Design Possibilities for Zeus: The Tool/Object Manager for Arcadia

Deborah A. Baker and Dennis Heimbigner

CU-CS-318-86 February 1986

University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Design Possibilities for Zeus:
The Tool/Object Manager for Arcadia

Deborah Baker
and
Dennis Heimbigner

Department of Computer Science
University of Colorado
Boulder, Colorado

4 February 1986

Arcadia Document: CU-86-01

Table of Contents

1. Introduction

Zeus is the software that manages tools and objects in Arcadia[1] . This report is intended to outline some of the design issues for Zeus. Some alternatives are discussed for each design issue. In each case, one of the alternatives is the Odin approach. We begin with a review of Odin to establish a common vocabulary and a common understanding of the basic Odin philosophy.

2. A Review of Odin

An understanding of Odin hinges on understanding the concept of derivation. The user must understand that some objects (in Odin, objects are synonymous with files) can be automatically constructed from other objects by using some specific tool(s). To the user, then, Odin appears to be a large database of all the possible objects that one might ever want to construct by the repeated application of some programs starting with some set of typed (atomic) objects. The principal user operation is to ask for the presentation of objects.

Assuming this, there are a number of other capabilities and properties that may be used to describe Odin:

1. **Inferencing:** Given a user request such as X:Y, Odin infers a sequence of tool invocations that form a path in the database from object type X to the object type Y.

2. **Side-effect free:** In Odin, no tool invocation can have side-effects (i.e., it cannot read and then modify any object).

3. **Unique derivations:** Since Odin will find a unique derivation path between any two object types, the possible sequence of tool applications must be completely determined by the endpoints of the path.

4. **Retrieval only:** Odin acts as if all requested objects already exist and that it is therefore only retrieving that object for presentation to the user. Odin has a simple form of update operation, but the updating mechanism is disjoint from the main part of Odin (file retrieval).

[1]Actually Zeus is more than just the object manager, but it serves as a convenient name for the object manager.

5. **Simple data model and database:** The persistent store of Odin may be usefully viewed as a set of relations over objects. There is one relation for each tool. For example, given a tool R that takes in an object of type X and produces an object of type Y, there is effectively a relation R(x:X,y:Y) in this database. This set of relations naturally form a (hyper-)graph with a node for each object type, and an edge for each relation. It is assumed that there is a unique path in this hypergraph between any two nodes.

6. **Objects are passive:** In simple Odin, it is the case that the tool is given the object as input and the tool interprets the contents of that object. This contrasts with the usual Smalltalk view in which the object interprets any actions requested of it.

7. **Lazy:** Odin does not reconstruct any object unless some object from which it is derived has changed.

8. **Type constructors:** Odin has some simple type constructors corresponding to vectors, pointers, and records.

9. **Sentinels:** Odin has a simple form of constraint checker in the form of sentinels. Sentinels are objects that are retrieved periodically to see if they are still valid objects.

3. Design Issues

The design issues related to Zeus can be loosely divided into three categories. The management issues are concerned with what tools and objects can and should be under the control of Zeus. Flow of control issues determine when Zeus will activate tools. Modeling issues concern the extent to which Zeus keeps information about its constituent tools and objects. The design issues that are discussed are interrelated to such an extent that it is difficult to discuss any one in isolation.

3.1. Management Issues

Zeus, like Odin, is intended to be a general purpose object manager. Two issues stand out in the design of Zeus: (1) what tools should Zeus manage, and (2) what objects should Zeus manage? For tools and objects that are outside of Zeus management, users will need to deal with a non-Arcadian interface. In addition, Zeus will be compromised in its ability to record the history of objects that it does not manage, or that are modified by tools that it does not manage. Thus, there should be a bias in Zeus towards inclusion rather than exclusion of tools and objects.

3.1.1. Tool management

The question here is: what tools should be managed by Zeus? There seems no doubt that simple, side-effect free tools should be included. Odin demonstrates that such tools can be managed very effectively.

There are several classes of tools with side-effects that it would seem natural to include in Zeus in some fashion:

1. Some tools may directly modify objects as side-effects of their operation. For example, a unique name generator might have a file containing the current sequence number. Invocation of the name generator has the side effect of modifying the sequence counter file.

2. Some tools (termed circular) may indirectly have side effects by producing an output object that is to be deposited (immediately or after subsequent tool invocations) into one of its input objects. For example, a deletion tool may input an object, say a complete tree, delete a subtree, and produce a new tree that is to replace the old one.

   Circular tools also appear in the context of co-equal representations. A program, for example, can be represented as either a piece of text or as a flow graph. It is desirable for users to be able to manipulate either representation and have the other representation automatically change to maintain equivalence. This can be done if there are tools that can transform between the two representations. However such tools are effectively circular.

