The Odin System: An Object Manager for Extensible Software Environments *

Geoffrey M. Clemm

CU-CS-314-86

University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

* This work was supported by grants from the U.S. Department of Energy numbered DE-FG02-84ER13283 and DE-AC02-80ER10718 and from the National Science Foundation numbered MCS-8000017 and DCR-8403341.

THE ODIN SYSTEM: AN OBJECT MANAGER FOR EXTENSIBLE SOFTWARE ENVIRONMENTS

by
Geoffrey M. Clemm

CU-CS-314-86 February 1986

The University of Colorado
Department of Computer Science
Boulder, CO 80309
(303) 492-7514

This work was supported by grants from the U.S. Department of Energy numbered DE-FG02-84ER13283 and DE-AC02-80ER10718 and from the National Science Foundation numbered MCS-8000017 and DCR-8403341

THE ODIN SYSTEM

AN OBJECT MANAGER FOR EXTENSIBLE SOFTWARE ENVIRONMENTS

by

Geoffrey M. Clemm

B.A., Harvard College, 1976

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

1986

Clemm, Geoffrey Michael (Ph.D., Computer Science)

The Odin System - An Object Manager for Extensible Software Environments

Thesis directed by Professor Leon J. Osterweil

The purpose of a software environment is to support the creation and maintenance of computer software. This support takes the form of a set of computer programs called "software tools", which are used by a programmer to generate and manipulate software and information about software.

The use of a software environment can be significantly simplified if the user's attention is focused on the information provided by the environment rather than the tools that create this information. The purpose of an "object manager" is to provide this focus by automating the process of tool invocation. An object manager will respond to a request for a piece of information, or "object", by invoking the minimal number of tools necessary to produce that object. If previously computed objects are automatically stored by the object manager for later re-use, significant improvements in response time can be achieved.

In an extensible environment, the kinds of information potentially provided by the environment are easily extended through the addition of new tools that generate and manipulate new kinds of information. The added complexity of an object manager designed to support this extensibility is significant, but the current rapid rate of technological change makes this flexibility essential.

In order to demonstrate the feasibility of efficient extensible object management in large software environments, an object manager called the Odin System was designed and implemented based on the results of this research.

ACKNOWLEDGMENTS

I gratefully acknowledge the support of my thesis advisor and friend, Lee Osterweil.

In addition, a special note of thanks to Stu Feldman, whose Make program provided the direct ancestor of the Odin System. Thanks also to Brent Welch and Mark Maybee who contributed to many stimulating discussions during the design of Odin.

And of course, Maribeth.

CONTENTS

CHAPTER

I. INTRODUCTION 1

1.1. Software Environments 1

1.2. Software Tools 2

1.3. Use of a Common Data Structure 4

1.3.1. Problems with a Common Data Structure 5

1.4. Use of an Object Manager 8

1.5. The Odin System 9

1.6. Thesis Organization 11

II. HISTORY AND BACKGROUND 13

2.1. Automated Object Managers 16

2.1.1. Large Software Systems 18

2.2. General Object Managers 19

2.3. Toolpack 22

III. SOFTWARE OBJECTS 24

3.1. The Store and the Cache 25

3.2. Basic Accessing Functions 26

3.2.1. Manipulate Accessing Function 27

3.2.2. Derive Accessing Function 28

3.2.3. Equal Accessing Function 29

3.2.4. ExtendCache Accessing Function 29

3.2.5. Move Accessing Function 29

3.3. Basic Accessing Functions Specification 29

3.4. Auxiliary Accessing Functions 33

IV. THE QUERY LANGUAGE 37

4.1. The Odin Query Language 39

4.2. Odin Objects 39

4.2.1. Atomic Odin Objects 39

4.2.2. Derived Odin Objects 40

4.2.3. Status Level of Odin Objects 44

4.2.4. Sentinels 45

V. THE SPECIFICATION LANGUAGE 48

5.1. Comments 49

5.2. Atomic Object Types 49

5.3. Derived Object Types 50

5.3.1. Derived Object Structure 51

5.3.2. Inputs 57

5.3.3. Tools 59

5.4. Linking Object Types 64

5.4.1. Equivalence Links 65

5.4.2. Cast Links 65

5.5. Pre-Defined Object Types 66

5.6. Compilation of the Specification Language 66

5.6.1. Disambiguation of Queries 67

5.6.2. Computation of Parameter Sets 67

5.7. Experience with the Specification Language 68

VI. THE ODIN COMMAND INTERPRETER 70

6.1. Display Command 70

6.2. Manipulate Command 71

6.3. Command Scripts 72

6.3.1. Guarded Copy 73

6.4. Utility Commands 74

6.4.1. History Substitution 74

6.4.2. Help 75

6.4.3. Odin Variables 81

VII. AN IMPLEMENTATION 88

7.1. An Implementation of the Odin Model 88

7.1.1. Manipulate and Derive 89

7.1.2. Equal 95

7.1.3. ExtendCache 95

7.1.4. Move 97

7.1.5. Delete 97

7.1.6. Size 97

7.1.7. Obj_Type 97

7.1.8. Obj_Key 98

7.2. The Physical Database Model 98

7.2.1. Example 98

7.2.2. Object Nodes 100

7.2.3. Source Graph 106

7.2.4. Element Graph 110

7.2.5. Parameter Lists 114

7.2.6. LRU List - Automatic Space Maintenance 114

7.2.7. Concurrent Access 115

VIII. WRITING AN ODIN SPECIFICATION 116

8.1. Phase 1 - Producing the Derivation Graph 117

8.1.1. Converting a Command Script 119

8.1.2. Algorithm for Conversion 120

8.1.3. The GAG Command Script 121

8.1.4. The GAG Derivation Graph 122

8.2. Phase 2 - Improving Efficiency 131

8.2.1. Methods of Optimization 131

8.2.2. Optimizing the GAG Specification 134

8.3. User Experience with GAG in Odin 137

IX. CONCLUSIONS 139

9.1. Contributions 139

9.1.1. Process View vs. Object View 139

9.1.2. Specification Language 140

9.1.3. Query Language 141

9.2. Future Research 141

9.2.1. Graphic Interface for Interactive Queries 141

9.2.2. Alternative and Constrained Derivation Paths 143

9.2.3. Object Granularity 144

9.2.4. Co-operating Object Managers 144

BIBLIOGRAPHY 146

APPENDIX

A. GAG Command Script 150

B. GAG Derivation Graph 157

C. GAG Optimized Derivation Graph 162

CHAPTER I

INTRODUCTION

1.1. Software Environments

Among the many problems software systems have addressed, one that has received increasing amounts of attention is the problem of producing and maintaining software itself. A software system which addresses this problem is commonly called a "software environment".

The first problem encountered in studying software environments is deciding what a software environment really is. The diversity of the perceived needs of people that produce and maintain software has generated a corresponding diversity of software systems intended to satisfy these needs.

In many ways, a software environment can be conveniently viewed as a structured repository of information about software. New software can be added to the repository, old software can be modified, and queries concerning the existing software can be made. A few examples of the kind of information that could be present in a software environment would be program source text, test data, modification histories, attributed syntax trees, compiled object code, data flow analysis, or results of symbolic execution.

It is important to note that a software environment is concerned with issues which are more difficult than those faced by most database applications.

Subsystems of the software environment concerned primarily with addition and modification of software information (such as structure editors) are often far more extensive than the update facilities provided by common databases. In addition, the kind of information requested by software environment queries often requires that the software environment have a deep understanding of the software being stored; the computation necessary to satisfy these queries is correspondingly complex and expensive. An example of this kind of query would be a request for the "results of symbolic execution of a program fragment X".

1.2. Software Tools

Historically, software environment research began with attempts to effectively assemble collections of discrete functional units that we now call software "tools". A software tool is a program used to manipulate or transform software or information about software. Initially, the major tools were language translators (assemblers and compilers). Later, editors were developed for creating and modifying software. This pair of tools, an editor and a translator, formed the earliest programming environment. The editor would be used create new software or modify old software, and the translator would be used to satisfy queries of the form "give me a (machine language) translation of this piece of software".

As software became more complex, simply looking at it in the form of source text or reading the output data produced by the executing programs was no longer sufficient. The plethora of detailed information in source text or runtime output needed to be tailored into a more comprehensible form. In response to this problem, additional tools that analyzed a program and produced a variety of different "views" of the software were designed and built. Some of these views were graphical descriptions of software, such as control flow charts [Nassi73] [Chapin74], data flow diagrams [Myers78] [Yourdon78], or decision tables [Montalbano74] [Metzner77]. Tools were also built for displaying and manipulating this new information. As software systems continued to grow in complexity, tools for managing and viewing multiple concurrent versions of the same software system were developed [Rochkind75]. As software systems were embedded in increasingly critical applications, tools were designed to provide extensive regression testing [Miller79], formal verification [Anderson79], and test coverage analysis [Ramamoorthy76].

This gives only a small sample of the number and variety of tools that were developed, but in practice only a few ever received extensive use. A critical problem with these collections of tools was the lack of uniformity among them. Each tool had its own idiosyncratic view of the software world, including how input should be specified and how output should be presented. Another related problem was overlapping and incomplete coverage. One tool would often produce the same information as another tool (usually in a different format and in response to different queries), while there was some related information that no tool would produce. Only a few tools that were understood and trusted would then be selected for use. Even today, the software environment for many programmers is limited to the editor and compiler that formed the earliest software environments.

1.3. Use of a Common Data Structure

An alternative to the view that software environments consist of a collection of independent tools arose from the Lisp programming perspective. The Lisp language specifies the data structure for a Lisp program (nested lists), and provides a simple Lisp function ("eval") for interpreting this data structure. The simplicity and convenience of this data structure encourages tool writers for Lisp to just extend the data structure to include whatever new information (if any) that is required for a new tool, rather than designing a new data structure for each application.

An advantage of this approach is the existence of a common data structure where all information is stored. Unlike a collection of individual tools, each with its own internal data structures inaccessible to other tools, any new kinds of information stored in a common data structure are available at runtime for use by other extensions. Another advantage is that the process of adding extensions encourages the re-use of existing input and output paradigms, rather than inventing a new paradigm in the context of each new tool. Since the same underlying data structure is being used in the extensions, the same input and output routines can often be re-used to display much of the extended data structure.

In several instances, systems organized around a central common data structure have provided software environments that in various ways far exceed the power and sophistication of most tool based software environments. A good example of a such an environment is the Interlisp system [Teitelman81]. Among the capabilities provided by Interlisp are a structure editor, an interactive cross reference analyzer called Masterscope, symbolic debugging with breakpoint capabilities, and extensive error recovery mechanisms through the DWIM (Do-What-I-Mean) facility. Integrating all of these capabilities through a common data structure allows each capability to make use of the functionality provided by the others. For example, while at a breakpoint in the debugger, the Masterscope capability can be invoked to examine system dependencies, and in turn, the DWIM capability can be invoked to correct errors encountered during this nested application of Masterscope.

1.3.1. Problems with a Common Data Structure

Although the use of a single common data structure can alleviate many of the problems encountered in environments formed from a collection of independent tools, it introduces a new set of equally difficult problems. The requirement that all data be stored in a single common data structure generates constraints that can significantly impact the design and functionality of the software environment.

1.3.1.1. Efficient Extensibility

There are a variety of tools that do not comfortably fit into environments organized around a single common data structure. The common characteristic of these tools is that they require large amounts of space and execution time ("large" is of course a relative term, but in any environment there will be tools whose space and execution time requirements are large with respect to that environment).

The problem with extending a single common data structure to support "expensive" tools is that even users of the environment that have no interest in that extension must pay the cost of both the increased size of the data structure and the increased computation time required to keep all the information in the extended data structure valid. The resulting decrease in response time and increase in storage usage of the extended environment can be high enough to discourage all use of the environment except for those applications that depend on the extension and therefore require the increased computation. Improved technology in the form of faster hardware and more sophisticated software gradually allows some of the expensive tools to become affordable, and thereby makes it feasible to add them to the integrated environment without disrupting the environment for other applications. Invariably though, some tools remain expensive and inevitably, even more expensive new tools are imagined and created.

Even when a tool potentially could be integrated efficiently into the environment, there frequently are pragmatic reasons for not doing so. For example, the necessary modifications to the existing data structure might require prohibitively expensive re-coding and re-design of the existing software environment. If the desired extension already exists or could easily be written in the form of a standalone tool, it might be desirable to test out the standalone tool in conjunction with the existing system before investing significant resources in modifying the existing system.

1.3.1.2. Evolvability

Evolvability is the ease with which the software environment can be modified in small increments. Especially when the software environment is being used to support its own evolution as a software system, it is very desirable for the current capabilities of the environment to still function while the environment is being modified or extended. In many cases, this requires that the system maintain two incompatible versions of the same data structure concurrently - a new version being developed and an old version that is used to support the development of the new version. The development of the software environment can be significantly hampered if the requirement for a single common data structure prevents the concurrent existence of both data structures.

1.3.1.3. Understandability

The mass of information provided by a powerful software environment can be overwhelming when dealt with as a whole. An approach to this problem is to partition the information stored in the environment into several objects, where each object contains a certain kind of information. Studying each kind of object in isolation can then provide a convenient mechanism for incrementally understanding the whole environment.

1.3.1.4. Robustness

When all information is gathered in a single central data structure, the results of system failure, either of software or hardware, can be disastrous. Damage to relatively small pieces of information can result in data inconsistencies that significantly corrupt larger bodies of information. If the central data structure is partitioned into multiple disjoint data structures, the damage can sometimes be quarantined to a single element of the partition.

1.3.1.5. Distribution

Networks of computers, especially networks of personal workstations, are becoming increasingly popular as the hardware support for software development. Distributing a common data structure across this network can be very difficult, unless the data structure is first partitioned into smaller objects. In particular, local control and modification of parts of the information can be hampered if a single data structure must be maintained across the entire network. Since one of the valued attributes of a distributed system is the ability to localize the effects of hardware failure, the robustness provided by partitioning the information is particularly significant for distributed systems.

1.4. Use of an Object Manager

Optimally, both a tool builder and a user of a software environment would probably like to be presented with a single data structure from which any information of interest can be retrieved. Unfortunately, as just seen, problems in areas such as extensibility, evolvability, and robustness often require that the information actually take the form of several disjoint data structures or "objects", where independent tools are responsible for maintaining each kind of object. The difficulty with this approach is that before a user or tool builder can access a piece of desired data, they must invoke the appropriate set of tools to generate the objects that contain that data.

Even when a given object is immediately available because it has been previously computed and stored, subsequent modifications to information from which this object has been computed can cause the information contained in the object to become out-of-date and therefore invalid. In order to deal with possibly out-of-date information, there must be some mechanism for producing an up-to-date version before a tool is invoked that requires it as input. In the worst case, the user must explicitly invoke each tool in the appropriate order.

Since obtaining up-to-date input information is a common problem faced by the user and each tool, a logical solution is to provide an automated system wide "object manager" that solves this problem in general. Each physically disjoint piece of information created by the user and produced by some tool would be considered an object, and providing the appropriate up-to-date object would be the job of the object manager. In order to provide the desired functionality, the object manager must have some global knowledge of the input and output behavior of all the tools, and must provide a mechanism whereby a user or tool can request the objects it needs. Given this information, the object manager can satisfy a request by retrieving the specified object (if it already exists and is up-to-date) or by generating the object through the invocation of the proper sequence of intermediate tools (if it can be produced from existing objects).

1.5. The Odin System

The issues involved in the design of an object manager is the subject of this dissertation. The main elements of this design are a specification language for describing the objects to be managed and the tools that produce them, a request language with which a user or a tool can name a desired object, and an interpreter that will accept a request for an object and produce that object. The specification language must be designed to allow the easy addition of new tools and new types of objects to the environment. The request language must be designed to allow convenient access to any object of interest. Finally, the interpreter must be designed to efficiently produce a requested object through use of existing objects and invocation of the various tools described in the specification language.

There are a large number of difficult problems in the design of a software environment that need not (and probably should not) be addressed in the design of an object manager. For example, the question of how to ensure that the set of tools provide a consistent interface to a user need not be a concern of the object manager. This separation of concerns allows experimentation with a variety of user interfaces to be performed concurrently under the control of a single object manager. Similarly, any other design decision that is embedded in the object manager, but is not essential to the function of object management, would limit the use of the object manager for testing out alternative designs in that area.

The central vehicle for this research has been a functional object manager called the Odin System. The Odin specification language is an extended production system, where each executable tool is described by a single production. The Odin query language is a functional language whose syntax most closely resembles that of message passing languages like SmallTalk. An Odin interpreter has been implemented, where the objects are host system files in a Unix operating system. The Odin System has been used as the object manager for a variety of software environments, including the Toolpack system for Fortran source code, a Unix tools system for C source code, and the GAG system for ordered attribute grammars.

