in diverse technical areas. Finally, we summarize our lessons.

2 Arcadia Overview

Arcadia believes an effective software development environment (SDE) is a collection of capabilities effectively integrated to support software developers and managers. For us, to be *effective* an SDE must be: extensible, incrementally improvable, flexible, fast, and efficient. Its components must be interoperable, it must be able to support multiple users and user classes, it must be easy to use, able to support effective product and process visibility, able to support effective management control, and it should be pro-active.

Through the years, Arcadia has evolved an architecture that addresses these objectives simultaneously. We have learned, however, that these various design objectives are not orthogonal and often conflict. Much of the most challenging work of Arcadia has been concerned with understanding the various tensions between these diverse desiderata and devising strategies for supporting adjustable compromises between conflicting SDE objectives. The focus of this paper is on the tensions that arose and the lessons we learned in our attempts to alleviate these tensions.

Arcadia's architecture is the result of our efforts to simultaneously achieve all of the above objectives in the presence of the various tensions. Several of the devices used to mediate these tensions are described later in this paper. In this section we briefly describe the principal components that form the basis for our architecture and indicate why we believe they are important to the structure of any SDE that attempts to meet the

above-enumerated objectives.

2.1 Process Definition and Execution

The needs for flexibility, extensibility, and visibility into project and product status are addressed by Arcadia's capability for developing software development processes in an explicit form and for supporting their execution. Process models show how people, tools, and software systems are used to address process requirements, and process code then implements these models. Various Arcadia environment components support development and execution of the code. The need to flexibly meet changing requirements is met by Arcadia's ability to support change to process models and code.

2.2 Object Management

Arcadia generally treats software objects as instances of abstract data types. Objects can be used locally by an individual process or shared by processes. Object management facilities must provide for persistence, type integrity, interoperability, constraint maintenance, and multi-access mediation. These capabilities facilitate visibility into project and product status and provide support for multiple users and for efficient use of computer resources.

2.3 User Interface Development

To support multiple, coordinated depictions of objects and to provide flexibility in meeting changing user needs, Arcadia supports the development of custom tailored user interfaces. Graphical presentations and effective interaction mechanisms are essential. Moreover, they must be readily alterable. Accordingly, Arcadia provides a user interface development system (UIDS) designed for the rapid alteration and enhancement of user interfaces.

2.4 Measurement and Evaluation

Continuous improvement of product and process qualities is a primary underlying objective of Arcadia. Demonstrable improvement requires quantifiable measures of these qualities. Thus, Arcadia incorporates a system for taking static and dynamic measurements of software processes and the products that they build. As software process measurement and evaluation is still a young discipline, there is little agreement on metrics for guiding improvement. Thus Arcadia's measurement and evaluation system is flexible and adaptable to changes in measurement and evaluation requirements and approaches.

2.5 Language Processing

Software products and processes contain many components, expressed in a variety of languages. An environment must recognize, analyze, and support these languages. This is most effectively done through general, tailorable language processing capabilities. In addition, uniform, language-independent representations of each language facilitate processing and analysis of these languages.

2.6 Analysis and Testing

The need to assure high quality software products and processes requires Arcadia to provide facilities for analysis and testing. Arcadia takes a broad view of what constitutes a software product, construing products to consist of a wide variety of types of artifacts. Also, Arcadia does not believe that quality is a monolithic notion, but rather that superiority with respect to a variety of quality attributes is desirable. Thus a correspondingly diverse set of testing and analysis tools is provided.

2.7 Component Composition

Our work indicates that there is a fundamental tension in environment design between the need for effective interoperability and the need for flexibility and extensibility. Tools must communicate about details of their activities, yet the suite of such tools must change. Arcadia incorporates component composition mechanisms to help strike compromises between these needs. Arcadia is exploring interprocess communication capabilities that enable the synthesis of higher level tools and processes out of lower level components implemented as separately executing programs, possibly written in different languages and/or executing on different platforms.

2.8 Summary

We believe all of the above capabilities are desirable in a contemporary environment, but we also acknowledge that different environment projects may weight the desirability of these capabilities differently. Thus, the previously enumerated seven capabilities should be selectable and combinable arbitrarily.

It is important to note, however, that any nontrivial subset of these seven capabilities cannot be expected to be smoothly integrated into an environment unless plans to do so have been made in advance, and unless the basic supports for these capabilities are firmly rooted in environment infrastructure elements.

In our efforts to investigate these seven capabilities and their integration, we have gained insight into their requirements, limitations, benefits, and their interactions. Some lessons have involved general observations

about environments and issues that arise in implementing environments. While some of these issues may not seem surprising, the extent of their impact sometimes was. In the remainder of this paper, we describe the cross-cutting issues that arose, not from the development of one capability or one project, but from our collective experience.

3 Heterogeneity

Our early concept of an Arcadia environment implicitly assumed relatively homogeneous approaches and infrastructure components. We rapidly discovered, however, that the range of issues faced by environment builders is so diverse that it exceeds the capabilities of available or expectable infrastructure components. Accordingly, Arcadia has been designed to facilitate the synergistic application of heterogeneous componentry. This heterogeneity is found across a surprising variety of infrastructure components, ranging from user capabilities such as artist-based visualizations to object management regimes. The following paragraphs briefly outline three areas in which the need for accommodating heterogeneity is apparent. Our techniques for accomplishing this are discussed in subsequent sections.

3.1 Multiple Prototypes

In order to simultaneously address open issues within the technical areas defined in Section 2, the Arcadia project has produced a wide spectrum of tools and infrastructure components for environment support. The diversity of issues addressed in each of these areas has necessitated the development of multiple prototypes embodying different solution approaches. In some cases a set of infrastructure components may be designed to provide a similar category of services, such as object management, but may have different interfaces due to fundamental differences in solution approaches or underlying models.