The above two types are not entirely disjoint because a side-effect producing tool may be viewed as a special case of the circular tool. Other examples of tools that have side effects are database systems and editors.

Odin treats tools with side-effects as a special class collectively called editing tools. Such tools are not included in the derivation database maintained for side-effect free tools. Rather, the user is responsible for explicitly invoking such tools when required.

There seem to be three possible choices in the way that Zeus can manage such tools:

1. Such tools can be kept separate. This would continue the current Odin practice. It would preclude control of side-effect producing tools.

2. Such tools can be integrated into the current derivation structure by extending that structure to contain cycles representing the side-effects. This impacts both lazy evaluation and the unique derivation property.

3. Such tools can be handled by functional programming techniques in which the "state" (i.e. the set of all objects modified by side-effect) is passed along as an extra parameter for all tools. This seems cumbersome, and is certainly an unnatural approach for programmers used to conventional programming languages.

3.1.2. Object management

The issue of object management is the question of what objects are to be managed by Zeus. Odin, for comparison, manages two sets of objects:

1. Atomic objects (i.e., non-derived objects) that are explicitly mentioned to Odin. Such an object is usually mentioned as the initial object in a retrieval request.

2. Derived objects explicitly mentioned in the output interface specification of some tool.

Not all objects referenced by tools are included in their interface specifications, and so they are not under Odin's management. As one example, consider compiling a C program that implicitly references an include file. Odin will not know of this dependency unless it is explicitly told of it. As a second example, the unique name generator mentioned above will take the sequence number file as implicit input.

Odin also assumes that the objects output by tools are distinct from its input objects. Circular tools, such as editors, must be specially handled by Odin.

Zeus may manage the same set of objects that Odin manages now. Alternatively, Zeus may expand one or more of the classes of objects to be controlled. This would mean extending one of the following classes:

1. The class of controlled atomic objects may be expanded. This would mean that all atomic objects would be made known to Zeus upon creation.

2. The class of implicit input objects may be expanded. This would mean that the input specifications for tools would need to be expanded to specify all

objects, including those modified by side-effect.

3. The class of overlapping input and output objects may be expanded. This would mean that the class of editing tools would be integrated into Zeus.

3.2. Flow of Control Issues

3.2.1. Lazy evaluation

Lazy evaluation, as represented in Odin, is a very desirable feature. It avoids the need for immediately rebuilding every object derived from some modified object. Instead, objects are recreated only when necessary. Depending upon other design decisions, the lazy evaluation policy may require some modification. For instance, if constraints are added (see below), then certain computations cannot be deferred. Lazy evaluation has worked well for Odin, and so it seems that it should be a Zeus policy to maintain lazy evaluation to whatever degree possible. It is not of overriding importance, however. Thus, if the addition of some feature requires a compromise with lazy evaluation, the compromise will generally be resolved against lazy evaluation.

3.2.2. Inferencing

The use of automatic inferencing facilities in Zeus is an important issue. Given this facility, users would not be required to explicitly state all operations involved in producing some object. Zeus would be able to automatically infer intermediate tool invocations from imprecise user requests. Odin already does this, and it works there because of the lack of side-effects, the relatively few object types, and the 1-1 correspondence between object types and tools. Some (possibly all) of these assumptions might not be true for Zeus, and so the opportunities for inferencing would be restricted.

The alternative to automatic inferencing is algorithmic tool invocation. In this case, someone (not necessarily the user) must specifically state which tools are to be invoked and in what order. This in turn implies that Zeus would need some form of procedural language (a programmable command language, in effect) so that users could build procedures for invoking many tools in the correct order. Such a model is certainly closer to the facilities provided by existing operating systems (e.g., Unix, VMS). It does mean, however, that incorporating new tools is more difficult; existing command procedures would need to be modified to make use of the new tools.

3.2.3. Passive versus active objects

Objects in Zeus can be expected to adhere to one of two paradigms: active or passive. Smalltalk is a good representative of the active object paradigm. Active objects accept requests (often termed messages) and interpret the request according to some internal procedure. Thus, two separate objects might accept the same message but interpret it quite differently. There is a cost for having objects be responsible for interpreting their actions; it is difficult to construct a global inferencing system that can determine what sequence of tool invocations need to be performed in response to an imprecise user request.

The alternative, passive objects, assumes that a tool is given an object as input and the tool interprets the contents of that object. This means that any tool invocation can have only one meaning: that specified by the corresponding tool. This is the Odin paradigm. It allows for global inferencing, but it does not allow for multiple interpretations of tool requests.