1.6. Thesis Organization

The dissertation is divided into nine chapters. Following this introduction, chapter two provides a brief history of previous work in the area of automatic object management. The remaining chapters describe the issues that arise in the design and implementation of an actual object manager, the Odin System.

A specification of the kinds of objects that can be managed by the Odin System is presented in Chapter Three. This specification takes the form of a set of axioms that define a class of objects - any objects that satisfy these axioms can then be managed by the Odin System. Since an object manager is specifically designed to be able to manage objects produced by arbitrary tools, it can make few assumptions about the behavior of the tools or the structure of the objects produced by the tools. Of course, some minimal functional assumptions must be made. For example, the object manager must be able to invoke the tool and must be told the names of the generated objects.

Chapter Four describes the query language used to request an object from the Odin System, and Chapter Five describes the specification language in which the user specifies the tools and objects in the environment. Chapter Six describes a command interpreter that provides a user interface to the Odin System. Chapter Seven describes an implementation of the Odin system where the objects are host system files. Chapter Eight indicates how an Odin specification can be created by describing the transformation of a command script for a tool system into an Odin specification. Finally, Chapter Nine contains conclusions and suggestions for future research in this area.

CHAPTER II

HISTORY AND BACKGROUND

The first object managers for software environments were the programmers themselves. The major software objects were source code, compiled object code, and test data. The source code was stored in one box of cards and the compiled object code was stored in another box, preferably nearby. When the source code was modified, it was up to the programmer to produce a box of cards with the new object code and to throw out the box of cards with the old object code.

The development of reliable random access mass storage devices decreased the physical storage space associated with software objects. Rather than carrying decks of cards (or reels of tape) to the appropriate input device, the programmer could simply name the object desired, and a program called an "operating system" would retrieve the appropriate information from the disk file with the specified name. This made it possible for a single programmer to have access to hundreds or even thousands of software objects simultaneously.

One problem with the early software objects was that they did not capture the evolutionary character of a software object. Each software object, whether it was stored in a box of cards, a reel of tape, or a file on a disk, would only contain a view of the object from a single instant in its evolutionary history.

One of the early approaches to this problem appeared in Control Data Corporation's Update and Modify systems [CDC76]. In these systems, a modification to a software object would be specified as a set of additions and deletions. These modifications would be stored in the software object, rather than actually performed on the object, which allows for retrieval of an arbitrary version of that object.

An extension of this approach appears in SCCS (Source Code Control System) [Rochkind75] [Glasser78]. In SCCS, the user prepares a new version using a text editor, and then enters this new version through a "check-in" operation. SCCS automatically computes a minimal set of additions, deletions, and replacements that will convert the previous version into the new version.

The importance of this capability is illustrated by the continual appearance of new systems that provide extended or modified facilities for storing and accessing multiple versions in a single software object. Tichy's Revision Control System [Tichy82] for example, stores a complete copy of the most recent version rather the original version. "Reverse deltas" are then stored to allow retrieval of earlier versions. In addition, versions can be given a symbolic attribute such as "Stable" or "Experimental", and then requests such as "the most recent Stable version created by John Smith" can be used to retrieve a specific version. Further extensions to the features provided by SCCS and RCS are provided in Digital Corporation's Code Management System [DEC84] designed for use on their VAX line of minicomputers, and AT&T's Change Control System [Bazelmans85] which is a proprietary system used internally at Bell Laboratories.

Concurrent with development of methods for capturing the temporal development of an individual software object was the development of systems for capturing the relationships between these software objects. Initially, software objects that were stored on a tape or disk appeared simply as a sequence of files. An improvement over this representation was the development of hierarchical file systems, where sets of files were collected together into special files called "directories". Since one of the files in a directory in turn could be another directory, this allowed sets of files to be both grouped and nested. A popular example of such a hierarchical file system is found in the Unix operating system [Ritchie74].

One advantage of a hierarchical file system is that it is very straightforward to develop naming conventions for the software objects that allow a variety of operations to be performed on sets of files, where the sets of files for a given operation are determined by their grouping in the file system rather than explicit specification by the user. An example of this approach is found in [Cargill79], where compilation is performed by specifying a root directory from which a tool called the "Inclusion Builder" determines the appropriate source files to compile.

The limitations of a simple tree structure to store and display software objects encouraged the development of software object databases. An early example of this approach appears in White's PLISS system [White77]. When a module is added to this system, the list of all modules referenced by the module and the list of all modules which reference the module are automatically computed. Information about a module, including a graphical description of the reference lists, can then be obtained through the use of "Picture" and "Inquiry" requests. Later systems incorporated increasing levels of detail about the software objects, and dealt with increasingly finer grained objects. A recent example of such a system appears in Linton's work with relational databases [Linton84], where the software objects appear as tuples in relations.

2.1. Automated Object Managers

A result of the increasing complexity of software objects was that it was no longer feasible for an unaided programmer to fulfill the role of object manager. Instead of a few boxes of cards and the associated boxes of object code, the programmer was faced with software systems composed of complicated hierarchies and networks of objects, where each object in turn consisted of a complex set of named and numbered versions. The logical solution was to try to automate the process of object management.

Initially, automatic object management consisted of the use of command scripts. The sequence of commands necessary to build and manipulate the software objects would be stored in a command script, which would then be invoked by the programmer when necessary. The problem with this approach is that unless the software objects being manipulated are few and simple, command scripts are inflexible, non-descriptive, and inefficient.

Command scripts are inflexible because the language understood by the operating system is usually quite primitive. This results in the need to create variants of commonly used command files, in order to satisfy the needs of different users. For example, one user might want to use an optimizing compiler on a few critical segments of a software system, while using another compiler for the rest. Although some methods of parameterizing command scripts are usually available, there are inevitably variant actions that cannot be performed without modifying the command scripts themselves.

Command scripts are non-descriptive because they describe how to build something, not what that thing is. It is usually difficult (if not impossible) to analyze a command script to determine whether a system is "consistent" according to some criterion. This implies that another object containing a system description must be maintained. This requires that the programmer be familiar with two different languages (the command language and the system description language). In addition, the programmer must always ensure that a modification to the system description be reflected by the appropriate modification to the command scripts.

The most severe problem of command scripts though, is that they are inefficient. In practice, programmers are willing to maintain several sets of command scripts and separate system description files, but waiting for three hours for a system to be ready for testing after a single line of source code has been modified will be unacceptable. The inefficiency of command scripts stems from the difficulty of specifying re-use of information. A variety of intermediate objects, such as compiled object versions of source code, are usually created during the execution of command scripts, and many of these objects would still be valid for later invocations of these command scripts. The difficulty of detecting which objects are still valid causes most command scripts to be designed to use the safe approach of recomputing all intermediate objects. The high cost of unnecessarily recomputing intermediate objects often encourages the programmer to explicitly take back control of object management. Unfortunately, in complex systems it is very easy to introduce subtle errors in object management. This often leads to the approach in which the command script for building the system is invoked whenever a bug is found, just in case the bug has been caused by incorrect object management.

A significantly better approach to object management involves the use of a system explicitly designed to re-use exactly those intermediate objects that are still valid. Initially these systems were developed to handle a specific class of intermediate objects. For example, the System Building System [DeJong73] was designed to manage the object code produced from PLI source code. In response to a request to recompile a given software system, SBS would only recompile files that had been modified, and would re-use any object code that was still valid from previous computations. Later systems of this kind such as the Software Development Control System [Haberman79] were designed with explicit knowledge of version control, to allow efficient management of the objects computed from the various versions of the software objects.

2.1.1. Large Software Systems

The problem of efficient object management is especially severe for large software systems. Techniques that are successful for medium-size systems (10-50k lines of code) are often insufficient for large systems (1 million lines of code). In particular, more detailed analysis of which derived objects are still valid after a change to the system is often necessary, in order to minimize the recomputation following the change. The computation in this case has inevitably been compilation, therefore the objects being managed are compiled versions of source code. Among the systems specifically designed to cope with this problem were the Intermetrics Pascal system [Avakian82], the CHILL Compiling System [Rudmik82], and ADA Language System [Thall83]. In the Intermetrics system, the process of deciding which pieces of object code are valid after a source level modification is complicated by the lack of modular interface specifications in the Pascal language. This resulted in the presence of a system wide "compool" structure containing the definitions of all symbols that are referenced by modules other than the modules in which they are declared. The need to recompile this compool structure (90 megabytes for a 1 million line software system) after a change to any symbol was a serious impediment to effective use of the Intermetrics system.

2.2. General Object Managers

The problem with special purpose object managers is that they are not extensible. Only the objects for which the system was initially designed could be managed. This problem motivated the development of general purpose object managers that were intended to manage arbitrary objects produced by arbitrary tool fragments.

The first successful general purpose object manager appeared the Make system [Feldman79]. The objects in this system are host system files, and the rules specifying the relationships between objects are specified in a text file called a "Makefile". The importance of such a general purpose object manager is indicated by the continued widespread use of the original Make system, as well as by its large number of successors which either provide extensions to the basic Make system or are just re-implementations for different operating systems.

One common extension to Make was to integrate it with a version control system. An example of merging Make with the SCCS version control system appears in the Software Manufacturing Facility [Cristofor80]. The Build software construction tool [Erikson84] provides an alternative mechanism for manipulating several versions of software objects by allowing multiple default paths on which software objects can be placed. An example of a simple re-implementation of the Make system appears in Digital Corporations Module Management System [DEC84b]. A central characteristic of the Make system and its variants is the use of the host file system as the database of information about current software objects. This provides many of both the strengths and weaknesses of the system. The advantages of this approach is that the Make system is extremely compact and efficient - it can depend on the operating system to maintain the required database of information (i.e. the host file system). In addition, the user provided tools can simply retrieve and store their information in the host file system, allowing most standalone tools to be conveniently integrated into the Make system without modification. The drawback to this approach is that only the information provided by the operating system about the file system can be used to store and retrieve information about the software objects. If some capability depends on having more information about the software objects than is provided by the operating system, then this capability cannot be provided.

A description of how additional information could be used to support a general object manager appears in [Huff81]. A more comprehensive treatment of this subject is provided by Cooprider in his PhD thesis [Cooprider79]. Both of these treatments suffer from the absence of an actual implementation of the ideas presented. In some cases the ideas are too vague to be evaluated, and in others cases the feasibility of a successful implementation is doubtful. A more concrete approach to this problem is provided by the System Modeler [Schmidt82] [Lampson83a] [Lampson83b] developed at Xerox for the Cedar programming environment. This system provides basically the same object management features as Make, except that these features are specifically tailored for the Cedar programming environment. In particular, the Cedar editor and the compiler/linker for the Cedar language, Mesa, are explicitly supported. An important extension present in the System Modeler is that it supports object management in a distributed network of homogeneous computers. Another object management system that significantly extends the features provided by Make appears in Apollo's DSEE (Domain Software Engineering Environment) [Leblang84] [Leblang85a] [Leblang85b].

The main problem with all of the existing object managers is that they fail to successfully separate declarative information about the objects from algorithmic information about the tools that manipulate the objects. Instead, this information is combined into a single text object - a "Makefile" for the Make system, and a "System Model" for both the System Modeler and the DSEE. Both Make and DSEE contain mechanisms for providing "default" rules, but the semantics of these rules are too simple to allow for the specification of complex tools. This means that the use of a complex tool must be specified repeatedly in each Makefile or System Model. Unfortunately these are precisely the tools that the user would most prefer NOT to specify - both because of the needless complication to the object descriptions, as well as the expense involved in updating all of these specifications when the interface to such a tool is modified. Instead of allowing a single tool expert to precisely specify the interface to a given tool, each programmer that wishes to use the tool in his Makefile or System Model must be capable of providing that specification. In compilation environments, where most tools have the comparatively simple interface of a compiler or linker, this problem is a relatively minor one. In environments intended to support a complex and fluctuating set of tools (such as the Toolpack system described below), the problem becomes critical.

2.3. Toolpack

The motivation for the study and design of automated object management was provided by the Toolpack project [Osterweil82]. In this project, a group of universities and private corporations collaborated to develop a software environment consisting of a loosely coupled network of co-operating tools. One of the premises of this collaboration was that a functional environment should be produced, therefore efficiency was a major goal. A sophisticated object manager would provide the coupling between the various tools developed for the environment, as well as provide a simple interface through which the results of these tools would be made available to a user of the environment. Critical elements in the design of such an object manager would be the flexibility necessary to experiment with alternative arrangements of tools within the environment, and the efficiency necessary to ensure that the resulting environment was usable. The research in this dissertation is focused on satisfying these requirements.

CHAPTER III

SOFTWARE OBJECTS

The Odin system provides a framework within which user defined objects and tools can be effectively integrated. In order to maximize flexibility, the interaction between the the Odin system and the user-defined objects is limited to five basic accessing functions. These accessing functions consist of :

"Manipulate" - a procedure for running tools to view and modify objects

"Derive" - a function to derive new objects by running tools

"Equal" - a test for equality between two objects

"ExtendCache" - a procedure to allocate a new object for the Cache

"Copy" - a procedure for copying one object to another

Specifications of the accessing functions will be presented later in this chapter. Any objects for which accessing functions that satisfy these specifications can be implemented are suitable as objects in the Odin system. A few examples of possible kinds of objects would be files on a disk, data in primary memory, or entities in a relational database.

A way of understanding these accessing functions is to view the Odin system as manipulating an arbitrary set of user defined objects in a black box, where all manipulations take place through the predefined set of accessing functions. The algebraic specification for these accessing functions must be satisfied in order to guarantee that the manipulations performed by the Odin system are valid. If the user wishes to manipulate a new set of objects, he must implement the required set of accessing functions and must ensure that the accessing functions he has provided will satisfy the algebraic specification. For an example of a full implementation of the Odin Model where the objects are files in a Unix operating system, see Chapter Seven.

3.1. The Store and the Cache

All Odin objects are accessed through a "Store" which maps object names into objects. The objects in the Store are partitioned into two sets, called "atomic objects" and "derived objects". The set of names of all derived objects is called the "Cache" - any object not in the Cache is an atomic object.

The atomic objects are those that can be directly modified by the user, while derived objects are mechanically produced from atomic objects. Examples of atomic objects could be program source text, test data, or tuples in a relational database. Examples of derived objects could be compiled object code, output from test runs, or results of database operations such as projection and join.

In general, any objects that cannot be mechanically derived from other existing objects will be atomic objects - these objects could have been created by a text editor, or imported from some external source. Whether an object is atomic or derived is therefore not a characteristic of the kind of object, but rather is based on the tools being used to manipulate the object. For example, if the tools are a standard text editor and a tree building parser, then source text would be an atomic object and a syntax tree would be a derived object. On the other hand, if the tools are a structure editor that operates on a syntax tree and a pretty-printer that prints out a text version of a syntax tree, then the opposite would be true, namely, a syntax tree would be an atomic object and source text would be a derived object.

Operationally, the critical distinction between atomic and derived information is that only atomic information can be directly modified by a user. The derived information is by definition mechanically generated from atomic information, and therefore can only change in response to a change in the atomic information from which it is derived. In order to reflect this difference, there are two distinct accessing functions to invoke user defined tools - one for producing derived objects (the Derive accessing function) and one for manipulating atomic objects (the Manipulate accessing function).

3.2. Basic Accessing Functions

The five basic accessing functions are a function for invoking browsing and editing tools, a function to derive new objects by running tools, a function to test for equality between two objects, a function to add a new name to the Cache, and a function for moving one object to another. These accessing functions are respectively :

1. Manipulate : <ToolName, Argument, State> -> <Argument, State>
2. Derive : <ToolName, Argument, State> -> <Argument, State>
3. Equal : <Object, Object> -> Boolean
4. ExtendCache : <State> -> <ObjectName, State>
5. Move : <ObjectName, ObjectName, State> -> State

The types of entities that appear in the accessing functions are as follows :

Boolean : a truth value (True, False).
ToolName : a character string.
Object : a user defined entity, with a distinguished element "Err_Obj".
ObjectName : a character string.
Argument : an ordered multi-set of ObjectNames (a list of ObjectNames).
Cache : a set of ObjectNames.
Store : a functional mapping from ObjectNames to Objects.
State : a Cache and a Store.
Integer : an integer (0, 1, 2, ...).
TypeName : a character string.
KeyName : a character string.

The object "Err_Obj" is returned by the Store for all names that the user has not associated with an object. The types Integer, TypeName, and KeyName will appear in the auxiliary accessing functions that are described later in this chapter.

3.2.1. Manipulate Accessing Function

The Manipulate procedure is used to invoke user defined tools. Tools invoked through the Manipulate procedure may be used to browse through existing objects, create new objects, or modify existing objects. The output argument of the Manipulate procedure is the list of objects actually modified during that invocation. The only restriction placed on these tools is that they may not modify objects in the Cache.