3.2 Multiple Languages

Software environments need to provide support for heterogeneity in programming and process languages. A software environment has a fundamental multilingual nature. It supports multiple process languages and multiple application development languages and incorporates components and infrastructure implemented in multiple languages. Environments also need to address software development-in-the-large issues that require execution and coordination across distributed platforms.

Initially the Arcadia project attempted to support a single product language and use a single implementation language (viz. Ada) exclusively so that the analysis tools developed could also be used on the environment itself. However, intrinsic shortcomings of Ada coupled with diverse needs for process and product representation and analysis resulted in our abandoning the single language approach and supporting a spectrum of language processing and communication requirements. In order to investigate process programming issues, Arcadia has developed three primary process programming languages: APPL/A [20] (with TRITON [8]), Ada/PGRAPHITE/R&R [24] [22], and TEAMWARE [26], which each address different problem areas of process representation and analysis. The implementation languages of the infrastructure components are primarily Ada and C++, with a small amount of Lisp.

3.3 Diverse Objects

Software environments also support and manage a wide spectrum of objects that vary in terms of type, persistence, granularity, etc. In Arcadia, the range of objects includes composite software products as well as their constituent parts, from requirement specifications and implementations to test cases, measurement data, and bitmaps. The operations on these objects vary in length and complexity as well as require manipulation of fine- and coarse-grain objects. Infrastructure support for this heterogeneity in objects and their operations is substantially different from the support required for queries on sets of large numbers of homogeneous objects such as is required in other problem domains.

4 Component Technology

To accommodate these demands for heterogeneity, as well as to address many other objectives, the Arcadia project has exploited a variety of techniques under the general heading of component technology. Component technology goes far beyond the recognition and definition of major components and their interfaces. It includes "wrapping" techniques, language independence, the pervasive use of generics, and meta-descriptions and translators. Our objective is to provide considerable flexibility so that environment builders, and in some cases software developers, can tailor the environment and tools to meet their specific needs.

4.1 Abstract Interfaces

A central research principle in the development of these prototypes has been the use of abstract interfaces. When possible, these interface definitions have been standardized. Interface standardization enables researchers to investigate heterogeneous solutions to similar problems, to interchange component implementations, and to facilitate environment reconfiguration.

As one example use of standardized interfaces in Arcadia, the CHIRON UIDS [10] provides a standard interface to two different look-and-feel presentations: XView and Motif. Another example is the SMI storage manager interface, which provides a standard interface to various underlying storage managers, including Exodus [5], Mneme [15], and Ada's "direct I/O" files.

On the other hand our experience has shown that creating standard interfaces is not always achievable, particularly when the underlying architectures are substantially different. A variety of other techniques, discussed below and in other sections, were developed to mitigate these situations.

4.2 Wrapping

The use of "wrapping" techniques has been useful to encapsulate unavoidable inconsistencies in interfaces. The CHIRON UIDS provides dispatchers that wrap arbitrary object ADT-interfaces. One key purpose of the dispatcher is to provide a common structure and interface to the ADTs for artists (which create visual depictions of the ADTs) to use. The AMADEUS measurement system [19] provides a script specification language for wrapping data collection tools and defining their conditions for execution. The script language provides the event monitoring subsystem of Amadeus with a consistent notation for describing these tools, which may have varied interfaces. Both the Chiron and Amadeus wrappers use event-based notification techniques, a topic which is considered more fully in Section 8.

4.3 Language Independent Common Representations

Analysis techniques have typically been applied to a small number of well recognized internal representations, such as abstract syntax graphs, control flow graphs, and call graphs. In the Arcadia project we have long recognized the need to agree on the interfaces to such abstractions so that, once created, these objects can be shared by many different analysis tools. Arcadia had the additional requirement that the representations be language-independent, so that the tools would be applicable to multiple, although somewhat related, languages. Our choice of an abstract syntax graph reflects this requirement.

We chose an IRIS-based abstract syntax graph [2] and negotiated an interface to the abstraction that supported creating and accessing the nodes, traversing the graph, and making graph instances persistent. The IRIS representation had several advantages. It is a language-independent representation where some aspects of the static semantics of the language are captured by a language description. This language description is also represented as an IRIS structure. Once a source program is translated into IRIS there is no distinction between a user-defined operator (e.g., pop) and a language-defined operator (e.g., +). This makes it easy to define new languages or extend or modify existing languages. Also, if tools judiciously use the description of the language instead of hardcoding any information about the language, they too will be language independent[1].

The control flow graph, call graph, family of program dependence graphs, and task interaction graph [12] are other examples of shared internal representations with negotiated interfaces.

4.4 Generics

Another way that reusability and flexibility have been achieved is by the use of generics to instantiate general purpose components with specific types. The Control Flow Graph Generator, for example, is a language-independent tool that takes a description of the language and associated instructions about how to build a control flow graph for that language and generates an abstract data type that is used to instantiate a generic control flow graph build procedure. The same generic can be instantiated to build a control flow graph or a call graph for a language, just by altering the instructions on how to treat different operators of the language. The PRODAGI program dependence graph generator [14] is a tool that constructs several packages of generic instantiations for a given dependence relation and associated build procedure; the instantiated packages provide an interface to manipulate and maintain the dependence relation.

4.5 Translators

Perhaps one of the most prevalent technologies employed in the Arcadia project is the use of translators, such as preprocessors and generators, that are often driven by specialized meta-languages.