3.2.4. Constraints

It may be desirable to allow Zeus users to define powerful constraints on the legal states for a given collection of objects. Some constraints can be handled by the use of derivation dependencies, but it may be more natural to separate derivations from constraints. For example, one might want to enforce a constraint that no procedure can be integrated into a larger module unless it has successfully been tested. This could be handled by requiring the module to depend on both the procedure and the test output. This seems unnatural, however, because the module does not really use the test output, only its success or failure. It would seem more appropriate to allow the user to establish a constraint between successful test output and the module. Generalizing, such constraints might be used to enforce specific object flow within a project structure.

3.2.5. Daemons

In order to enforce constraint checking and to implement Critics, Zeus may need an active component, which we can generically term daemons. By a daemon we mean an activity that Zeus can perform on its own initiative. One of the common problems with daemons is that the initiation of activities may come at inopportune times. For example, multiple daemon activations may so overload a workstation that they can prevent a user from performing any useful work. As another example, a daemon may be initiated at the wrong time and complain about a transient inconsistency.

3.3. Modeling Issues

3.3.1. Typing

The current expectation is that all objects in Zeus will be strongly typed. The alternative is to have untyped (or mono-typed) objects as is found in Lisp.

An untyped object system has the advantage that it is often possible to build powerful tools that are useful in manipulating many kinds of objects. In a typed Zeus, by contrast, these powerful operators would have to be replicated for various object types. Of course, an Ada-like generic facility would simplify the replication process.

Strong typing presumably would be carried out in accordance with data abstraction principles. This means that each tool must specify the types of its inputs and outputs. Of course, this will not preclude type conversions where appropriate. It can be expected that a strongly typed Zeus would also have a general set of type constructors. An obvious set corresponds to those found in various programming languages: records, scalars, vectors, pointers, graphs, arrays, lists, functionals (for tool-generating tools), and user defined primitive types.

In the case that strong typing is chosen, another issue arises: should tools and object types be in one-to-one correspondence? Odin assumes this in order to facilitate automatic inferencing. The price is a proliferation of types and type conversions.

3.3.2. Meta-data

Zeus will maintain its own internal database. Depending upon other design decisions, it will record at least information concerning types, derivations, and histories. Collectively, this database may be said to contain meta-data (data about the structure of other data). Should this meta-data be accessible to users? If the answer is no, then it may be hard to ask questions about the state of Zeus: questions such as "what is the history of object X?" If the answer is yes, then it is not clear how to present this information to the user; should it be consistent with the normal user view of non-meta-data? Should the user be able to do things such as derive types from other types and keep histories of the changes to meta-data?

3.3.3. Extended history

Some history information is implicit in the derivation information kept by Odin. It may be desirable for Zeus to keep additional histories of modifications performed outside the derivation structure. This might correspond, loosely, to the information maintained by tools like SCCS or RCS. Given that extended histories are maintained, then Zeus should provide mechanisms for manipulating histories. This would, for example, provide undo-redo facilities.

3.3.4. Support for user-defined relationships

If Zeus is to be extensible, it seems that it must allow users to define their own relationships among objects. As a corollary, Zeus will also need a complete set of operations for managing such relationships. They might be built-in, or they might be provided by tools. In either case, they should allow users to ask questions such as: "where are all references to object X?", or "what other object has relationship R to object X?"

3.3.5. Granularity of objects

The Arcadia consortium has already debated the issue of object granularity. In the following discussion, the terms large and small do not refer to the physical size of objects, but rather to the number of relationships between objects. It might be more accurate to refer to coarse grain objects or fine grain objects. There are three obvious choices:

1. Zeus might restrict itself to only manage "large" objects. This is in line with Odin. The obvious consequence is that Zeus might not be able to represent certain kinds of tools and objects representing abstract data types (e.g., a collection of objects and tools representing a stack or queue). This is because the tools correspond to the operations of the abstract data type, and they typically have side-effects.

2. Zeus might allow only "small" objects. This would mean that large objects must be explicitly broken up into the smallest possible objects. As a result of maintaining small objects, there will be an increase in the number of explicit relationships. For example, when an IRIS tree is split into many small node objects, the connections (the edges) among the nodes become explicit relationships among the node objects. They are no longer hidden inside the single IRIS tree object.

3. Zeus might manage objects of varying granularity. This might be done in two ways. In the first way, objects could be of any size, but all would be disjoint. In the second way, objects might be dynamically decomposed into smaller objects, or dynamically grouped into larger objects. Thus objects might not be disjoint over time. For example, at some time, a whole IRIS tree might be treated as one object by Zeus. At another time, the tree might be treated as individual nodes and edges.