3.2.2. Derive Accessing Function

The Derive function is used by the Odin system to create objects through the invocation of user defined tools. A tool invoked through the Derive function must be referentially transparent (no read access to its context) and side effect free (no write access to its context). This requirement ensures that the result of the tool is the same whenever it is applied to an "equivalent" sequence of arguments, where the equivalence of two arguments is specified by the Equal function described below. In particular, this means that interactive input cannot be used to guide the operation of a tool.

The Odin system takes advantage of the purely functional nature of Derive tools by saving the resulting objects to avoid later recomputation of the same Derive request. The objects resulting from a Derive invocation are then added to the Cache (with the Move accessing function). These new objects are listed in the output arguments of the Derive function.

Since the results of a Derive invocation can be cached for later re-use, it is preferable to invoke a tool through the Derive function whenever possible (i.e. whenever a tool is referentially transparent and side-effect free). This is not to indicate that the Derive function is in any way "superior" to the Manipulate procedure; rather, that they each have their appropriate applications. With an editor, it would be pointless to cache the results of the first edit session, and then each time the editor is invoked, simply return the results of this first session. The purpose of the editor invocation is to allow the user to perform non-deterministic modifications to the atomic objects. On the other hand, recomputing the output of a compiler each time it is invoked on a given object is pointless if each time it is guaranteed to produce the same result.

3.2.3. Equal Accessing Function

The Equal function is used by the Odin system to determine when two objects are functionally equivalent. The results of this function are used in conjunction with the Cache of previously computed information to avoid unnecessary recomputation.

3.2.4. ExtendCache Accessing Function

The ExtendCache function is used by the Odin system to allocate a name for a new object in the Cache.

3.2.5. Move Accessing Function

The Move function allows Odin to move objects produced by the Derive accessing function into the Cache. If it is the first time the object has been computed, a new Cache object name is allocated with the ExtendCache function.

3.3. Basic Accessing Functions Specification

The semantic specification of the accessing functions consists of a set of algebraic and logical axioms. This specification is complete in the sense that an implementation of the accessing functions is correct if and only if it satisfies these axioms. In these axioms, the symbol "=>" will be used for logical implication. For functions that return booleans, the operation "= True" will be omitted. For example, "Equal(A ,B)" will be used as a shorthand for "Equal(A, B) = True".

The first axioms state that Equal must be an equivalence relation.

```
Let ObjA, ObjB, and ObjC be any Object

  Equal(ObjA, ObjA)

  Equal(ObjA, ObjB)
    => Equal(ObjB, ObjA)

  Equal(ObjA, ObjB) & Equal(ObjB, ObjC)
    => Equal(ObjA, ObjC)
```

When these axioms are satisfied, a variety of intuitive properties of objects will hold and therefore need not be verified at runtime. For example, later axioms will state that certain properties of objects will hold if those two objects are equal. If "Equal(ObjA, ObjA)" were not necessarily true, then these properties could not be assumed for ObjA without first testing the result of "Equal(ObjA, ObjA)".

The next axioms state that the Manipulate function can only modify atomic objects (objects not in the Cache) and that these objects must be listed in the output Argument of the Manipulate function.

```
LET Tool be any ToolName
LET Arg be any Argument
LET InState be any State
LET Name be any ObjectName
LET OutArg = Manipulate(Tool, Arg, InState).Argument
LET OutState = Manipulate(Tool, Arg, InState).State

  InState.Cache = OutState.Cache

  OutArg INTERSECT InState.Cache = EMPTY

  Name NOT IN OutArg
    => Equal(InState.Store(Name), OutState.Store(Name))
```

These axioms guarantee that the Cache will not be corrupted by a tool invoked through the Manipulate function. In addition, they guarantee that the only derived objects that could become invalid as a result of the Manipulate call will be objects derived from the atomic objects listed in the output Argument. This allows the Odin System to efficiently maintain derived object validity information.

The next axioms state that the only result of the Derive function is the creation of the objects listed in the output argument.

```
LET Tool be any ToolName
LET Arg be any Argument
LET InState be any State
LET Name be any ObjectName
LET OutArg = Derive(Tool, Arg, InState).Argument
LET OutState = Derive(Tool, Arg, InState).State

  InState.Cache = OutState.Cache

  OutArg INTERSECT InState.Cache = EMPTY

  Name NOT IN OutArg
    => Equal(InState.Store(Name), OutState.Store(Name))

  Name IN OutArg
    => (InState.Store(Name) = Err_Obj)
```

These axioms guarantee that the Cache will not be corrupted by a tool invoked through the Derive function. In addition, they guarantee that since no atomic objects can be modified by a Derive call, no existing derived objects will become invalid as a result of a Derive call.

The next axiom states that the Derive function is referentially transparent, i.e. that applying the function to equal input objects will produce equal output objects.

```
LET Tool be any ToolName
LET ArgA, ArgB be any Argument
LET InState be any State
LET OutArgA = Derive(Tool, ArgA, InState).Argument
LET OutArgB = Derive(Tool, ArgB, InState).Argument

  Equal(InState.Store(ArgA[i]), InState.Store(ArgB[i]))
  {1 <= i <= length(ArgA)}
    => Equal(OutState.Store(OutArgA[J]), OutState.Store(OutArgB[J]))
      {1 <= j <= length(OutArgA)}
```

This axiom guarantees that using previously cached results of Derive requests will produce correct results. This axiom implies that editors and other tools that take interactive input cannot be invoked through the Derive function.

The next axioms state that the result of the ExtendCache accessing function is to add a new object name to the Cache. This new name and the modified State is returned as a result.

```
LET InState be any State
LET Name be any ObjectName
LET OutState = ExtendCache(InState).State
LET OutName = ExtendCache(InState).Name

  OutName INTERSECT InState.Cache = EMPTY

  OutState.Cache = InState.Cache U { OutName }

  Equal(InState.Store(Name), OutState.Store(Name))
```

This axiom (in conjunction with the axioms that require that the other accessing functions not modify the Cache) gives the implementor of the accessing functions explicit knowledge of what objects are in the Cache, namely, only objects with names resulting from an ExtendCache call. This knowledge is necessary for the implementor to guarantee that the accessing functions such as Manipulate and Derive do not modify objects in the Cache.

The last axioms state that the effect of the Move accessing function is to make the second object equal to the first object and to delete the first object.

```
LET InState be any State
LET Name1, Name2, OtherName be any ObjectName
LET OutState = Move(Name1, Name2, InState).OutState

  InState.Cache = OutState.Cache

  Equal(InState.Store(Name1), OutState.Store(Name2))

  OutState.Store(Name1) = Err_Obj

  OtherName != Name2
    => Equal(InState.Store(OtherName), OutState.Store(OtherName))
```

These axioms ensure that after an object resulting from the Derive function is moved into the Cache, it will be Equal to the object originally produced by the Derive function, and therefore can be re-used in place of that object.

3.4. Auxiliary Accessing Functions

In addition to the five basic accessing functions, there are four auxiliary accessing functions that optionally can be provided by the user to allow the Odin system to perform automatic storage management and automatic classification of atomic objects. These accessing functions consist of :

"Delete"- a procedure to delete an object

"Size" - a function that returns the size of an object

"Obj_Type" - a function that returns the "type" of an object

"Obj_Key" - a function that returns a "key" for an object

For certain classes of objects, it might be difficult or impossible to implement some or all of these optional accessing functions. In this case, the user would implement only the feasible subset and will only lose the functionality provided by the omitted accessing functions.

The first two auxiliary accessing functions allow the Odin system to automatically manage the storage of objects in the Cache. These accessing functions consist of a procedure to delete an object and a function that returns the size of an object :

6. Delete : <ObjectName, State> -> State

7. Size : <ObjectName, State> -> Integer

Usually a user will want to place size restrictions on the amount of space used by the objects in the Cache. If the Odin system is able to delete an object and find out the size of an object, it can delete objects from the Cache until the total space used by the Cache is less than the amount specified by the user. Currently, the deletion strategy is LRU (Least Recently Used), but alternative deletion strategies could easily be implemented.

The last two auxiliary accessing function allows the Odin system to automatically classify and name user defined objects. These accessing functions consist of a function that returns the type of an object and a function that returns a key for an object.

8. Obj_Type : <ObjectName, State> -> TypeName

9. Obj_Key : <ObjectName, State> -> KeyName

In some situations, atomic objects will be created without the mediation of the Odin system. In these cases, it is convenient to provide an accessing function that allows the Odin system to determine what kind of object was created so that the appropriate Derive and Manipulate tools can be applied to that object. The "type" returned by the Obj_Type function is a string that refers to one of the object types that the user declared in his Odin Specification.

A simple example of an Obj_Type function would be one that derives the type of an object from the name of the object. Following a convention common in many operating systems, the type of the object could be specified in the file name extension (the characters following the last period in the file name). The Obj_Type function would then just return a string corresponding to the extension of the file name.

The Obj_Key function returns a string that the Odin system will store as a "key" for specifying that object. This key can then be used to select a specific object from a set of objects.

3.4.1. Auxiliary Accessing Functions Specification

The first axioms state that the only effect of the Delete accessing function will be to remove the object being deleted from the State.

```
LET Name, OtherName be any ObjectName
LET InState be any State
LET OutState = Delete(Name, InState)

  InState.Cache = OutState.Cache

  OutState.Store(Name) = Err_Obj

  OtherName != Name
    => Equal(InState.Store(OtherName), OutState.Store(OtherName))
```

These axioms ensure that following a Delete call, the Odin System can re-use all objects in the Cache except for the object deleted.

The next axiom states that the size of an object that does not exist is zero.

```
LET InState be any State
LET Name be any ObjectName

  InState.Store(Name) = Err_Obj
    => (Size(Name, InState) = 0)
```

This axiom, in conjunction with the axioms specifying the behavior of the Delete function, ensures that the space used by the Cache as measured by the Size function can be made zero. A trivial implementation of the Size function would be a constant function that always returns zero. This is in fact the function that is used by default if an implementation omits the Size function.

The last axioms state that the Type and the Key of an object remains the same unless the object is modified with the Manipulate accessing function.

```
LET Tool be any ToolName
LET Arg be any Argument
LET InState be any State
LET Name be any ObjectName
LET OutState = Manipulate(Tool, Arg, InState).State

  Equal(InState.Store(Name), OutState.Store(Name))
    => (Obj_Type(Name, InState) = Obj_Type(Name, OutState))

  Equal(InState.Store(Name), OutState.Store(Name))
    => (Obj_Key(Name, InState) = Obj_Key(Name, OutState))
```

These axioms allow the Odin System to initially compute the type and key for an object, and then assume that these values are still valid until the object is modified by a Manipulate call.

CHAPTER IV

THE QUERY LANGUAGE

The behavior of the user of a software environment will be modeled as a sequence of requests, where each request is either a query or an update. A query is a read-only access to the information in the environment while an update is a modification to this information. One possible request sequence would be :

1(update):	create new program X
2(update):	create test data Y
3(query):	results of running X on Y
4(query):	data flow analysis of X
5(update):	modification to X
6(query):	symbolic execution of X
7(query):	results of running X on Y

In this chapter, a query language designed to provide complete access to all objects in a software environment is described. Commands for performing updates will be described in Chapter Six.

The complexity of software environment databases can generate some confusion concerning the normally straightforward classification of requests as queries or updates. One method of distinction takes the viewpoint of the physical database, and classifies an access to the database as an update or a query according to whether the access modifies the physical database. Another method of distinction takes the viewpoint of the user, and classifies an access to the database according to whether it modifies the information potentially returned by later accesses.

In a simple database that either stores or retrieves information, these two classification methods are identical. For more complex databases, information is stored both explicitly in the physical database, as well as in the form of procedures for computing the desired information. In such a database, the user viewpoint classifies a larger class of requests as "queries", since a query can invoke procedures that modify the physical database without modifying the information potentially returned by later accesses.

For example, a query for the "results of running X on Y" might involve compiling, linking, and loading the program X, and then giving the resulting executable code to the host operation system to obtain the test run results. Although computing the response to this query might involve extensive internal modifications to the database (such as storing the computed object code and executable for later re-use), it is still a query rather than an update from the user's viewpoint because it does not affect the information produced by later queries.

The ability to store information in the form of procedures to compute that information is critical for software environments, and therefore this ambiguity arises. Since our focus is the user of the software environment rather than the physical database, the definition of a request from the user's viewpoint will be the one we will use.

4.1. The Odin Query Language

The Odin Query Language is an object-oriented query language, whose syntax most resembles that of message-passing languages like Smalltalk [Goldberg83a] [Goldberg83b]. Each query specifies a single (but possibly compound) Odin object, and tools are invoked only as needed to create the specified object. For example, if an executable object were requested, various compilers and loaders might be invoked. The tools "might be" invoked because the Odin System automatically saves the objects from previous requests, so that a given object might already exist and therefore be immediately available. The tools necessary to satisfy a query are invoked through the Derive accessing function (see Chapter Three) in order to ensure that the results of the query will be a read-only access to the information in the environment.

4.2. Odin Objects

Each Odin object is either a user defined object (simple object) or a set of objects (compound object). Examples of simple objects would be source code, executable binary, or output from a test run. Examples of compound objects would be the set of objects containing the source code of a single program, the set of objects containing different versions of the same source code, or an executable program with objects containing input data for the program.

4.2.1. Atomic Odin Objects

The atomic Odin objects are created either as a result of calls to the Manipulate accessing function (see Chapter Three) or in some fashion external to the Odin system. These objects are called "atomic" because they cannot be automatically recreated by the Odin system, and therefore they are the basic building blocks from which the Odin system creates all other objects.

Every atomic object is given a type by the Odin system based on the result of the Obj_Type accessing function applied to that object. The type of an atomic object determines which derived objects can be produced from that object. In case the type is not recognized by the Odin system, no derived objects can be produced from that object.

4.2.2. Derived Odin Objects

A derived Odin object is an object (or set of objects) that can be produced from an atomic object (or another derived object) through the invocation of one or more tools. Examples of objects that can be derived from source code would be cross reference listings, executable binary code, or a formatted version.

There are two basic Odin operations for specifying derived objects : "derivation" and "parameterization". Derivation is used to transform an object while parameterization is used to add information to an object. Together, derivation and parameterization are sufficient to specify any object in a software environment.

A common kind of derivation called "selection" is also provided as a primitive operation, where selection is used to obtain a piece of an object. In some ways, selection can be thought of as the "inverse" of the parameterization operator - selection takes away pieces of the object while parameterization adds on new pieces.

4.2.2.1. Derivation

A derivation is specified by appending to the name of an Odin object a colon (':') and the name of the desired derivation. For example,

```
test.c :fmt
```

would request a formatted version of test.c, and

```
test.c :fmt :run
```

would request the result of compiling and executing the formatted version of test.c.

4.2.2.2. Parameterization

It frequently occurs that there is a variety of additional information that can be associated with an object and that will affect the derivatives produced from that object. In the Odin System, this additional information is associated with an object as the "parameters" of that object. For example : a debug parameter could cause the compile derivative to contain run-time checks; a library parameter could cause the load derivative to have undefined externals satisfied from a non-default library; and a format parameter could cause all printable derivatives to be generated in line-printer format.

A parameterized object is specified by appending to the specification of an object a plus sign ('+') and a parameter. For example, a debug parameter can be added to the object "test.c :fmt" as follows :

```
test.c :fmt +debug
```

If this new object is then run, e.g.

```
test.c :fmt +debug :run
```

the "run" object produced would contain debugging information.

It is often the case that a value should be associated with a given parameter. Such a value can be specified by appending to the parameter an equal-sign ('=') and the value. For example, if array bound violations are to be checked or if dereferencing of nil pointers are to be checked for the object "test.c", then respectively

```
test.c +debug=arrays
```

or

```
test.c +debug=nilref
```

would be specified.

If the value associated with a parameter is contained in another Odin object, the value is specified as the Odin object surrounded by parentheses. For example, suppose that there is a derivation named "lib" that will produce a library from source code. Then the result of running "test.c" using the library produced from an object called "util.c" would be specified as :

```
test.c +lib=(util.c :lib) :run
```

4.2.2.3. Selection

As mentioned earlier, an Odin object can be either a simple object or a compound object (i.e. a set of objects). Frequently, it would be desirable to specify some subset of the objects in a compound object. To allow this, Odin associates a "key" with every Odin object.

An atomic object is given a key based on the result of the Obj_Key accessing function applied to that object. A derived object is given a default key equal to the key of the atomic object from which it was derived. For example, if the key of the atomic object "src/test.c" is "test", then the key of "src/test.c :run" would also be "test".

In case a derived object is a compound object, the key for each element of the compound object is generated by the tool that produces the compound object. For example, suppose "src/test.c :output" specifies the output objects generated when running "src/test.c". This derived object is a compound object because a program can generate more than one output object. Since the tool that executes a user's program is responsible for giving keys to the output objects, they could be arbitrarily given the keys "out1", "out2", etc. A more useful and more likely convention would be for the tool to use the object names given by the user's program to the output objects as the keys for the output objects.