As described in section 5, we have extended the type model and functionality of Ada, our primary implementation language. These extensions have usually been accomplished via the use of syntactic language extensions that are preprocessed into Ada. Examples include APPL/A, a process programming language that adds extended relational data base capabilities to Ada; PIC [25], which adds module interconnection commands; PGRAPHITE, R&R, and REPOMANGEN, which together add graph, relation, and relationship types and persistence to the language. CHIRON extensively employs preprocessor technology to generate artist templates, client managers, and dispatchers, among other things.

[1] Cedar [21] earlier recognized the need to be able to ask the same kinds of questions about predefined operators as user-defined operators. The Cedar abstract machine provided such a capability.

Development of these processors has been supported by several translator building tools, which have made this approach viable for us.

4.6 Component Composition

Language heterogeneity, the advantages of a distributed systems architecture, reuse considerations, and other reasons have required development of mechanisms to support component composition. Three kinds of mechanisms have played important roles in Arcadia: (a) interprocess communication providing representation-level interoperability between processes, (b) multiple language bindings to services, and (c) automated development of interfaces providing type translation between programs in different languages.

An interprocess communication mechanism, called Q [13], has been developed based on Sun RPC/XDR and has been used to connect Ada and C programs. Distributed process execution is enabled through the Q IPC mechanism as well as through direct use of Unix IPC mechanisms such as RPC and sockets. Q is a generic that is instantiated with information about the representation of types and how they are to be encoded and decoded before and after transmission.

Many components provide a layer of language bindings to support inter-language component composition. CHIRON provides a language extension to Ada whose translation enables (remote) use of a library of C++ gadgets. The TRITON OMS provides an Ada binding to the underlying Exodus storage manager written in a C++ extension. The AMADEUS measurement system provides Ada, C, and Unix shell script language bindings to its underlying measurement capabilities.

The Specification Level Interoperability (SLI) mechanism [23] attempts to automate the translation from one type model to another at a higher level of abstraction than is usually employed. In this approach, type declarations from programs written in different languages are translated into a common representation, called a unifying type model (UTM). In the current SLI prototype, this common representation is based on the OROS type model [18]. When a program in one language needs to make use of a type defined in another language, an interface to the defined type in the requested programming language can sometimes be automatically generated based on the representation in the common representation. For example, if an Ada program needed to call an existing C abstraction, the interoperability mechanism would use the common internal representation, first to help find the compatible type and then to create an Ada module implemented using the C abstraction.

5 Type Models

Our work in Arcadia has repeatedly revealed the need for rich type models to support software engineering. Features of such type models include expressive type definitions for the most frequently used software engineering abstractions, uniform treatment of entities as first-class objects, polymorphism and inheritance, and appropriate support to manage persistence, consistency, and concurrency.

To satisfy these needs, Arcadia has looked to both database systems and programming languages, but has found the type models typically provided in these disciplinary areas to be lacking. The database community has recently acknowledged many of the deficiencies of database models [3, 27, 1] for software engineering. In particular, the type models of database systems tend to be too simplistic. For example, traditional database systems fail to provide the basic building blocks to support graph objects, a pervasive type in software engineering, and the typical navigational operations needed on such structures. There has been recent work on enhancing database type models so that they provide a richer set of base types and some support for inheritance, but these initial attempts, although promising, are still too restrictive.

Programming languages tend to offer richer type models than database systems in terms of constructors and primitive types, but fail to support relationship and relation types. Moreover, programming languages offer only limited support for persistence, consistency, and concurrency control, all of which tend to be supported by database systems. Some recent languages have attempted to provide support for polymorphism and multiple inheritance, but this support has been limited. Finally, neither database nor programming language type models treat most higher-level entities, such as types and operations, as first-class citizens.

The remainder of this section describes the kind of support we found that we needed and some of our efforts to address these needs.

5.1 Type Model Extensions

Many of the objects manipulated by software engineering environments are graphs, such as abstract syntax graphs/trees, control flow graphs, or call graphs. Relationships (n-ary tuples) and relations (collections of relationships) are also ubiquitous types in environments. For example, software developers might want to maintain relationships between abstract syntax trees and their corresponding control flow graphs. In addition, software developers might want to ask questions about these relationships. For example, one might want to know which nodes in an abstract syntax graph actually describe the invocations that are captured in the call graph representation. It is clearly more convenient, re-

liable, and efficient to have built-in support for these abstractions than to have individual programmers developing their own models and implementations. Thus, as part of the Arcadia project we have extended Ada's type model to support graphs, relations, and relationships as though they were primitive types of the language.

In the GRAPHITE system [6], and its successor PGRAPHITE, a model of directed graphs is provided that is a natural extension to the Ada type model. The graph abstraction includes operations for creating and manipulating nodes of the graph, as well as graph operations such as various forms of traversal. As an indication of the pervasiveness of graph objects in software development environments, GRAPHITE and PGRAPHITE have been used in over a dozen tools within Arcadia.

APPL/A has extended the Ada type model to support relations and provides operations for creating and manipulating those relations. The R&R system was modeled after APPL/A but moved farther away from the traditional database model of relations by allowing relationships to be first class objects. This means that a relationship can be shared by more than one relation, a concept that has proven very useful to the analysis tools.

Both APPL/A and R&R provide some support for simple queries. Neither of these systems provides the type of support for complex and ad hoc queries that we feel is truly needed. Moreover, there needs to be better support for both navigation and queries over an object, including the ability to access the same object through both types of operations when appropriate.

5.2 First Class Citizenship

In our work on Arcadia we have frequently been frustrated by our inability to treat key programming language entities, such as types, tasks, and operations, as first class citizens. The inability to manipulate these objects, as we would any other object in a language, limits the flexibility of the components we are developing. For example, we would like to be able to pass operations and tasks as parameters. Without such support we have had to use preprocessors to generate low level code to get around these limitations. In CHIRON, for instance, when an operation of a depicted abstract data type is called, the corresponding artist operation must be called for each active artist associated with the object. Maintaining a list of active artists and then passing each package and operation that must be invoked would be a straightforward way to handle this. Unfortunately, as in most programming languages, packages and tasks are not first class citizens in Ada; thus, this can not be done. There are numerous examples in Arcadia where such flexibility would be beneficial.

5.3 Polymorphism and Multiple Inheritance

A type model that supports polymorphism and inheritance is also desirable. Such a typing model clearly reduces the burden on the programmer when designing and coding complex systems. For example, Ada's polymorphism mechanism (generics) is not sufficiently powerful to support specification of a variable number of generic formal parameters. We have mimicked these facilities through the use of generator/preprocessor technology instead, e.g. in CHIRON and TRITON.

5.4 Consistency

Being able to control consistency is important when dealing with many different abstractions that may be related in complex ways. It can be argued that exception handling mechanisms provide a limited form of consistency control, depending on their recovery model. A more general model of consistency is desired where programmers can define arbitrary constraints that will cause an operation to be triggered if the constraint is violated. We have also found that it is desirable to allow programs to dynamically control constraint enforcement.

Although not currently implemented, APPL/A's definition supports the definition of constraints over relations. If a constraint is violated, a system defined exception is raised. The R&R system also defines a constraint mechanism. Currently R&R constraints can only be defined over relations and relationships but the intent is to extend the applicability to objects of any type. When a constraint is violated, a user defined operation is triggered. If no such operation is provided, a system defined default is triggered instead. Both APPL/A's and R&R's constraint constructs are programming language-level representations for the underlying event-based control integration mechanisms described in Section 8.

Constraints and triggers provide a powerful mechanism for achieving interoperability between abstractions that were not originally designed to coordinate their activities. This was our experience in implementing a demonstration that involved tools originally designed independently of each other. It can be a dangerous programming style, however, if not judiciously applied since it relies on side effects that can lead to circular dependencies.

6 Dynamic Definition and Access

Various forms of dynamism are essential in enabling the evolution of the Arcadia environment while it is in operation and being used for productive work. As with heterogeneity, Arcadia has struggled with the issue of

dynamic definition and access with regard to a variety of objects in the environment. Of course, any interesting environment supports large classes of dynamically defined objects, along with some means for accessing those objects. But there is also a significant portion of any environment that is not easily evolvable. This induces a qualitative measure of "granularity" for dynamism. The degree of granularity is a measure of the effort needed to make a change that is visible to the environment.

Our initial emphasis on using Ada caused us to choose program recompilation as our unit of granularity. That is, many changes were not visible until dependent program units were re-compiled and re-instantiated in the environment. Over time, we have come to recognize that this level of granularity is not sufficient; in some situations, it is undesirable to recompile some collection of programs in order to effect changes. Rather, we are pursuing more interpretive approaches where changes can be immediately exported into the environment.

This section addresses two main issues: identifying the objects that can change dynamically and defining the granularity of dynamic changes.

6.1 Sources of Dynamism in Arcadia

Object management is one obvious source of dynamic change in any environment. By definition, an object manager allows for the dynamic creation and manipulation of a variety of objects: application objects and indices, for example. Here, and subject to transaction semantics, changes to objects can quickly be made visible.

A surprising number of object managers do not allow immediate changes to the schema, including even the addition of new types, without significant delays. For example, one of our initial object management systems, CACTIS [9], used recompilation in order to implement dynamic changes to schemas.

Our current object management systems, which are PGRAPHITE and TRITON, both provide substantial improvements in dynamic schema management. We recognize the difficulties around general schema changes [11], and in both PGRAPHITE and TRITON we have chosen to provide a level of support for dynamic schema changes that is useful without being comprehensive.

PGRAPHITE supports a structural object-oriented model in which the structure of objects is embedded into the accessing programs but provides a level of interpretation in accessing fields to these objects. As a result, one can extend objects to include new fields without having to re-compile accessing programs. Of course, deletion of fields or changes in the semantics of existing fields require tracking and re-compiling programs that depend on the modified fields.

TRITON provides a behavioral object-oriented model derived from the E [17] type model. TRITON embeds this model in a client-server architecture that supports certain kinds of immediate augmentation for recorded schemas. Specifically, it allows for the dynamic definition of functions, methods, classes, and triggers and deletion of the same. Changes require deletion followed by (re-)definition.

Dynamic procedure definition is provided by a facility for dynamically loading the code for methods and triggers, and by providing mechanisms by which clients of TRITON can invoke those newly defined methods with relatively low overhead. Addition of new structural fields, something that is easy in PGRAPHITE, is difficult in TRITON since that information is encoded into the compiled method code. TRITON also provides for the deletion of schema elements, although we are not sure that we have the appropriate deletion semantics. In order to address dynamism in the presence of multiple models, we have developed the A LA CARTE heterogeneous data management system [7]. A LA CARTE presents tools for incrementally integrating multiple object managers at various levels of processing, such as physical object management, transaction management, and data modeling.

We recognize that there are dependencies between the TRITON schema and the client programs that use it. So some schema changes will require (at least) re-compiling dependent clients. It is possible to write clients that obtain sufficient schema information at execution time to react to changes. Type and instance browser programs are often written in this fashion.