The subset, from a given compound object, of objects with a certain key can be specified by appending to the name of the compound object an at-sign ('@') and a key. For example, suppose that running "src/test.c" produces three output objects named "DATA", "source.list", and "source.errors". These three objects could be specified as the three Odin objects,

```
src/test.c :output @DATA
src/test.c :output @source.list
src/test.c :output @source.errors
```

4.2.3. Status Level of Odin Objects

Associated with each Odin object is a status level, where a status level is one of OK, WARNING, ERROR, NOREAD, NOFILE, and ABORT. OK is considered the maximum status level and ABORT the minimum. The status of an atomic object is always OK. The status of a given derived object depends on the results of the tools needed to produce that object. If any tool generated warning messages, the status level of the given object is at most WARNING. If any tool generated error messages, the status level of the given object is at most ERROR. If any object that was needed to generate the given object was not readable, the status level of the given object is at most NOREAD. If any object that was needed to generate the given object did not exist, the status level of the given object is at most NOFILE. If any object that was needed to generate the given object had status level ERROR, then the status level of the given object is set to be ABORT.

If the status level of an object is less than OK, the status level is indicated whenever that object is requested. The actual warning or error messages that were produced can be displayed by requesting the results of running the internal WARNING tool or the internal ERROR tool (see Chapter Five). Assume that the ":warn" and ":err" derivations invoke the WARNING tool and the ERROR tool respectively. If the request for the object,

```
test.c :run
```

indicated that abort status was set for that object, the errors that caused the generation of the abort status would be listed in the object,

```
test.c :run :err
```

Error messages are included in the list of warning messages, so the list of errors is always a subset of the list of warnings. The difference between an error and a warning is that an error prevents the tool from generating its output, while a warning indicates that although output was generated, it might be faulty. An example of an error message from a loader would be

```
Unsatisfied external reference : "proc1".
```

An example of a warning message from a loader would be

```
Multiply defined external : "proc2", first copy loaded.
```

4.2.4. Sentinels

In any software system, it is very useful to be able to specify semantic constraints on the software objects in the system. For large systems, this functionality is often provided by a system manager. Any change to a part of the system is submitted to the system manager, who is then responsible for performing all the necessary regression testing and other analysis necessary to ensure that the modification is acceptable. If the modified system satisfies all the tests, the system manager would then install the modification.

In Odin, this capability is automated through the use of distinguished Odin objects called "sentinels". Examples of sentinels would be

```
thesis.txt :spell
prog.c +input=(thesis.txt) :run
```

If sentinels are activated, a modification to an atomic object is rejected if it would cause the status level of any sentinel to become ERROR or less. If a modification is rejected, Odin would generate an error message indicating which sentinels have been violated. The user desiring the modification would then have to either develop an alternative modification that does violate the sentinel, or would have to delete the sentinel that is being violated.

In the above list of two sentinels, assume that the ":spell" object receives ERROR status if any spelling errors are detected, and that the ":run" object receives ERROR status if any error messages are generated in the attempt to compile and run "prog.c" with input object "thesis.txt". Then if a modification to "thesis.txt" is attempted, Odin will check that the "thesis.txt" object is spelled correctly, and that the "prog.c" program with "thesis.txt" as input will run with no errors. If either of these checks fail, the modification to thesis.txt will be rejected. Alternatively, assuming "prog.c" uses the system library "/usr/lib/jobs", if a modification to "prog.c" or "/usr/lib/jobs" is attempted, Odin will again check that prog.c runs successfully with "thesis.txt" as input, before permitting the modification.

The list of sentinel objects is stored in the special Odin object specified as a vertical bar ("|"). Any user is permitted to add arbitrary Odin objects to this special object, and thereby specify arbitrary semantic constraints. For example, suppose that the derivation ":output" computed the output from running a program and the derivation ":diff" compared two objects. Then a regression test could be specified by adding the following sentinel to the sentinel list :

```
prog.c +input=(test.1) :output +compare=(test.1.out) :diff
```

This sentinel would be violated if any errors occur in compiling and running "prog.c" with "test.1" as input, or if the output from this run was not identical to the file "test.1.out".

The advantage of the sentinel approach over many other formal semantic constraint specification techniques is that the constraints can be specified in terms familiar to an ordinary programmer. Even a beginning programmer will be able to state that "my program must produce file X as output". As the sophistication of a programmer increases, more precise constraints will be specified, and assuming that a program to verify these constraints exists, these constraints would be entered into the system as new sentinels. The overhead of performing the computation necessary to detect sentinel violations is minimized through the use of the same mechanisms used by the Odin system to optimize the recomputation of derived objects.

CHAPTER V

THE SPECIFICATION LANGUAGE

The specification language is designed to allow the integration of any existing tool or set of tools into the Odin System, with no modification to the tools themselves. This is critical when a tool only exists in the form of executable binary, as is often the case for host system provided tools. The only tools provided by the Odin System itself are ones whose purpose is to support this task of integration.

For example, a compiler would be provided in an Odin environment by describing the host system compiler in the Odin specification language. On the other hand, Odin itself provides a tool that will interpret an object containing a list of object names as a "collection of objects", so that this collection of objects can be treated as a single object by a user of Odin. Odin would ensure that a request to run a tool on this collection would in fact invoke the tool on each of the elements in the collection.

The specification of each tool is entered into a text object called a "derivation graph". Basically, a specification consists of the name of the tool and a description of the input and output behavior of the tool.

For example, a simple formatter could be described as follows :

```
fmt "formatted version of C code" :
      USER pol_c.cmd
              : c
```

where fmt is the name of the results of applying a C code formatter, the string in quotes on the first line describes this object, the name following the keyword USER on the second line is the name of the formatting tool, and c names the kind of object which is suitable as input to the formatter.

In general, the i/o behavior of a tool can be far more complex than this simple example, but this basic model of the naming the output of a tool, naming the procedure that invokes the tool, and then describing the input to the tool, will always be followed.

5.1. Comments

Comments can be placed anywhere within the derivation graph. A comment is initiated with the sharp character ('#') and is terminated by the end-of-line character.

5.2. Atomic Object Types

Every type of object that is to be edited directly by the user is given a unique "atomic object type". Each atomic object type is declared in the derivation graph by specifying the name of the atomic object type followed by the keyword ATOMIC and a string that provides a short English description of that type of object. For example, atomic object types for C and Fortran source code could be declared as follows :

```
c ATOMIC "C source code"
f ATOMIC "Fortran77 source code"
```

The English description can be used in a help or menu system that informs the user about what atomic object types are currently known by the system.

5.3. Derived Object Types

Every type of object that is produced by some computer program or tool is given a unique "derived object type". Each derived object type must be described in the derivation graph. A description of a derived object type consists of a description of the structure of the derived object followed by a description of the tool that produces the derived object and a description of the inputs needed by the tool. For example, in the following rather complex derived object type description :

```
dbx <
  exe-dbx ^null "executables for a dbx run"*
  srcs-dbx (null) "sources for a dbx run"*
  keys-dbx (null) "names of source objects for a dbx run"*
  core-dbx "core dump for a dbx run"*
  > "Berkeley symbolic debugger run" :
       USER dbx.cmd
              : exe
              : (objsrcU)
              : (objkeyU)
              : PARAMETERS(id)
```

the description of the structure of the derived object is :

```
dbx <
   exe-dbx ^null "executable for a dbx run"*
   srcs-dbx (null) "sources for a dbx run"
   keys-dbx (null) "names of source objects for a dbx run"*
   core-dbx "core dump for a dbx run"*
   > "Berkeley symbolic debugger run" :
```

the description of the tool is :

```
USER dbx.cmd
```

and the description of the input is :

```
: exe
: (objsrcU)
: (objkeyU)
: PARAMETERS(id)
```

As with atomic object types, derived object types are associated with a string that provides a short English description of that type of object. This English description can be used to provide at runtime a menu listing what object types can be derived from a given object, based on the object type of that object.

This description can be marked with an asterisk indicating that it describes an "intermediate derived object type". An intermediate object type indicates an object that is not usually requested directly by a user, and therefore would usually not be displayed on a help menu. In the example above, exe-dbx, keys-dbx, and core-dbx were declared as intermediate derived object types.

5.3.1. Derived Object Structure

Due to the great variety in output behavior of tools, it is necessary to provide a flexible language for describing the various possible kinds of derived object types. Examples of different kinds of outputs that a tool might generate would be a single data object, a single object that refers to another object, a fixed number of different kinds of output objects, or an arbitrary number of similar output objects. The description of the structure of a derived object is always terminated by a colon.

5.3.1.1. Simple Derived Object

An object with a "simple" derived object type is just an ordinary text or data object. Some common simple object types would be assembler code generated from a higher level language, executable binary, cross reference listings, and error reports. A simple object type is analogous to a basic variable type in a programming language, such as boolean, character, or integer. Odin allows a user to introduce an arbitrary number of such basic types.

A simple derived object type specification consists of the name of the derived object type followed by a text string describing the type and a colon. For example, in :

```
exe "executable binary" :
```

"exe" is declared to be a simple derived object type.

5.3.1.2. Reference Derived Object

An object with a "reference" derived object type is an object that refers to another object. This is analogous to a pointer type in a programming language. Whenever such an object is used, such as when it is displayed or when it is given as input to a tool, it is automatically dereferenced by Odin so that what is displayed or received as input is actually the object referred to. An example of a tool that would produce a reference derived object type would be a tool that selects from a group of modules the module containing the definition of a specified procedure. If each module were stored in a single object, this tool would produce as output a reference to the object containing the appropriate procedure.

Any tool that produces a reference derived object could just as well have produced a simple derived object, by generating a copy of the appropriate object rather than generating a reference to it. The advantage of producing a reference rather than a new copy is that the Odin system can re-use any information derived from the original object when the corresponding information is requested from the reference object. For example, if

```
system.ref +proc=DoAll :select
```

generated a reference to the object "support.c" (i.e. the object support.c contains the definition of the procedure "DoAll"), then the Odin system would know that the object

```
system.ref +proc=DoAll :select :obj
```

is identical to the object

```
support.c :obj
```

If on the other hand, the tool "select" produced a simple object, the Odin system would have no way of knowing that the result of the ":select" was identical to "support.c".

There are two kinds of reference derived object types - pointer reference and name reference. Pointer reference objects contain the host system names of objects in the State (atomic objects) or in the Cache (derived objects). Name reference objects contain Odin queries that are translated by the Odin system into pointer references. For atomic objects, pointer references and name references will be identical, since in both cases atomic objects are referred to by their host system names. For derived objects, a pointer reference will be some string "/usr/odin/ODIN/FILES/c/157823" that refers to an object in the Cache, while a name reference will be an Odin query such as "test.c +lib=(/usr/lib/network.a) :run".

A pointer reference derived object type specification is like a simple derived object type specification except that immediately following the name of the object type is added a carat ('^') and the object type of the object being referred to. For example, in :

```
tgi_ptr ^ tgi "parser grammar" :
```

"tgi_ptr" is declared as being a pointer to an object of type "tgi".

A name reference derived object type specification is like a pointer reference derived object type specification except immediately following the name of the referred to object type is added an at-sign ('@'). For example, in :

```
f_main ^ fcast@ "scanner default main program" :
```

"f_main" is declared as containing the name of an object of type "fcast".

5.3.1.3. Compound Derived Object

An object with a given "compound" derived object type consists of a set of objects, each of which has the same object type called the "element object type" or is another compound derived object of the given type. A compound object that contains only objects of the element object type is called a "flat compound object" - one that also contains other compound objects is called a "nested compound object". A flat compound object is analogous to an array in a programming language - a nested compound object is analogous to a tree.

A tool should be specified as producing a compound object type when it produces an arbitrary number of files of the same type, or produces references to an arbitrary number of files. There are two kinds of compound derived object types - compound reference types and compound source types.

Compound Reference Derived Object

An object with a "compound reference" derived object type consists of a list of references to other objects. These references can be either by pointer or by name, as with reference derived object types.

A compound reference derived object type specification is like a simple derived object type specification except that immediately following the name of the object type is added the name of the element object type in parentheses. For example, in :

```
objC (obj) "list of object modules" :
```

"objC" is declared as containing pointers to elements of type "obj".

If the reference is by name, an at-sign ('@') is appended to the element object type name. For example, in :

```
so_ref (null@) "list of nroff included objects" :
```

"so_ref" is declared as containing the names of elements of type "null".

Compound Source Derived Object

An object with a "compound source" derived object type consists of a set of objects, all of which were generated by the tool. This is distinguished from compound reference objects where only references to existing objects are generated by the tool.

A compound source derived object type specification is like a compound reference derived object type specification except that square brackets ('[' ']') are used instead of parentheses. For example, in :

```
output [data] "output objects from a test run" :
```

"output" is declared as being a set of objects of type "data".

5.3.1.4. Composite Derived Object

An object with a "composite" derived object type consists of a set of a fixed number of objects, each of which has a specific, although possibly different, object type. This is analogous to a record or structure type in a programming language. In Odin, most tools that are normally considered to produce multiple outputs are instead considered to be tools that produce a single composite object as output. The members of a composite object type can be compound, reference, or simple object types.

A composite derived object type specification is like a simple derived object type specification except that immediately following the name of the object type is added a pair of angle brackets ('<' '>') containing a list of member object type specifications. Each member object type specification is either a compound, a reference, or a simple object type specification, except that the terminating colon is omitted. For example, in :

```
fscan <
  fst "scanner tables"*
  fst_lst "fscan compiler listing"*
  f_drive ^fcast@ "scanner driver routines"*
  f_main ^fcast@ "scanner default main program"*
  > "scanner tables"* :
```

"fscan" is declared as being a structure containing four elements - a simple type "fst", a simple type "fst_lst", a name reference type "f_drive", and a name reference type "f_main". The tool that produces "fscan" would be responsible for generating an "fst", an "fst_lst", an "f_drive", and an "f_main" output object - the Odin system would then be responsible for producing the fscan composite object from these four members.

5.3.2. Inputs

In order to produce an object of a given type, one or more input objects are needed by the tool that creates this object. These input objects are specified as a list of object types, each preceded by a colon. These object types can be atomic object types, derived object types, or parameter object types. For example,

```
f-scan (f) "source objects for a scanner module"* :
       COLLECT
             : fst
             : f_drive
```

specifies that the object types "fst" and "f_drive" are needed as input.

In addition, it is sometimes convenient to have a constant object as an input object, where this constant object contains data needed by the tool. In this case the name of the constant object is placed in quotes, again preceded by a colon. In the above example, if "f_drive" is the same for all tool invocations, the specification could be modified to read :

```
f-scan (f) "source objects for a scanner module"* :
        COLLECT
               : fst
               : "/usr/lib/std.f_drive"
```

5.3.2.1. Parameter Types

Normally, when a derived object is being produced, the actual inputs to a tool are determined automatically by Odin based on the object from which the object is derived. It sometimes is the case that a user would like to pass additional information to certain of the tools. This can be done when a derived object is requested at run time by appending to the description of the object from which the object is derived, a list of parameters. A parameter consists of a parameter type followed by the information that is to be placed in the input object corresponding to that parameter object type. Normally a tool will allow a parameter to be omitted, in which case a default value will be assumed.

The parameter types used as input to the tool producing a given object type are described in the derivation graph by specifying the keyword PARAMETERS followed by a list of parameter names separated by commas. For example,

```
: PARAMETERS ( debug, lib )
```

would indicate that the "debug" and "lib" parameters will be used.

5.3.2.2. Transitive Needed Object Types

In case one of the needed object types is a compound object, the question arises whether just the list of names of elements of the compound object is needed, or whether the data in those objects is needed as well. The default is that only the list of names is needed. If the data in these objects is needed, this is specified by placing parentheses around the appropriate needed object type. For example,

```
: (cmpd)
```

would indicate that the elements of the "cmpd" input are required, while

```
: cmpd
```

would indicate that only the names of the elements of the "cmpd" input are required.

5.3.3. Tools

The tool specifies what process must be executed to produce the specified derived object from the specified inputs. There are two kinds of tools - "internal tools" that are provided by Odin and "external tools" that are provided by the user.

5.3.3.1. Internal Tools

An internal tool is selected in a derived object specification with the keyword for that internal tool. For example, in the specification

```
ckey "name of c object"* :
        KEY
                :c
```

the internal tool KEY is selected.

Currently there are fourteen internal tools :

STRUCT

The STRUCT internal tool produces a composite object from a text object containing a sequence of Odin object specifications, one per line. Each specified object in order is placed as the corresponding member of the composite object. If the number of lines in the text object is not equal to the number of members of the composite object, the STRUCT tool generates an error message.