User interface is another area where dynamic operation is desirable, and the CHIRON UIDS has evolved in the direction of increasing dynamism. In its first version (CHIRON-0), artist procedures had to be compiled into the application code and there was a limit of one artist per abstract data type(ADT). In CHIRON-1.0, the artists still had to be compiled with the application, but the single artist restriction was removed by the use of an event dispatcher per ADT. Multiple artists may register interest in the same ADT. Very recent changes in CHIRON have opened up the possibility (as yet unexploited) for adding and deleting artists on-the-fly without the need for recompilation.

Evaluation (in the form of AMADEUS) is both a producer and consumer of dynamism in Arcadia. As a consumer, it requires an ability to dynamically insert measurement probes into various processes in the environment without disrupting those processes. As a producer, AMADEUS uses its event mechanism to dynamically define and activate scripts that can process collected measurements.

6.2 Mechanisms Supporting Dynamism

Arcadia provides a variety of mechanisms supporting dynamism at a level of granularity requiring recompilation. Without being exhaustive, we can single out three mechanisms that directly support more rapid dynamic changes in Arcadia: events, dynamic loading, and client-server architectures.

As described in section 8, the various event dispatchers in Arcadia allow for dynamic specification of procedures (or scripts or triggers) to be invoked whenever certain events occur. In effect, we have an anonymous invocation mechanism in which the recipients to be invoked can be defined and changed in a highly dynamic fashion.

TRITON supports dynamic loading of code into the TRITON server along with simple mechanisms for invoking that dynamically loaded code. This is currently only usable for object methods and for triggers, but the basic mechanism has potential applications in other components such as the CHIRON server where it could be used to extend its functionality on-the-fly.

The whole client-server apparatus in Arcadia (Q) provides a significant degree of dynamism. It allows a client program to invoke an (almost) arbitrary server program and to decide, on-the-fly, the particular server and operation within the service in which it is interested. CHIRON currently exploits this capability. This gives a rather different flavor to Arcadia compared to, for example, Cedar [21], which had the mixed blessing of a single address space where it was possible to dynamically bind code, but in so doing, inter-component dependencies were lost that made it difficult to unbind components.

6.3 Costs

We are aware that the use of dynamism, especially structural dynamism, does not come free. Often it is difficult to type-check with a compiler, or analyze with various tools. This introduces the possibility of run-time errors, which are the most expensive ones to find. When errors occur, they may be time or context-dependent, and the programmer may not even be able to capture the context. These difficulties should not be read as arguments against dynamism so much as they are arguments for using caution when it is introduced and for providing appropriate support to make its use as safe as possible.

7 Concurrency

There are a wide variety of reasons for supporting concurrency in a software development environment. For example, Arcadia has found that all too many SDEs are implicitly developed for single users. Our goal of providing a pro-active, heterogeneous environment for multiple users and multiple classes of users demanded that we use languages and system programming mechanisms that effectively support concurrency.

Some of the many demands for concurrency in Arcadia are as follows. With regard to user interfaces we recognize that both users and tools may be simultaneously active and in need of periodic communication with each other. Consequently the UIDS must not preordain one or the other to be "in charge" and must therefore exhibit a concurrent control model. CHIRON is designed to support multiple users working cooperatively, such as through multi-view editing sessions. Similarly, the process programming mechanisms in Arcadia are designed to orchestrate the actions of multiple, concurrent users, cooperating on tasks such as creating a requirements specification. These process mechanisms must enable specification and enaction of cooperating concurrent activities. This objective has further consequences, requiring, for example, support for concurrency controls on shared data. We have also found that many process components are effectively expressed as reactive control units, which can best be described logically using formalisms involving concurrency. Effectively supporting flexible measurement and evaluation of processes also demands concurrency: monitoring and analysis activities should, in many cases, take place transparently, unobtrusively, and simultaneously with the monitored process. Concurrent mechanisms in the AMADEUS system support this non-interference approach. It is also clear, in all the above situations as well as others, that performance benefits can be achieved through the programming of concurrent activities.

Though not as obvious and stringent a demand for concurrency, the heterogeneous systems approach exhibited by the Arcadia environment is also perhaps best supported by distributed computation. For example, use of some tool may require execution on a particular hardware platform, distinct from the primary platform of the environment. Integration of that tool into a process would likely require distributed systems support, and would bring the potential for concurrency along at the same time.

There are three practical consequences of the needs for concurrency in Arcadia.

First, environment components supporting requirements wherein concurrency is a key part benefit from being programmed in a language which has effective constructs for using and controlling concurrency. This reduces the burden on the developer (as compared to simulating the concurrency in a sequential language) and keeps the implementation cleaner. On the other hand, for various reasons one does not always have this capability and therefore a fall-back strategy must be provided. Use of operating system capabilities from a sequential language is a natural resort. (This often leads

to the creation of heavyweight processes with separate address spaces, which has other advantages discussed below.)

Second, the presence of concurrency presents many well-known challenges to developers: developing bug-free concurrent code is notoriously difficult. As a result, we found that our development benefited substantially from the use of good formal analysis. For example, the CHIRON system, consisting of approximately twenty Ada tasks in two Unix processes, was analyzed by the CATS analyzer (which performs a type of reachability analysis and temporal logic checking). Two race conditions and a deadlock possibility were thus identified and subsequently removed.

Third, because many tools do not anticipate working in a concurrent world, it is often necessary to adopt a defensive implementation strategy that protects tools from each other. For example, one version of the CHIRON system makes use of the XView user interface toolkit and processor. Since XView assumes a sequential control model, while CHIRON is concurrent, it was necessary to place them in separate address spaces, since the XView notifier destroyed the Unix signals used by the Ada (tasking) run-time system. More generally, the basic solution is to guarantee separate resources for each run-time system, including signals, file-descriptors, and possible heap memory. Heap memory does imply separate address spaces, while technically signals and descriptors do not (though on Unix they do.)