COMPOUND

The COMPOUND internal tool produces a compound pointer reference object from a compound name reference object.

COLLECT

The COLLECT internal tool produces a single compound reference object from a set of input objects by constructing a compound reference object whose elements are the input objects.

FLATTEN

The FLATTEN internal tool produces a flat compound object from a nested compound object. This is done by performing a depth first search of the input compound object, and adding a reference to each simple object found, in the order in which it is visited, to the output object.

UNION

The UNION internal tool produces a flat compound object from a nested compound object. This is similar to the FLATTEN internal tool, except that only one copy of each element object is placed in the result - if an object has already been placed into the result object, any later occurrences of that object in the input compound object will be ignored.

HOMOMORPHISM derivation-spec

The HOMOMORPHISM internal tool produces a compound object from another compound object by applying the derivation following the HOMOMORPHISM keyword to each element of the input compound object. A derivation is specified for homomorphisms in the same way that a derived object is specified in the Odin query language, except that the keyword HOMOMORPHISM is treated as the atomic object, and vertical bars ('¦') are used in place of colons (':'). For example, if it is desired that the "obj_src" derivation be applied to each element of the input compound object, then the tool would be specified as

```
HOMOMORPHISM ¦ obj_src
```

P-HOMOMORPHISM derivation-spec

The P-HOMOMORPHISM (parameterized homomorphism) internal tool is identical to the HOMOMORPHISM internal tool, except that the parameters used to produce the input to the tool are added to the parameters specified for the P-HOMOMORPHISM tool. This is used primarily when a tool is to be applied recursively to its results, in which case it is desirable that the parameters be passed along to the recursive invocations.

APPLY

The APPLY internal tool is similar to the HOMOMORPHISM tool, except that the derivation to be performed is stored in an object rather than specified in the derivation graph. Unlike the HOMOMORPHISM tool which applies one derivation to each of the elements of its input object, the APPLY tool applies each of the derivations in its first input object to its second input object. The APPLY tool provides the ability to apply derivations that were determined at runtime.

KEY

The KEY internal tool generates an object containing the key of the input object. For atomic objects, the KEY internal tool would invoke the Obj_Key accessing function (see Chapter Three). This is the key that would be used by the Odin selection operator.

CAT

The CAT internal tool produces a simple object from a compound object by concatenating together the contents of all simple objects that are elements of the compound object. The order of concatenation is the same depth first order of the FLATTEN and UNION internal tools.

ERROR

The ERROR internal tool produces a simple object from an arbitrary input object. This simple object contains all error messages generated by any tool in the process of creating the input object.

WARNING

The WARNING internal tool produces a simple object from an arbitrary input object. This simple object contains all warning and error messages generated by any tool in the process of creating the input object.

SENTINEL

The SENTINEL internal tool produces a compound object from an arbitrary input object. This compound object will consist of all sentinels that depend on the input object.

NAME

The NAME internal tool produces a compound name reference object from a compound pointer reference or composite object. This is the inverse of the COMPOUND internal tool.

5.3.3.2. External Tools

An external tool is selected in a derived object specification with the keyword USER followed by the name of the external tool. For example, in the specification

```
o "object code" :
        USER cc
             : c
```

the external tool "cc" is selected.

5.4. Linking Object Types

A linking object type is declared in the derivation graph by specifying the name of the linking object type followed by the keyword DERIVED and a string that provides a short English description of that type of object. Linking object types are used to specify relationships between other object types in the derivation graph.

It frequently occurs that the input necessary to produce a given derived object type, TypeX, can be provided by two or more different object types, Src1 and Src2. Rather than specify two derived object types, TypeX1 and TypeX2, where TypeX1 can be derived from Src1 and TypeX2 can be derived from Src2, it is more convenient to link the two possible input object types to a new object type, SrcX, and specify that this new object type is the input object type to produce TypeX.

For example, suppose that input to produce an executable binary object type "exe" can be provided by both the object type "obj-c" produced by a C compiler and the object type "obj-f" produced by a Fortran compiler. Rather than specifying two different object types, e.g. "exe-c" and "exe-f", that produce executable binaries from "obj-c" and "obj-f" objects respectively, a linking object type "obj" can be specified :

```
obj DERIVED "relocatable binary"
```

This "obj" object type is then specified as the input to the tool that produces an "exe" object type. Equivalence links are then specified to indicate that either "obj-c" or "obj-f" can be used as an "obj" object type.

5.4.1. Equivalence Links

An equivalence link is created by specifying the "to" object type followed by an arrow ('<=') followed by the "from" object type. In the preceding example these links would be added to the derivation graph :

```
obj <= obj-c
obj <= obj-f
```

5.4.2. Cast Links

It sometimes occurs that an object type that is derived from a given object type can be used in the same way that the given object type could be used. The most common example of this would be a program formatter. The output from the formatter can be used in all the ways that the original object could be used - it can even be formatted again. This situation is indicated in the derivation graph by specifying a cast link from the derived object type to the given object type. A cast link is specified like an equivalence link except that the head of the arrow is a vertical bar ('|='). For example, to indicate that formatted c code can be used whenever c code can be used, the following would be specified :

```
c |= fmt-c
```

5.5. Pre-Defined Object Types

Four pre-defined object types are provided by the specification language to facilitate the construction of generic tools that accept virtually any text or data as input. An example of such a tool would be a "diff" tool that detects differences between two objects.

These pre-defined object types could be thought of as being specified in a standard derivation graph prelude of the form :

```
.composite    ATOMIC "Any Composite Object"
.compound     ATOMIC "Any Compound Object"
.derived      ATOMIC "Any Derived Object"
.simple ATOMIC "Any Atomic or Simple Derived Object"
```

The diff tool could then be specified as :

```
diff "list of differences between a set of objects" :
        USER diff.cmd
                : .compound
```

5.6. Compilation of the Specification Language

The purpose of a specification language compiler is to translate the user specification into a sequence of tables designed for efficient interpretation by the Odin System. In addition to the straightforward mapping from the symbolic specification to internal data structures, some preprocessing of the user specification can be performed.

5.6.1. Disambiguation of Queries

Unlike many rule-based systems, the Odin System is not intended to explore alternative legal rule sequences to satisfy a given request. Instead, a canonical legal sequence is determined for each possible request. This canonical sequence would then be encoded into the tables produced by the specification language compiler.

The motivation for this is that users making requests to the Odin system are not expected to understand which tool fragments should be invoked to satisfy their requests. Therefore, a user would not be expected to be able to choose between one legal tool invocation sequence and another.

5.6.2. Computation of Parameter Sets

Since a canonical legal sequence is selected for each kind of user request, it is also possible to precompile the list of parameter types that are significant for each intermediate product of a given request. This list significantly increases the potential for re-use of intermediate objects. An example of this would be in the two requests :

```
test.c +stdin=(data.3) :output
test.c +stdin=(data.5) :output
```

These two requests ask for the output from compiling, linking and then running the program "test.c" - in the first request, with input file "data.3", and in the second request, with the input file "data.5". The only tool fragment that is interested in the parameter of type "stdin" is the final fragment that gives the executable and an optional input file to the host operating system for execution.

Since the compiler and linking loader are not interested in or affected by the "stdin" parameter, the same executable object can be used to satisfy both of these requests. The Odin system can perform this kind of optimization by utilizing the parameter list information generated by the specification language compiler.

5.7. Experience with the Specification Language

The specification language has proven to be the heart of the object manager. A detailed example of the use of the specification language appears in Chapter Eight, but a couple of general observations are appropriate here.

First, the declarative nature of the specification language is critical. Usually a tool can be added or deleted without knowledge of any other tools in the system except for those tools that produce objects used by that tool or use objects produced by that tool. The existence of syntactic items such as "equivalence arcs" encourages this view by allowing the writer of a derivation graph to first specify a tool in isolation, and then to link the input and output object types of the tool to the appropriate existing object types.

Second, the detailed nature of the specification language is also critical. User queries can be significantly simpler when the object manager can take advantage of the information present in a comprehensive specification. Since the most frequent contact with an object manager is through user queries rather than modifications to the specification, this provides significant leverage to the user of the object manager.

In many ways, these two characteristics of the specification language provide the tension that drives the design of such a language. On one hand, simplicity is desired to allow the easy addition of new tools and modification of old tools. On the other hand, complexity is desired to maximize the amount of information in the specification that can be used to simplify the queries. The specification language presented in this chapter was designed to strike a balance between these two forces. Additional experimentation and usage is required, though, before it can be determined if the balance struck is the correct one.

CHAPTER VI

THE ODIN COMMAND INTERPRETER

In order to allow convenient access to the capabilities provided by the Odin System, a command interpreter was designed and implemented. Since much of the work normally requiring explicit commands from a user are performed automatically by the Odin System, only a very simple command interpreter is required.

There are three basic commands to the Odin interpreter : the display command which is used to view an object, the manipulate command which is used to create or modify an atomic object (through invocation of the Manipulate accessing function), and the command script which is used to invoke a sequence of predefined Odin commands. In addition to the basic commands, there are a set of utility commands that allow the user to re-invoke previous Odin commands and to modify various characteristics of the Odin system.

6.1. Display Command

The display command displays an Odin object on the current standard output device, normally a terminal screen. An Odin object is displayed by specifying its name. For example,

```
test.c
```

would display the file named "test.c" in the current directory.

```
test.c +lib=(/usr/lib/simple.a) :run
```

would display the objects resulting from running the file "test.c" when loaded with the library "/usr/lib/simple.a".

6.2. Manipulate Command

The basic form of the manipulate command copies the contents of one Odin object into another Odin object. The second object (the one being changed) must be an atomic object, since only atomic objects can be modified. An Odin object is copied by appending to the name of the first object a right-angle-bracket ('>') and the name of the second object. For example,

```
test.c > test2.c
```

would put a copy of the contents of "test.c" into "test2.c".

```
test.c :run :err > test.err
```

would put into "test.err" a copy of the list of errors generated in attempting to run "test.c".

The basic form of the manipulate command will only be performed if the status level of the first object is no worse than WARNING. This convention is used because in practice we have found that if the new object is erroneous, a user wishes to preserve the old object. In case it is desired that additional constraints be satisfied before the copy is effected, the "guarded copy" form of an Odin command script should be used (see "Command Scripts").

An extended form of the manipulate command allows the user to invoke the Manipulate accessing function with an arbitrary host system tool on a specified Odin object. This allows the use of host system "editors" or "viewers". In this form of the manipulate command, appended to the object is a right-angle-bracket ('>'), a colon (':'), and the name of the host system tool to be invoked. For example,

```
test.c > :vi
```

would invoke the host system editor "vi" on the file "test.c", while

```
test.c :run :err > :more
```

would display the list of errors by running the host system tool "more" with the list of errors as its input.

In case the colon and host system command name is omitted, a default host system tool is invoked. The name of the default host system tool is specified in the Odin "Editor" variable (see "Odin Variables"). For example, if the default host system tool is "vi", then the following two commands are equivalent :

```
prog.ref >
prog.ref > :vi
```

To allow the perusal of erroneous objects, the restriction that the first object have a status level no worse than WARNING is relaxed for extended manipulate commands.

6.3. Command Scripts

An Odin command script consists of an Odin object that contains a list of Odin commands. This command script can be invoked by specifying a left-angle-bracket ('<') and the name of the Odin object. For example, if "script.odin" contained a list of Odin commands,

```
< script.odin
```

would invoke all the commands in script.odin.

6.3.1. Guarded Copy

In case it is desired that a sequence of copy commands be performed only if no sentinels would be violated by the results of performing these commands, the "guarded copy" form of command script invocation should be used (see Chapter Four for a description of sentinels). In this form, the left-angle-bracket is immediately followed by a vertical bar ('|'). All commands in a script invoked from a "guarded copy" must be simple manipulate commands (see "Manipulate Commands"). For example, if the file "test.copy" contained the following text :

```
/usr/tmp/test.c > /sys/rn.c
/usr/tmp/fix_lib.a > /usr/lib/jobs.a
```

then the command :

```
<| test.copy
```

would cause Odin to check the effect of these two manipulate commands on all the sentinels affected by them, with the changes only being made if no sentinels are violated. If simply

```
< test.copy
```

were used, then the changes would be made whether or not any sentinels were violated in the process.

6.4. Utility Commands

The utility commands concern history substitution, help, and Odin variables. History substitution allows the re-invocation of previous Odin commands. The help commands provide some guidance to the user with respect to valid command syntax and semantics. The Odin variable commands allow the user to query and modify various characteristics of the Odin interpreter.

6.4.1. History Substitution

A list of all basic commands invoked during a given Odin session is maintained by the Odin system. This list can be displayed and modified, and commands from this list can be selected and modified for re-execution.

The exclamation point character ('!') is used to refer to the Odin object containing the history list. Since the history list can grow quite large, when it is displayed only a given number of the most recent commands are actually presented. The number of commands displayed is specified by the History variable (see "Odin Variables"). The history list can be modified by specifying the history list as the object in a manipulate command. For example,

```
! > :vi
```

would invoke the host system editor "vi" on the history list.

A given command from the history list can be selected for execution by following the exclamation point with an integer or a word. An integer, i, selects the i'th most recent command in the history list. For example,

```
!1
```

would select the last basic command for re-execution. A word selects the most recent command that contains that word, where a word is an alphanumeric string. For example,

```
!run
```

could be used to select the command

```
test.c :run :err
```

The selected command can be modified before being re-executed by specifying the selection in the form of a manipulate command. For example,

```
!run > :vi
```

specifies that the "run" command should be given to the host system editor, "vi", before being re-executed. In case no change has been made by the host system editor, the command will not be re-executed.

6.4.2. Help

In order to provide some guidance to a user of the Odin System, various forms of help messages are provided. Currently, the help system is intended to provide a "reminder" function for use by those already familiar with the Odin system. A help system with a greater "tutorial" flavor would be necessary for a novice user.

6.4.2.1. Syntax

A simple syntax help facility is provided to describe the syntax of Odin commands and Odin objects. A list of topics is generated in response to a single question-mark ('?'). Following is a complete list of the current topics, and the results of requesting each topic.

```
?
Topics :
   Syntax
   Help
   Quit
   Display
   Copy
   Edit
   Script
   Variables
   OdinFileName
   HostFileName
   Operation
   Parameter
   ParameterKey
   ParameterValue
   FileType
   BaseType

? syntax
   [a] is optionally a
   [a]... is zero or more a's

? help
   ?
   ? Topic
   OdinFileName : ?
   OdinFileName + ?
   OdinFileName + ? : FileType

? quit
   Control-D

? display
   OdinFileName

? copy
   OdinFileName > OdinFileName

? edit
   OdinFileName Editor

? editor
   >
   > : HOST_TOOL

? history
```

```
   !
   ! Editor
   ! HISTORY_ENTRY
   ! HISTORY_ENTRY Editor

? script
   < OdinFileName
   <| OdinFileName

? hostcommand
   % [NAME]...
   % 'STRING'

? variables
   ? =
   VARIABLE = ?
   VARIABLE =
   VARIABLE = VALUE

? odinfilename
   HostFileName [Operation]...
   | [Operation]...

? hostfilename
   FILENAME.BaseType

? operation
   + Parameter
   : FileType
   @ KEY

? parameter
   ParameterKey [= ParameterValue]

? parameterkey
   Use a help command of the form
     OdinFileName + ?
   or
     OdinFileName + ? : FileType
   to determine appropriate ParameterKeys

? parametervalue
   NAME
   'STRING'
   ( OdinFileName )

? filetype
```

```
      Use a help command of the form
        OdinFileName : ?
      to determine appropriate FileTypes

   ? basetype
      Use a help command of the form
        ? :
      to determine appropriate BaseTypes
```

6.4.2.2. Atomic Type Help

If a list of the possible types for atomic objects is desired, typing a question mark followed by a colon ("? :") to the Odin System would generate a message of the form :

```
    Possible Base Types :
   c .............    C source code
   vc ............    C code stored in rcs format
   f .............    Fortran77 source code
   vf ............    Fortran77 code stored in rcs format
   int ...........    an integer
   tgi ...........    tree-building parser grammar
   fsi ...........    scanner grammar
   tgiref ........    tree-building parser/scanner grammars
   mf ............    Fortran77 source code with m4 constructs
   h .............    include data
   i .............    m4 Include Data
   ref ...........    reference file containing a list of file names
```

where the contents of this message is derived from the user's Odin specification.

6.4.2.3. Derived Type Help

If the name of the desired derivation has been forgotten or a list of the possible derivations is desired, a question mark ('?') can be put in place of the derivation name, and the Odin System will respond with a list of the possible derivation names that could appear at that position. For example,

```
test.c :fmt : ?
```

would generate the following message :

```
Possible Derivations from an Object of Type "fmt" :

obj ........   object code from c compiler
fmt ........   formatted version
xref .......   cross reference listing
run ........   results of executing a c program
```

This states that all of the following would be legal objects :

```
test.c :fmt :obj
test.c :fmt :fmt
test.c :fmt :xref
test.c :fmt :run
```

6.4.2.4. Parameter Type Help

If the name of the desired parameter has been forgotten or a list of the possible parameters is desired, a question mark ('?') can be put in place of the parameter, and the Odin System will respond with a list of the possible parameters that could appear at that position. For example,

```
test.c :fmt + ?
```

would generate the following message :

```
Possible Parameters :      id lib debug
```

This states that all of the following would be legal objects :

```
test.c :fmt +id
test.c :fmt +lib
test.c :fmt +debug
```

In fact, both id and lib should be associated with parameter values, such as :

```
test.c :fmt +id=run5
test.c :fmt +lib=(/usr/lib/network.a)
```

but since this required value information is not stored in the derivation graph, an unexpected parameter value (or lack of a value) will only be detected by the appropriate tool after the erroneous object has been requested.

A more exact form of parameter help can be requested by specifying which derivation you are intending to apply to the parameterized object. For example,

```
test.c :fmt + ? :obj
```

would generate the following message :

```
Possible Parameters :      debug
```

This states that the following would be a legal object :

```
test.c :fmt +debug :obj
```

Since the id and lib parameters are not relevant to the derivation from fmt to obj, these are not listed.

6.4.3. Odin Variables

The Odin interpreter provides to the user a set of variables. These consist of read-only variables and user-modifiable variables. A read-only variable provides to the user status information about the Odin system; a user modifiable variable allows the user to affect the operation of the Odin interpreter in various ways. Currently the functions affected by changing the values of user modifiable variables are the working directory, the default editor, the help facility, the history facility, the log facility, and the maximum total file space used by derived objects.

Dir

In commands, file names that do not begin with a slash ('/') refer to files with respect to the current working directory. Initially this directory is the one from which Odin was invoked by the user. This directory can be changed by modifying the value of the Dir variable.

ErrFile

All error messages are sent to a file which is initially set to be the standard output device. These messages can be redirected by modifying the ErrFile variable.

Editor

The Editor variable specifies the name of the default host system tool to be used for the abbreviated form of the manipulate command. An alternative default editor can be chosen by modifying the Editor variable.

HelpLevel

The HelpLevel variable specifies what degree of detail should be provided when the user asks for a list of possible derived file types (with the "file :?" help command). Normally, only commonly used file types are described, but the HelpLevel can specify that all possible file types should be described.

History

The History variable specifies how many of the recent commands should be listed when the history list is displayed (see "History Substitution").

LogFile, LogLevel

The "log" contains a brief description of each of the tools that were invoked to satisfy the request for an object. In addition, whenever a derived file is deleted by Odin to conserve disk space, a message describing the file deleted is sent to the log file. Since objects are saved by the Odin System between requests, the tool executions needed to satisfy a given request will vary. In particular, if an object is requested immediately after a request for that same object, no tools will be invoked. The LogFile variable specifies where the log information should be placed and the LogLevel variable specifies how detailed the generated log information should be.

MaxSize, MinSize, Size

Since the Odin system has the capability of deleting and recreating derived objects at will, parameters of interest are how much total space the derived files should be allowed to occupy and how much space is currently being occupied. After an Odin command is completed, derived objects will be deleted if necessary until Size is less than the MaxSize. The MinSize variable is provided to allow Odin scripts to specify that MaxSize should be at least a specified amount, without affecting MaxSize if it is already larger than that amount.

Sentinel

The Sentinel variable can be set by the user to have the value "on" or "off". Normally this variable has the value "on" and any modification to a file during an Odin session will cause a broadcast of the change to all affected derived files. Any objects specified as sentinels will automatically be updated to reflect the modification. If the Sentinel variable is given the value "off", no broadcasts will take place, and no sentinels will be updated until such an update is explicitly requested. The purpose of allowing the sentinels to be turned off is to allow a modification to a file to occur without requiring the computation of the possibly large number of sentinels affected by that modification. Turning off the sentinels is an inherently dangerous operation since it allows violations of the semantic constraints specified by the sentinels.

Verify

The Verify variable can be set by the user to have the value "on" or "off". Normally this variable has the value "off" and Odin assumes that any modification to a host system file will take place through an Odin manipulate command, and that the modification will be broadcast to all derived files that this would affect. In case host system files were modified other than through an Odin manipulate command, or were modified while the Sentinel variable was turned off, the user should set the value of the Verify variable to be "on". This indicates that actual host system date stamps should be inspected to determine if host system files have been modified.

6.4.3.1. Variable Manipulation Commands

Four commands are provided to manipulate these variables :

6.4.3.2. Show Variables

A list of the available variable names is generated in response to the command,

```
? =
```

Currently, this command would generate the list,

```
Dir Editor ErrFile HelpLevel History LogFile LogLevel
MaxSize MinSize Sentinel Size Verify
```

Of these variables, only Size is read-only.

6.4.3.3. Describe Variable

A description of the possible values that can be assigned to a given variable is generated in response to the command,

```
Variable = ?
```

The descriptions of the current set of variables are as follows :

```
Dir = ?
  The current working directory.

Editor = ?
  The default editor.

ErrFile = ?
  1 : Error information sent to standard output.
  2 : Error information sent to standard error.
  filename : Error information sent to file named "filename".
```

HelpLevel = ?
1 : Help returns information for common file types.
2 : Help returns information for all file types.

History = ?
The number of lines displayed from the history file.

LogFile = ?
1 : Log information sent to standard output.
2 : Log information sent to standard error.
filename : Log information sent to file named "filename".

LogLevel = ?
1 : No log information is generated.
2 : Insert commands executed by scripts into the log.
3 : And names of objects generated by external tools.
4 : And names of objects generated by internal tools.
5 : And names of objects deleted.
6 : And names of objects touched by broadcast.

MaxSize = ?
The maximum disk space (kilobytes) to be used by derived objects.

MinSize = ?
A minimum value for MaxSize (will not decrease MaxSize).

Size = ?
The current amount of disk space (kilobytes) used by derived objects.

Sentinel = ?
off : Sentinel validation off.
on : File modification validated by Sentinels.

Verify = ?
off : Assume all host system files modified through Odin.
on : Check all host system files for external modification.

6.4.3.4. Show Variable Value

The value of a variable is generated in response to the command,

Variable =

where "Variable" is a legal variable name. The default values of the current variables are :

Dir	the directory from which Odin was invoked
Editor	vi
ErrFile	1
HelpLevel ..	1
History	5
LogFile	1
LogLevel ...	3
MaxSize	5000
MinSize	5000
Size	0
Sentinel ...	on
Verify	off

6.4.3.5. Set Variable

A variable is given a new value with a command of the form,

```
Variable = Value
```

For example,

```
Dir = ../src
Editor = emacs
ErrFile = err.out
MinSize = 7000
Sentinel = off
Verify = on
```

An attempt to set the value of a read-only variable will generate an error message.

CHAPTER VII

AN IMPLEMENTATION

An implementation of the Odin system in the language C [Kernighan78] has been in use at the University of Colorado at Boulder since 1983. In this implementation, the objects manipulated by the Odin system are files in a Unix file system. This implementation includes the command interpreter described in Chapter Six. Concurrent multi-user access in either batch or interactive modes of operation is provided.

This chapter is divided into two sections which discuss aspects of the implementation. The first section describes the implementation of the Odin Model that was presented in Chapter Three. The second section describes the physical database model that is used to store the information needed by the Odin System to efficiently manage the software environment objects.

7.1. An Implementation of the Odin Model

All of the accessing functions (both basic and auxiliary) are provided in this implementation of the Odin System. The "State" is implemented by the Berkeley Unix 4.2 file system. The atomic objects are arbitrary files that were created either through the Manipulate accessing function or through some means external to the Odin System. A special directory called the FILES directory is specified by the user as the location for all derived objects. All names in the

Cache refer to files in this special directory. In order to ensure that user tools invoked through the Manipulate and Derive accessing functions do not modify objects in the Cache, only the Odin System has write access to this directory.

7.1.1. Manipulate and Derive

1. Manipulate : <ToolName, Argument, State> -> <Argument, State>

2. Derive : <ToolName, Argument, State> -> <Argument, State>

The Manipulate and Derive accessing functions are implemented through the creation of a command script that is given to the Unix operating system call, "system()", for execution. When a user adds a new tool to the Odin System (by specifying it in the Derivation Graph), he also creates a skeleton for the command script for that tool which is stored in a special directory called the CMD directory. A command script skeleton is identical to a host system command file except that macro names are specified in place of the input files it will use and output files it will produce. When it is necessary to generate a given derived file whose tool is an external tool, Odin creates a copy of the command script skeleton with macro names replaced with actual file names. The resulting command script is then given to the Unix system for execution.

For example, if the obj_f type was specified in the derivation graph as follows :

```
obj_f <
  obj-f "Fortran77 object module"*
  obj_key-f "Fortran77 source code file name"*
  obj_src-f ^null "Fortran77 source code"*
  > "Fortran77 object module information"* :
       USER obj_f.cmd
              : f
              : fkey
              : PARAMETERS(debug)
```

then the "obj_f.cmd" file for a Berkeley 4.2 Unix machine could be :

```
cd $(RUNDIR)

set source = `cat $(fkey)`.f
ln -s $(f) $source

set flags = ''
if (-e $(PRM)/debug) set flags = '-g'

(f77 $flags -c $source) >&! ERRORS

sed -n '/error/p' < ERRORS >! $(ERROR)
sed -n '/warning/p' < ERRORS >! $(WARNING)
if (-e $source:r.o) mv $source:r.o $(obj-f)
echo "$source" >! $(obj_key-f)
echo "$(f)" >! $(obj_src-f)

echo 0 >! $(OK)
```

In general, command script macros consist of a dollar sign ('$'), a left parenthesis ('('), a macro name, and a right parenthesis (')'), with no embedded spaces. In the preceding command script skeleton for "obj_f", the following macros appear :

```
$(RUNDIR)
$(fkey)
$(f)
$(PRM)
$(ERROR)
$(WARNING)
$(obj-f)
$(obj_key-f)
$(obj_src-f)
$(OK)
```

Since Odin interprets each occurrence of "$(" in a command script skeleton as the beginning of a macro, if the string "$(" is to appear in the command script after macro substitution, it must be escaped with an extra leading dollar sign. For example, if the line :

```
set special = '$(@#'
```

is to be a line in the macro expanded command script, it must be written as :

```
set special = '$$(@#'
```

7.1.1.1. Input File Name Macros

An input file is specified in a command script with a macro name that is the derived file type name for that input. For parameter inputs, there is a standard directory whose macro name is PRM into which all files for parameter inputs are placed by name. Therefore a parameter input is referenced by $(PRM)/parameter-name. For example, with the preceding specification for obj_f, the macro $(f) would stand for the input file of type f, the macro $(fkey) would stand for the input file of type fkey, and the macro $(PRM)/debug would stand for the input file associated with the debug parameter.

In case the parameter value is a compound file, the elements of the compound file will be linked into a directory that can be referred to as $(PRM.DIR)/parameter-name. The name of an element in this directory will be the same as its Odin key.

An alternative specification of an input file is with a macro name that consists of a left angle bracket ('<') followed by an integer. Assuming k is an integer, $(<k) would refer to the k'th input in the derived file type specification. In the example above, $(<1) would be equivalent to $(f), and $(<2) would be equivalent to $(fkey). The purpose of this alternate method is to allow one command script to be used for several different but related external tools even when they have different input file types. For example, in the following specifications :

```
inc_ref-i (null@) "list of m4-style included files"* :
        USER inc_m4.cmd
                : i

inc_ref-mf (null@) "list of m4-style included files"* :
        USER inc_m4.cmd
                : mf
```

the process that is run on a file of type i is the same as the one that is run on a file of type mf. In this case, $(<1) would have to be used in the inc_m4.cmd command script to refer to the input file. For example, on a Berkeley 4.2 Unix machine, the following inc_m4.cmd file could be used :

```
cd $(RUNDIR)
($(TOOL)/inc_m4.exe < $(<1) >! $(>1)) >&! $(ERROR)
echo 0 >! $(OK)
```

In addition to the macros for input files, there are three standard macro names, CURDIR, RUNDIR, and TOOL. $(CURDIR) stands for the directory containing the host system file from which the output file is derived. $(RUNDIR) stands for a temporary working directory in which the command script will be executed. $(TOOL) stands for the standard directory in which is placed the executables for external tools that are not provided by the host operating system.

7.1.1.2. Output File Name Macros

An output file is specified in a command script with a macro name that is the derived file type name for that output. For simple, reference, and compound reference derived file types, there would be just one output file. For compound source derived file types there would be one output directory in which each element of the compound source file will be created. For composite derived file types, there would be one output file for each member of the composite type. For example, with the specification :

```
run <
  stdout "standard output from a test run (when +out is set)"
  output [data] "output files from a test run"
  core-run "core dump of a test run"*
  > "test run" :
      USER run.cmd
            : exe
```

the output from the test run would be placed in the file $(stdout), the files generated by the test run will be placed in the directory $(output), and the core dump if any will be placed in the file $(core-run). An example of a run.cmd file for Berkeley 4.2 Unix would be :

```
cd $(output)

($(exe) >! $stdout) >&! $(WARNING)
if ($status != 0) echo run failed >>! $(ERROR)

if (-e core) mv core $(core-run)

echo 0 >! $(OK)
```

Analogously with input files, an output file can be specified with a macro name that consists of a right angle bracket ('>') followed by an integer. Assuming k is an integer, $(>k) would refer to the k'th output in the derived file type specification. In the example above, $(>1) would be equivalent to $(stdout), $(>2) would be equivalent to $(output), and $(>3) would be equivalent to $(core-run).

In addition to the macros for output files, there are three standard macro names for error reporting : ERROR, WARNING, and OK. If any fatal errors are encountered, these should be written to the file specified as $(ERROR). If any recoverable errors are encountered, these should be written to the file specified as $(WARNING). Finally, when the script terminates, a line consisting of the character '0' should be written to the file specified as $(OK). The file $(OK) will be used by Odin to determine if the script was able to terminate - if the script itself dies $(OK) will be left empty and Odin will assign abort status to the output of the tool. If the script did not abort, the files $(ERROR) and $(WARNING) will be used by Odin to determine if error or warning status should be set for the output of the tool.

7.1.2. Equal

3. Equal : <Object, Object> -> Boolean

The Equal accessing function is implemented as a sequence of two tests. First the size of each object is determined through a call to the host operating system. If the sizes differ, Equal will immediately return with the value False. If the sizes are identical, a byte by byte compare of the two input files is performed. Equal will then return True only if both files are equal at every byte position.

7.1.3. ExtendCache

4. ExtendCache : <State> -> <ObjectName, State>

In early implementations of the ExtendCache accessing function, an attempt was made to allocate names for derived objects corresponding to the atomic objects from which they were derived. For example, the derived object named

/usr/test.c :key

would be given the host name

/usr/test.c/key

Unfortunately, "/usr/test.c" is the name of an atomic object, and cannot also be used as a directory for the placement of derived objects. This name collision was initially resolved by modifying the names for derived objects. However obscure the modified name might be though, there always is the chance that this derived name will collide with the name chosen by a user for an atomic object. Another problem that arises in multi-programmer projects is that a user would frequently like to generate derived information from atomic objects of other users. If the derived objects were placed in the same directories as the atomic object from which they were derived, all users in a project would require write permission to the source directories of all other users in the project.

The solution to these problems used in the current implementation is for the user to specify a special directory in which all derived objects should be placed. This has the advantage that the user's source directories are no longer cluttered with the various derived objects. This is particularly important when the user is browsing through source files or archiving source files. Another advantage of placing all derived files in a special location is that it helps prevent the users from disrupting the contents of derived files. Since the purpose of derived files is to provide a cache of valid derived information, it is vital that this cache not be corrupted if its contents are to be re-used. Any derived file can of course be copied into a user directory and then modified, since the copy is then no longer a part of the cache. An initial version of this solution still derived the host name of an object from it's associated node name. For example, the object node named

/usr/test.c :key

would be given the host name

/user_specified_directory/usr/test.c/key

But this approach resulted in long skinny directory trees with unacceptable time and space costs associated with generating the large number of intermediate directories. The current implementation associates a unique "DataNumber" with each derived object, and this DataNumber is then used to locate the object in a short fat directory tree. This alternative method produced up to a 50% decrease in the storage space required for derived objects.

7.1.4. Move

5. Move : <ObjectName, ObjectName, State> -> State

The Move accessing function is implemented with the Unix system call "rename()".

7.1.5. Delete

6. Delete : <ObjectName, State> -> State

The Delete accessing function is implemented with the Unix system call "unlink()".