The use of separate address spaces also contributes, in an incidental fashion, to addressing several other issues, including strongly controlled interfaces and multi-language issues. Multi-language issues should be clear from the preceding paragraph. The issue with strongly-controlled interfaces is that, in the absence of effective language constructs and analyzers that guarantee that design-level interface rules are not violated, placing service providers and service requesters in separate address spaces yields an effective operational way of ensuring conformance to the interface rules. That is, "thin-wire" communication helps enforce the abstractions.

8 Event-based Control Integration Mechanisms

To facilitate the sorts of highly flexible control flow necessary in the face of rapid change in the structure and componentry of an environment, we found event-based control integration mechanisms to be broadly useful. For example, event-based control has been used from statistics gathering, to enforcing constraints, to maintaining consistency between multiple simultaneous graphical views of objects. In this section, we will first describe the various purposes served by the event-based mechanisms. The details of the mechanisms will then be contrasted.

The common objective of all the mechanisms discussed is the combination ("integration") of separate components to perform desired services. Such mechanisms are necessary when no single component suffices to perform the service and the necessary compositions of components are unpredictable. Moreover, an extensible mechanism is needed, so that new components can be added in a convenient, and often dynamic, manner. "Components" in the systems above include, for example, artists (CHIRON), data analysis agents (AMADEUS), and data-constraint maintenance programs (APPL/A and R&R).

In CHIRON, dispatcher events are accesses to the interface functions of abstract data type (ADT) instances (Ada packages, in the primary implementation). These events and the supporting mechanism are used to provide dynamic, non-invasive coordination of tools and artists, where artists are code units that provide customized graphical depictions of the state of the ADT instance. Among the functionality provided is simultaneous updates of all views of an object, regardless of which views are used as editing interfaces.

In AMADEUS, an event is a fundamental abstraction on which process or product measurement and evaluation is based. (Events may be aggregated and analyzed to yield insight into products or processes.) The purpose of AMADEUS' event-based integration mechanism is to detect events and enter them into the measurement and evaluation framework, possibly resulting in changes to components in the environment.

Events in APPL/A are operations on relations. Their purpose is to allow automatic, reactive response to operations on relations. Response to a change to a file, for example, might be to issue a warning message that a configuration was out of date and to initiate recompilation of other components to bring the configuration back into a consistent state.

In TRITON, events are invocations of methods or functions stored in the TRITON server. Functions can be triggered upon such invocations, and may be used to support, e.g., unobtrusive monitoring, to provide forward and backward inferencing, and to export database events to the rest of the environment.

Events in NEXUS, the event-based control mechanism underlying R&R's constraint facilities, are similar to those in CHIRON in that NEXUS events are accesses to the interface operations of ADT instances. NEXUS events are used to support consistency maintenance over shared objects (e.g., R&R relations and relationships, and graphical depictions) where changes in the state of an object might be of interest to other entities in the environment.

It should be clear that the various Arcadia event-based integration mechanisms exist and operate at different levels of abstraction for the different purposes de-

scribed above. Nevertheless, a common set of design issues can be used to highlight both similarities and differences. We first consider the notion of event types, then present several issues that arise in the context of event recognition and processing. Cross-cutting issues of dynamism and concurrency are then considered.

8.1 Event Types

Three types of events that are explicitly identified in AMADEUS cover all the event types in the Arcadia mechanisms: changes in data values, time-based events, and messages. Messages may represent (name, value) pairs or procedure/function calls (which includes operations on relations or relationships). The information content of messages, including the notion of the type system to which they belong, varies, and is discussed below. It is not clear that there is any intrinsic difference between event types; each may be modeled (and represented) different ways. An obvious but important observation is that the definition of an event is tied to a particular set of abstractions (e.g. what constitutes an event for Chiron's user interface purposes may not be an event for Nexus' constraint-managment purposes, and vice versa).

8.2 Event Occurrence and Processing

Registration. Prior to event occurrence, event emitters may register descriptions of the kinds of events they may produce. Components interested in receiving events may register their interest in certain types (or values) of messages. Filtering, or transformation, protocols may also be registered. These items may be registered several places. There may be a single central authority for handling registrations or there may be many registration agents.

Event generation. Events may be "naturally occurring" or be seeded, either manually or automatically. Events are naturally occurring in CHIRON, APPL/A-TRITON, NEXUS, and in some AMADEUS components, in that the events occur as an ordinary part of achieving some other functionality, e.g., accessing an ADT instance's access function or operating upon a relation. Events in AMADEUS may be seeded. For example, a process program may have event generation code inserted to yield the raw data needed by AMADEUS; other events monitored by AMADEUS may be natural, such as watching for a file to be edited.

Recognition. Once an event occurs it must be recognized and thereby entered into the event-processing mechanism(s). This may happen by, e.g., the event-generating component, a database mechanism, or a "watcher" — a specific tool designed for that purpose. CHIRON events are recognized and initially handled by dispatchers, which are located in the application's address space. In TRITON, trigger detection and invocation is performed as part of the TRITON interface operations that invoke methods in the server. The NEXUS server is a component separate from object management services and is capable of operation on types not described via OM technology. AMADEUS events are recognized by specialized components (AMADEUS event servers), each having its own address space.

Representation and meaning. Events may be represented with strong or weak typing, and may convey their semantics either within the message or with respect to an external (to the message) definition. CHIRON messages are strongly typed, with their type system being the type system of the application. The message consists of the name of the operation performed, the value of all arguments to the operation, and the value of any return parameters. The meaning of the message is confined to the address space in which the event occurs. Both APPL/A and TRITON messages are similar. NEXUS messages are strongly typed, where the NEXUS type system is derived from the UTM. AMADEUS messages are weakly typed, being essentially (name, value) pairs encoded as ASCII strings, the interpretation of which is conventionalized by AMADEUS.