7.1.6. Size

7. Size : <ObjectName, State> -> Integer

The Size accessing function is implemented through access to the data returned by the Unix system call "fstat()".

7.1.7. Obj_Type

8. Obj_Type : <ObjectName, State> -> TypeName

The Type of an atomic object is based on the extension of the host system file name for that object. The extension of a file name is the string following the last period in the final segment of the file name, where segments are separated by a slashes. For example, the extension of each of the following file names is "c" :

```
/usr/geoff/src/test.c
src/test.c
test.c
test.1.c
```

7.1.8. Obj_Key

9. Obj_Key : <ObjectName, State> -> KeyName

The Key of an atomic object is based on the last segment of the host system file name for that object. It consists of the string preceding the extension, without the trailing period. For example, the key of the object with the host name "/usr/geoff/src/test.c" is "test".

7.2. The Physical Database Model

All information concerning the objects in the Odin system are stored in a database that is implemented as a single Unix file called the INFO file. This database is structured as a network with four distinct types of nodes. A given node is identified by its byte offset within the INFO file. The following sections provide a high level description of the structures and information stored in the INFO file. In order to illustrate these structures, the following example will be used.

7.2.1. Example

The INFO file in this example is the result from the request :

```
/usr/test.c :o
```

which is the object code resulting from compiling /usr/test.c.

For this example, a simple environment has been specified that has as atomic objects source code and include files in the C language. The purpose of the environment is to provide compiled object modules. Source code can contain statements of the form :

%include "filename.h"

which are interpreted by the compiler to mean that the text of "filename.h" should be inserted at this point. Included files can themselves contain include statements.

In this example, the user has created three files in the directory /usr : test.c, sys.h, and file.h. The first file, test.c, contains the source text for a program written in the language C. This file contains two include statements that refer to the files sys.h and file.h. The file sys.h also contains an include statement, which refers to the file file.h.

The environment has been specified so that a source code file is only recompiled if it changes or if one of the (possibly nested) included files has changed. In addition to the command script to invoke the host system compiler, the user has provided a command script for a tool fragment that scans a file for include statements and generates a list of the filenames specified in these statements. The full specification for this environment is as follows :

```
o "object module" :
        USER obj_c.cmd
                : (all_inc)
                : ckey
                : c

all_inc (h) "list of C-style transitively included files"* :
        COLLECT
                : trans_inc
                : inc

trans_inc (h) "list of C-style indirectly included files"* :
        HOMOMORPHISM |all_inc
                : inc

inc (h@) "list of C-style included files"* :
        USER inc_c.cmd
                : .simple

ckey "name of C file"* :
        KEY
                : c

h ATOMIC "C-style included file"

c ATOMIC "C source code"
```

7.2.2. Object Nodes

In the Odin database there is one "Object Node" for each object known by the Odin system. An object is either an atomic object or a derived object (see Chapter Four). A Unix directory is considered to be an atomic object, and therefore an Object Node corresponding to each known directory will be present in the database.

In the specified example, following the request "/usr/test.c :o" there will be fifteen object nodes in the database : one for the directory "/usr", three for the files "test.c", "sys.h", and "file.h", and eleven for derived objects (see Figure 7.1).

The names of these objects are :

```
/usr
test.c
sys.h
file.h
test.c :inc
test.c :trans_inc
test.c :all_inc
test.c :key
test.c :o
sys.h :inc
sys.h :trans_inc
sys.h :all_inc
file.h :inc
file.h :trans_inc
file.h :all_inc
```

The following sections describe the information stored in each of the object nodes.

7.2.2.1. Object Node Names and the Object Node Tree

Each object node is given a name, where a name is broken into a sequence of segments. The nodes are connected in the form of a tree, where the segments of the name (reading left to right) specify the path from the root of the tree to the object. Only the last segment of the name is stored in the object node, since the preceding segments are contained in the nodes found by walking up the tree to the root.

The name of an object node corresponding to an atomic file is the host system pathname for that file. Each directory name in the pathname of the file specifies a segment of the object node name for that file. For example, for the atomic object

```
/usr/test.c
```

the object node name would be

```
root-usr-test.c
```

A hyphen ('-') is used here to indicate the separation between two object name segments, and "root" is the name of the object node that corresponds to the root directory of the host file system.

The name of a derived file consists of the name of the host system file from which it was derived followed by a sequence of name segments corresponding to how the file was derived. For example, the object specified as

```
/usr/test.c :key
```

would be named

```
root-usr-test.c-key
```

7.2.2.2. Object Class and Type

The Odin system associates with each object a "class" and a "type". The class of an object specifies whether the object is atomic or derived. If an object is atomic, the type of the object is determined by its host system name. If the object is derived, the type of the object is determined from the kind of tool that produced the object.

In the example database, the type of the directory "/usr" is ".simple" (this is the default type for atomic objects with a missing or unrecognized file name extension). The type of "test.c" is "c" and the type of "sys.h" and "file.h" is "h". The type of the derived objects is specified by their final derivation (i.e. "inc", "trans_inc", "all_inc", "key", or "o").

7.2.2.3. Base Object

The node for a derived object contains a pointer to another object called the "base object". The base object is defined in terms of another set of objects called "source objects". A source object of a derived object is an object from which can be derived all sources needed to produce the given object (a source object can be one of these sources). The base object for a given derived object is then defined as the unique source object that can be derived from all other source objects of the given object.

In the example, "test.c" has no base object since it is an atomic object (see figure 7.2). The objects "test.c :inc", "test.c :key", and "test.c :o" have "test.c" as their base object, since each of them can be derived from "test.c", but cannot be derived from anything that is derived from "test.c". Both "test.c" and "test.c :inc" are source objects for "test.c :trans_inc" and "test.c :all_inc", but "test.c :inc" is their base object since it can be derived from "test.c".

7.2.2.4. Object Key

Every object has an associated key which is a character string. For atomic objects, the key consists of the last component of the host path name for the object with the file extension removed. For example, the key of the object "/usr/sys.h" would be "sys".

For compound source derived objects (see Chapter Five), the tool that produces the derived object will assign a distinct key to each element of the compound source object. For all other derived objects, the key is identical to that of the Base Object of the derived object. For example, the key of the object "test.c :all_inc" would be "test".

The principal purpose of a key is to allow the user to select a specific component from a compound source object. Assigning keys to all objects allows this kind of selection from compound reference objects as well. Unlike keys for elements of compound source objects, keys for elements of compound reference objects are not necessarily unique. Therefore, the result of selecting by key from a compound object is another compound object consisting of all those elements that have the appropriate key (see "Selection" in Chapter Four).

7.2.2.5. Object Status

The status of an object is stored in the object node associated with that object. The status value is either OK, WARNING, ERROR, CANNOT_READ, NO_FILE, or SYSTEM_ABORT. The interpretation of these status values is described in Chapter Four.

In addition to the status of the object itself, if the object is a compound object, the minimum status of all objects that are elements of the compound object is also stored. This element status is stored to avoid searching through the element graph each time this information is required.

Finally, a third status, the "non-abort status", of an object is also stored in the object node. Whenever the inputs necessary to create an object are erroneous, the Odin system will not attempt to create the object, but rather give SYSTEM_ABORT status to its object node. Instead of deleting the old value of the object, this old object is kept for potential future use. The old status of this object is then stored as the non-abort status. Under certain conditions, this object can be simply restored rather than recomputed, in which case its pre-abort status must be available so that it can also be restored.

7.2.2.6. Odin Clock

The Odin system keeps an internal clock which ticks every time some atomic object is modified. Associated with each object is a set of dates which are used to determine whether the object is valid (up-to-date).

7.2.2.6.1. Modification Dates

There are three "modification dates" associated with an object. The "primary modification date" indicates the last time the object was modified. The other two dates, the "dependency modification date" and the "element modification date" are computed from the primary modification dates of other objects in the system. The dependency modification date is the maximum primary modification date of all objects whose contents can affect the contents of the given object. The element modification date is only computed for compound objects, and it is the maximum primary modification date of all elements of the given object.

The dependency and element modification dates are computed and stored to improve the efficiency of determining whether a given object is valid. In particular, the validity of a derived object is determined by scanning the dates of all other objects upon which the derived object depends. If a dependency modification date has already been computed for an object, this scanning can be omitted.

7.2.2.6.2. Verification Dates

There are two "verification dates" associated with an object. The "primary verification date" contains the value of the system clock at the last time the system verified that the object was valid. The "element verification date" is only computed for compound objects, and it indicates the last time the system verified that all the elements of the object were valid.

Both verification dates are used to improve the efficiency of determining whether a given derived object is valid. If the verification date is equal to the current system date, then the object is guaranteed to be valid. Without a verification date, the system would have to check the modification date of every atomic object upon which the derived object depended to determine if the derived object were valid.

7.2.3. Source Graph

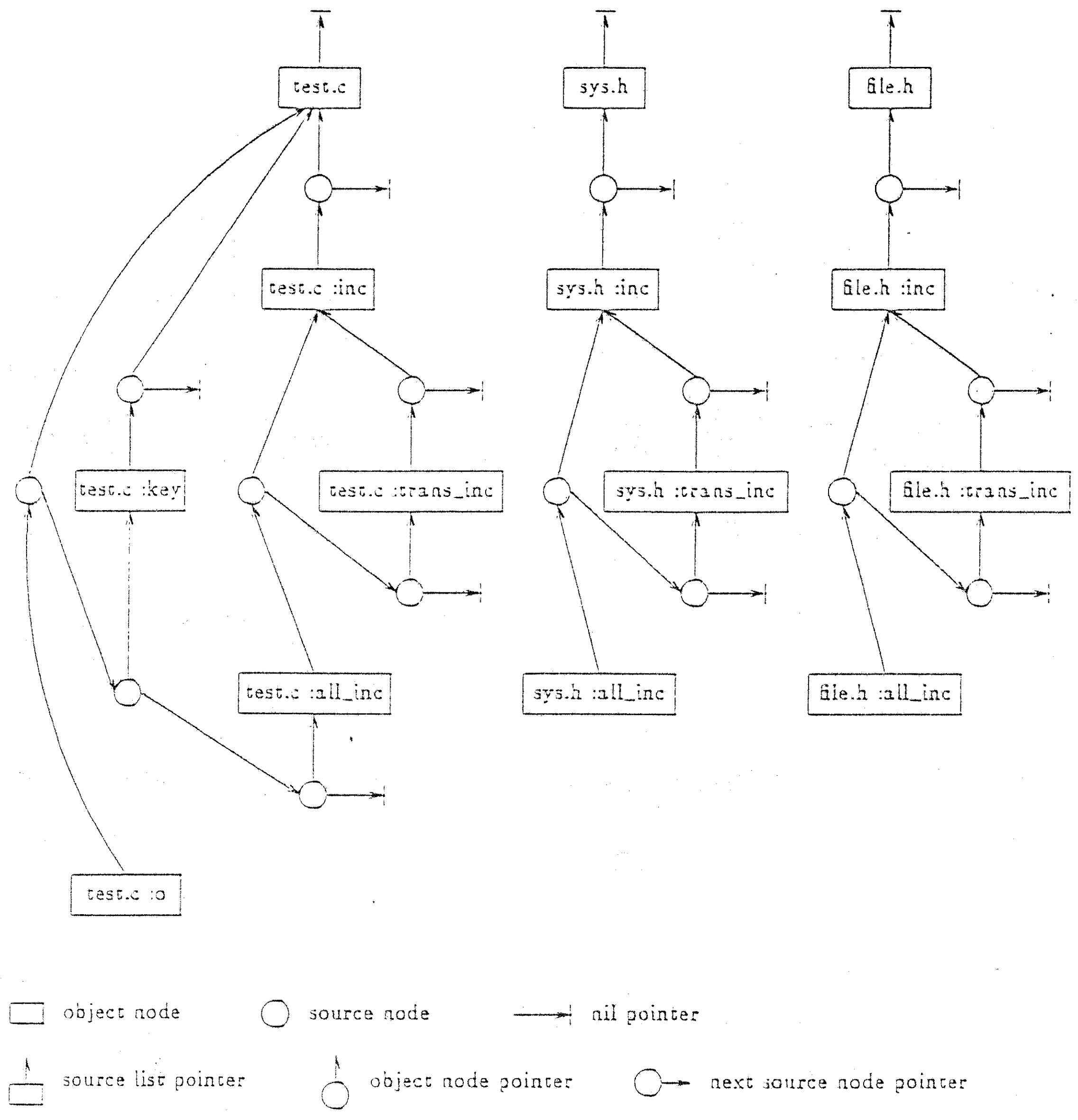
The object nodes are linked together via "source nodes" to form a directed acyclic graph called the Source Graph. Each source node specifies an edge in the Source Graph. An edge in the source graph from object node X to object node Y indicates that the object corresponding to X is produced by a tool that uses the object corresponding to Y as input.

7.2.3.1. Source List and Output List

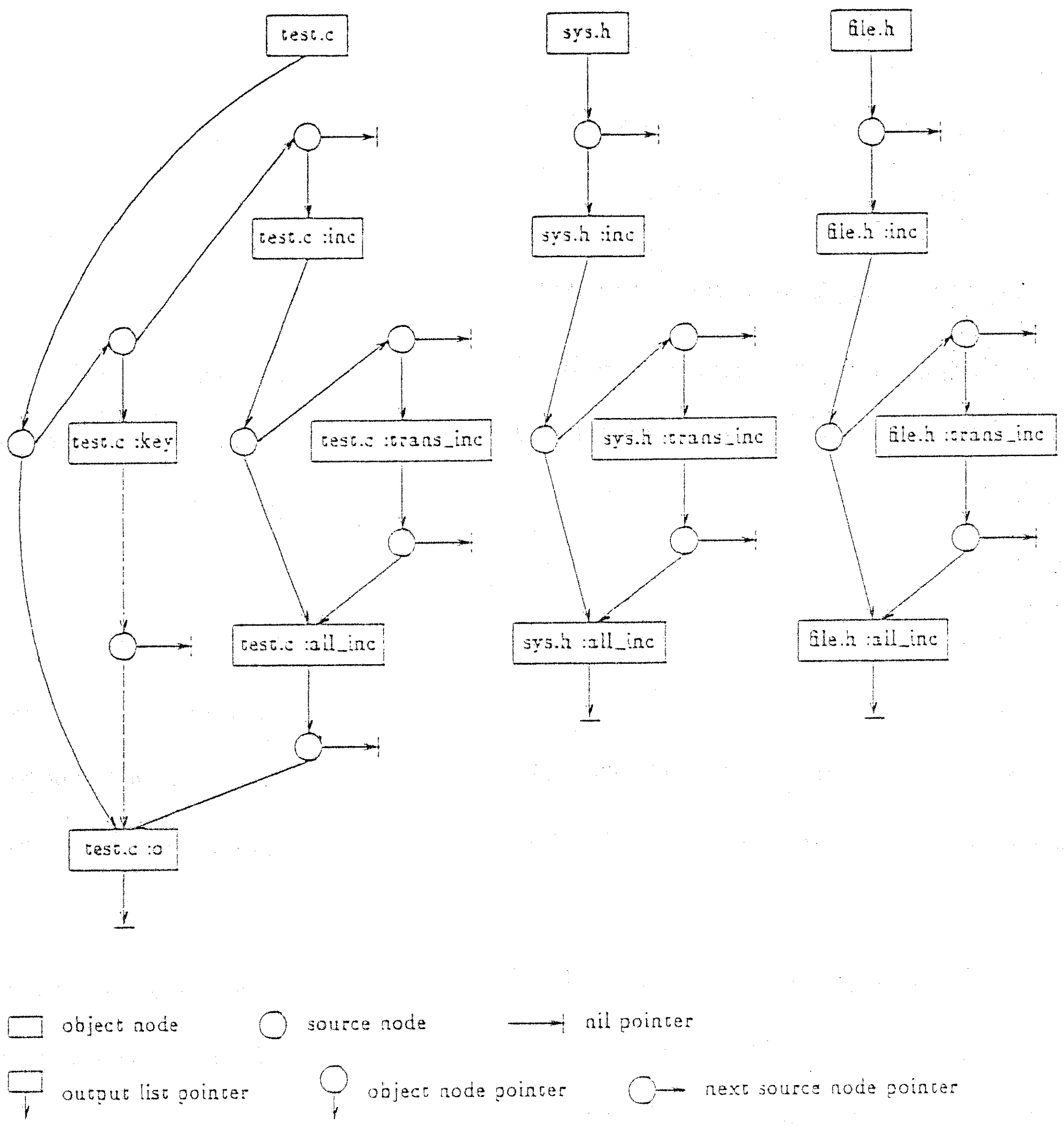
To allow convenient traversal of the source graph, source nodes are linked together through two kinds of lists, the Source List and the Output List. The Source List is a singly linked list of Source Nodes that specifies the complete set of objects needed as input to produce a given object. Each Object Node contains a pointer to the head of its Source List. The Output List is a doubly linked circular list of Source Nodes that specifies the inverse of the "source" relationship, namely, the complete set of objects that are produced by tools that use a given object as input. Each Object Node contains a pointer into its Output List.