Processing. In general, event processing may involve collection and aggregation, filtering according to registered protocols, and propagation. Propagation may occur via a single distributor per environment, per event type, per object type, or per object. Distribution may be cascaded. Notions of transactions and rollback may be present. Propagation may be either synchronous or asynchronous, and whether it is synchronous or asynchronous may depend on the level of granularity considered. CHIRON events are processed by dispatchers, where there is one dispatcher per object. Some asynchrony of processing is possible but the primary notification activity is synchronous (involving the originator of the operation on the ADT, the dispatcher, and the listening artists). AMADEUS events are filtered and distributed by event servers under the control of scripts which define the desired processing in terms of which actions to take, such as which tools or processes should be executed. There may be multiple servers per AMADEUS application, and scripts are explicitly declared to have synchronous or asynchronous processing. In NEXUS, collection and propagation are performed by the NEXUS server, where there is one server per namespace. Synchronization in NEXUS is listener-controlled; the informer specifies a maximum level of synchronization that can be supported, and the listener specifies the minimum level that is desired.

Post-processing. Subsequent to processing of events, various activities may occur, such as sending of acknowledgments, modification of filtering policies, or generation of new events. CHIRON's dispatcher synchronization/transaction policy ensures one dispatch is complete before another can begin. AMADEUS events may cause

further events to be generated and propagated which may cause additional scripts to be interpreted. Event generation may be turned off (or on) depending on the actions associated with event processing. TRITON triggers are invoked with a pre-defined interface that includes the arguments to the function or the result of the function. The trigger may perform arbitrary manipulations on that information and may even abort the actual invocation of the target method. NEXUS listeners may control the receipt of events by dynamically registering and unregistering with the server.

Cross-cutting Issues. Cutting across all the above are issues of dynamism and concurrency. Dynamism is the degree to which changes can be made to any of the definitions or mechanisms after their initial construction. Concurrency is the degree to which parts of the mechanisms may operate simultaneously. Some aspects of dynamism have already been identified. AMADEUS is the most dynamic of all the current mechanisms, as new event kinds may be generated while an Arcadia process is executing and new event servers to receive and dispatch events may be activated. Such dynamism goes hand-in-hand with AMADEUS' weakly typed message structure. The other mechanisms, more strongly typed, are less dynamic.

8.3 The Point

Though serving a wide range of purposes and though developed independently, a high degree of commonality of structure is present in the above subsystems. Tradeoffs among specific design choices are apparent. We conclude that this type of mechanism is widely adaptable and useful in the software environment context, well beyond the typical uses of early systems such as Field [16] and Softbench [4], which have been largely confined to infrequent control integration of large, monolithic (from the perspective of Field/Softbench) tools.

9 Conclusion

If we were pressed to summarize our lessons, we would have to say that Arcadia is about abstraction and flexibility in the face of the multiple tensions created by a broad set of goals and a wide variety of component technologies.

We remain convinced that appropriate use of abstraction remains a key to effective large-scale system development. All environment capabilities and artifacts (e.g., processes, operands, etc.) should be captured through disciplined used of abstraction. Further, we believe that these abstractions must not only capture functionality, but also support viewability, measurability, and persistence.

There do seem to us to be certain risks, however, in premature codification of abstractions. In the absence of a fixed structure, it is important to favor flexibility in the structuring of abstractions. That being the case, we found that it was preferable to define smaller and more general abstractions rather than larger and more specific abstractions.

As illustrated by our experience with abstractions, flexibility is the other hallmark of Arcadia. Many of the lessons we have learned involve our attempts to move in the direction of increasing flexibility. Heterogeneity, meta-languages, dynamism, events, concurrency, and powerful type systems are all driven by a requirement for flexibility.

In sum, we believe that the Arcadia project shows that it is possible to provide a system that begins to match the ambitious goals that we established for ourselves. In the process of producing such an environment, we have learned a number of lessons that, while specific to our own diverse research efforts, seem likely to be of interest and value to many environment projects outside Arcadia.

Acknowledgments

This paper represents the opinions of the principal investigators of the current university Arcadia grants: University of Massachusetts: Lori A. Clarke, Jack C. Wileden; University of California at Irvine: Leon J. Osterweil, Debra J. Richardson, Richard W. Selby, Richard N. Taylor; University of Colorado: Dennis M. Heimbigner, Roger King.

The work upon which this paper is based was the result of efforts of many people, including the following whom we would particularly like to acknowledge: Stephanie Leif Aha, Jennifer Anderson, Ken Anderson, Deborah Baker, Robert Balzer, Douglas Bell, Frank Belz, Jim Berney, Navdip Bhachech, Barry Boehm, Greg Bolcer, Billie Bozarth, Debra Brodbeck, Mary Burdick, Mary Cameron, Yidong Chen, Satish Chittamuru, Pamela Drew, Jose Duarte, Matthew Dwyer, Stuart Feldman, Joseph Fialli, Charles Fisher, David Fisher, Kari Forester, Susan Graham, Thomas Huynh, Greg James, Rajesh Jha, Takuya Katayama, Alan Kaplan, Ruedi Keller, Walter R. Kopp, Peter Lee, Barbara Lerner, David Levine, Chyun Lin, Doug Long, Dave Luckham, Craig MacFarlane, Kent Madsen, Mark Maybee, Erik Mettala, Cynthia Tittle Moore, Elliot Moss, Kurt Olender, Owen O'Malley, Lolo Penedo, Adam Porter, Ron Reimer, William Rosenblatt, Wilhelm Schafer, Bill Scherlis, John Self, Izhar Shy, Xiping Song, Craig Snider, Tom Souksamlane, Stephen Squires, Stan Sutton, Peri Tarr, Kojii Tori, Dennis Troup, Sandy Wise, Alex Wolf, Harry Yessayan, Michal Young, Patrick Young, Steven Zeil, Hadar Ziv.

References

[1] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The Object-Oriented Database System Manifesto. In *Proceedings of the First International COnference on Deductive and Object-Oriented Databases*, 1989.

[2] D. A. Baker, D. A. Fisher, and J. C. Shultis. The gardens of IRIS. Technical Report Arcadia-IncSys-88-03, Incremental Systems Corporation, August 1988. Draft.

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

[4] M. R. Cagan. The HP SoftBench environment: An architecture for a new generation of software tools. *Hewlett-Packard Journal*, 41(3):36–47, June 1990.

[5] M. J. Carey, D. H. DeWitt, D. Frank, G. Graefe, M. Muralikrishna, J. E. Richardson, and E. Shekita. The architecture of the EXODUS extensible DBMS. In *International Workshop on Object-Oriented Database Systems*, pages 52–65, 1986.

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

[7] P. Drew, R. King, and D. Heimbigner. A toolkit for the incremental implementation of heterogeneous database management systems. *Very Large Database Journal*, 1(2), 1992. To appear.

[8] D. Heimbigner. Triton Reference Manual, 1 July 1990.

[9] S. E. Hudson and R. King. The Cactis project: Database support for software environments. *IEEE Transactions on Software Engineering*, 14(6):709–719, June 1988.

[10] R. K. Keller, M. Cameron, R. N. Taylor, and D. B. Troup. User interface development and software environments: The Chiron-1 system. In *Proceedings of the Thirteenth International Conference on Software Engineering*, pages 208–218, Austin, TX, May 1991.

[11] B. S. Lerner and A. N. Habermann. Beyond Schema Evolution to Database Reorganization. In *Proceedings of the Joint ACM OOPSLA/ECOOP '90 Conference on Object-Oriented Programming: Systems, Languages, and Applications*, Ottawa, Canada, October 1990.

[12] D. L. Long and L. A. Clarke. Task interaction graphs for concurrency analysis. In *Proceedings of the Eleventh International Conference on Software Engineering*, pages 44–52, Pittsburgh, May 1989.

[13] M. Maybee, L. J. Osterweil, and S. D. Sykes. Q: A multi-lingual interprocess communications system for software environment implementation. Technical Report CU-CS-476-90, University of Colorado, Boulder, June 1990.

[14] C. T. Moore, T. O. O'Malley, D. J. Richardson, S. H. L. Aha, and D. A. Brodbeck. ProDAGI: A program dependence graph system. Technical Report UCI-91-25, Department of Information and Computer Science, University of California, 1991.

[15] J. E. B. Moss. Implementing persistence for an object oriented language. In *Proceedings of the Workshop on Persistent Object Systems: Their Design, Implementation, and Use*, Port Appin, Scotland, 25-28 August 1987.

[16] S. P. Reiss. Connecting tools using message passing in the field environment. *IEEE Software*, 7(4):57–66, July 1990.

[17] J. E. Richardson and M. J. Carey. Programming constructs for database system implementation in EXODUS. In *Proc. ACM SIGMOD Conf.*, pages 208–219, 1987.

[18] W. Rosenblatt, J. Wileden, and A. Wolf. OROS: Toward A Type Model for Software Development Environments. In *OOPSLA Conference Proceedings*, pages 297–304, October 1989. Published as *ACM SIGPLAN Notices*, 24(10).

[19] R. W. Selby, A. A. Porter, D. C. Schmidt, and J. Berney. Metric-driven analysis and feedback systems for enabling empirically guided software development. In *Proceedings of the Thirteenth International Conference on Software Engineering*, Austin, TX, May 1991.

[20] S. M. Sutton, Jr., D. Heimbigner, and L. J. Osterweil. Language constructs for managing change in process-centered environments. In *Proceedings of ACM SIGSOFT '90: Fourth Symposium on Software Development Environments*, pages 206–217, Irvine, CA, December 1990.

[21] D. Swinehart, P. Zellweger, R. Beach, and R. Hagmann. A structural view of the Cedar programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):419–490, October 1986.

[22] P. Tarr, J. Wileden, and L. Clarke. Extending and Limiting PGRAPHITE-style Persistence. In *Proceedings of the 4th International Workshop on Persistent Object Systems, Martha's Vineyard, MA*, pages 74–86, August 1990.

[23] J. Wileden, A. Wolf, W. Rosenblatt, and P. Tarr. Specification Level Interoperability. *Communications of the ACM*, 34(5):73–87, May 1991.

[24] J. C. Wileden, A. L. Wolf, C. D. Fisher, and P. L. Tarr. PGRAPHITE: An experiment in persistent typed object management. In *Proceedings of ACM SIGSOFT '88: Third Symposium on Software Development Environments*, pages 130–142, Boston, November 1988.

[25] A. Wolf, L. Clarke, and J. Wileden. The AdaPIC toolset: Supporting interface control and analysis throughout the software development process. *IEEE Transactions on Software Engineering*, 15(3):250–263, March 1989.

[26] P. S. Young and R. N. Taylor. Team-oriented process programming. Technical Report UCI-91-68, Department of Information and Computer Science, University of California, 1991.

[27] S. Zdonik and D. Maier. Fundamentals of Object-Oriented Databases. In S. Zdonik and D. Maier, editors, *Readings in Object-Oriented Data Systems*, pages 1–32. Morgan Kaufman, San Mateo, California, 1990.