A source node contains a pointer to its source Object Node and a pointer to its output Object Node. In addition it contains fields for implementing the Source List and Output List. The asymmetry in the implementation of Source Lists and Output Lists is because source nodes can be deleted from Output Lists but not from Source Lists. The doubly linked list implementation of Output Lists takes up more space but allows for more efficient implementation of this delete operation.

An example source graph is drawn in Figures 7.1 and 7.2. Figure 7.1 contains the Source Lists and Figure 7.2 contains the Output Lists.



Source Graph with Source Edges
Figure 7.1



Source Graph with Output Edges
Figure 7.2

7.2.4. Element Graph

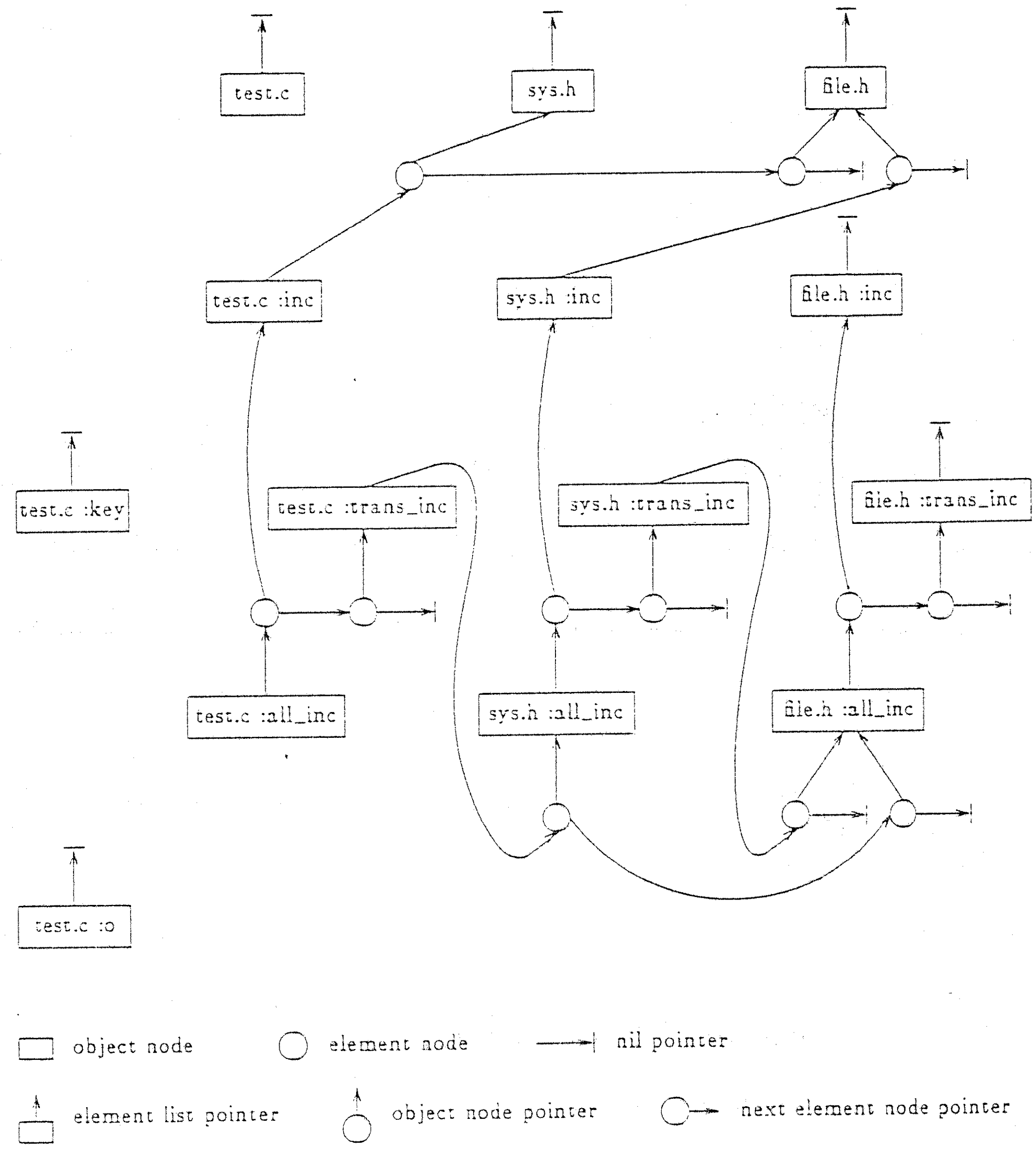
The object nodes are linked together via "element nodes" to form a directed (potentially cyclic) graph called the Element Graph. Each element node specifies an edge in the Element Graph. An edge in the element graph from object node X to object node Y indicates that the compound object corresponding to X has as an element the object corresponding to Y.

7.2.4.1. Element List and Compound List

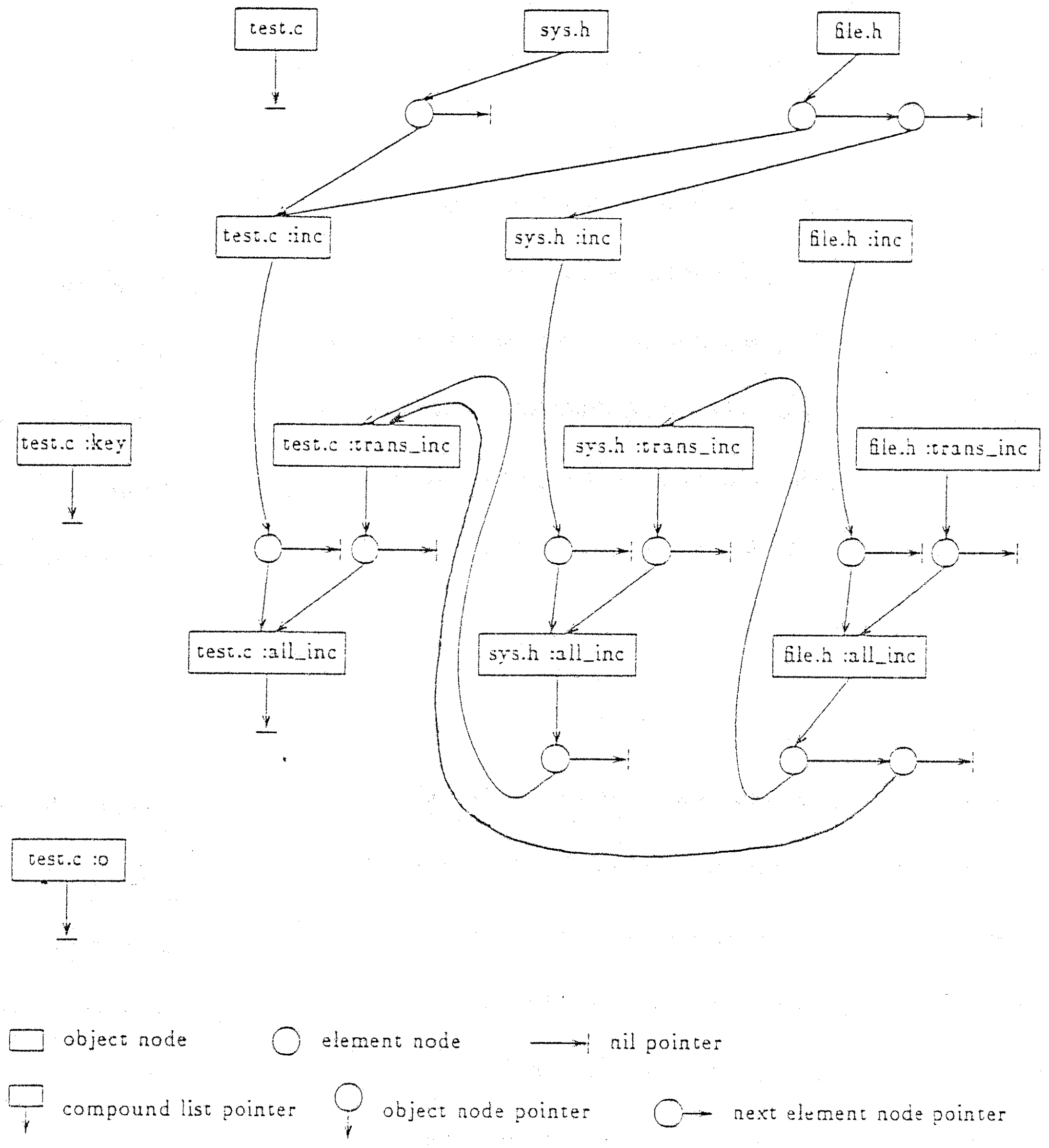
To allow convenient traversal of the element graph, element nodes are linked together through two kinds of lists, the Element List and the Compound List. The Element List is a singly linked list of Element Nodes that specifies the complete set of objects that are elements of a given compound object. Each Object Node contains a pointer to the head of its Element List. The Compound List is a doubly linked circular list of Element Nodes that specifies the inverse of the "element" relationship, namely, the complete set of compound objects that contain a given object. Each Object Node contains a pointer into its Compound List.

An element node contains a pointer to its element Object Node and a pointer to its compound Object Node. In addition it contains fields for implementing the Element List and Compound List. The asymmetry in the implementation of Element Lists and Compound Lists is because element nodes can be deleted from Compound Lists but not from Element Lists. The doubly linked list implementation of Compound Lists takes up more space but allows for more efficient implementation of this delete operation.

An example element graph is drawn in Figures 7.3 and 7.4. Figure 7.3 contains the Element Lists and Figure 7.4 contains the Compound Lists.



Element Graph with Element Edges
Figure 7.3



Element Graph with Compound Edges
Figure 7.4

7.2.5. Parameter Lists

Each Object Node contains a pointer to a (possibly empty) Parameter List that specifies the list of parameters that were used to produce that node from its Base Object. A Parameter List is implemented as a singly linked list of Parameter Nodes.

Each Parameter Node specifies a single parameter. It contains a field indicating the type of parameter, and two fields for storing the value of the parameter. A parameter may have as its value either a character string or an object. If the value of a parameter is a character string, this string is stored in the first value field; if the value is an object, a pointer to the appropriate object node is stored in the second value field. A Parameter Node also contains a field used to implement Parameter Lists.

7.2.6. LRU List - Automatic Space Maintenance

The Odin system maintains a data structure called the LRU (Least Recently Used) list. This list contains all object nodes for which objects exist. This list is used by the Odin system to determine which objects should be deleted when space is needed. Whenever a reference to an object is made, the object node associated with that object it is placed at the tail of the LRU list. Whenever the space occupied by derived objects is greater than the user specified maximum, object nodes at the head of the LRU list are removed and the corresponding objects are deleted until the space occupied by derived objects is less than the user specified maximum.

7.2.7. Concurrent Access

To ensure correct usage of the Odin database during multi-user concurrent access, some form of database locking is required. The observed usage pattern of the Odin system involves short bursts of database access followed by lengthy tool invocations or waits for further user requests. Thus, it has proven satisfactory in practice to lock the entire database for a given user while that user is accessing the database. The database is then unlocked when a tool is invoked or when the system prompts the user for additional input. Different usage patterns might require that the database locking take place at a finer level. Additional experimentation is required to determine if these different usage patterns arise in actual practice.

CHAPTER VIII

WRITING AN ODIN SPECIFICATION

In March of 1985, after the Odin system had been successfully used to integrate the Toolpack tool system and most common Unix tools, the design and implementation of Odin was frozen. This was the appropriate time to attempt to specify a completely new tool system. Unlike the previous systems integrated, such a new tool system might have characteristics and problems that were not being considered during the development of the Odin system. This test would thereby provide a qualitative measure of the flexibility of the Odin system.

The system selected for this test was the GAG attribute grammar system [Kastens82]. The GAG system takes as input an attribute grammar specifying a language analyzer and the code for an associated lexical analyzer, and produces a pascal program that performs the specified analysis. The core of the GAG system consists of sixteen executable tool fragments - thirteen tools that are invoked in sequence to produce the analyzer and three support tools that produce information about the attribute grammar. In addition, there are six support tools : two tools for producing a parser, a simple tool for producing a lexical analyzer, a tool for combining the generated Pascal program fragments into a complete program, and two host system dependent tools that produce an executable binary from the generated pascal analyzer and then run the executable analyzer on user specified input.

The GAG system test consisted of two distinct phases. In the first phase the Odin specification of the GAG system was designed to follow as closely as possible the way GAG is used outside of Odin. In the second phase, this specification was significantly modified to take advantage of the expressive power of the Odin specification language.

In both phases, the self-imposed constraint was that the executables for the GAG tool fragments must remain unchanged from the original system. The source code for the GAG tool fragments was in fact available, but restricting the specification to making use of the original tools provides a more severe test of the expressive power of the Odin specification language.

8.1. Phase 1 - Producing the Derivation Graph

When used outside of Odin, the GAG system is controlled by a large host system command script, with separate command scripts for some of the support tools. There are four basic operations in the GAG script :

(1) An executable tool is invoked.

(2) A file is linked to a new name.

(3) A file is copied to a new file.

(4) A file is deleted.

In addition to these operations, there are flow of control constructs that abort certain parts of the script after errors take place, and attempt to save information for later re-use. Since error-abort and re-use of information are automatically performed by the Odin system, these constructs can be ignored while producing the Odin specification.

The major work of producing an Odin specification consisted of determining what were the significant pieces of information produced by the GAG system, and specifying which tool fragment produces a given piece of information. In the simplest case, where a tool fragment creates a new file, it is trivial to determine the object and what tool produced it. In other cases, some analysis is required to identify what are the significant objects.

An Odin specification consists of two parts. First, there is the derivation graph describing the types of objects in the system, what tool fragment is needed to produce each type of object, and what types of objects are needed as input by a tool fragment. Second, there is a set of host system command scripts, one for each tool frsgment. For GAG, producing the host system command scripts is trivial, since most tool fragments consist of a few lines from the original GAG script. Producing the correct derivation graph requires some care, though, since the extensive moving, copying, and tool side effects can obscure the linkage between the tool fragments. In particular, the following situations require some care :

A file is modified by a tool :

In this case there are really two objects - the file before the tool invocation and the file afterwards.

A file is linked :

In this case there are two names for the same object.

A file is copied :

In this case there are two names for the same object, until one of the objects is modified in some way.

A file is deleted :

Any reference to the name of this file is illegal until some file is later linked or copied to this name.

8.1.1. Converting a Command Script

An Odin derivation graph is a "hyper-graph", where a hyper-graph is like a directed graph except that an edge connects a set of nodes to another set of nodes rather than just one node to another node. Each node in the derivation graph corresponds to an object type, a constant object, or a parameter object (see Chapter Five). An object type is used to represent an input file that is provided by the user or a file that is produced by a tool. A parameter object is an input file that is optionally provided by the user. A constant object is used to represent a file that is provided by the tool system (the tool writer or installer can modify this file, but a user cannot). Each hyper-edge then corresponds to a tool fragment.

A derivation graph can be produced from a command script by starting with a simple initial specification describing the input files, and then successively refining this specification based upon the sequence of operations in the command script. In order to produce the appropriate derivation graph, it is convenient to annotate nodes with the filenames used in the command script. These annotations will take the form of sets of sets of filenames. The inner sets (sets of filenames) will be "link-sets" and will indicate the set of filenames that are linked to a given object. A link-set can be thought of as a single object, where the filenames in the link-set indicate the set of distinct aliases for that object. The outer sets (sets of link-sets) will be "copy-sets" and will indicate the set of objects that are equal due being copies of each other.

Each operation in the command script will then have two potential effects on the graph : either introduce a new node or modify the annotations. Since the operations specified in the command script are in terms of filenames rather than objects, the annotations are necessary to provide a mapping from a command script filename to the appropriate node of the derivation graph.

8.1.2. Algorithm for Conversion

To initialize the derivation graph, create a distinct node for each file that is assumed to exist before the invocation of the script. Each node is annotated by a single name vector that contains the name of the corresponding input file. Then, for each line of the command script, modify the specification as follows :

File X is linked to file Y :

Add Y to the name vector that contains X. If there was no name vector containing X or if there was already a name vector containing Y, signal an error and abort.

File X is copied to file Y :

Delete the name vector containing Y from its current annotation set and add it to the annotation set that contains the name vector containing X. If there was no name vector containing Y, add a new name vector containing just Y to the annotation set of the name vector containing X. If there was no name vector containing X, signal an error and abort.

File X is deleted :

Delete X from the vector that contains it. If the vector is now empty, delete the vector from its annotation set.

Tool Z is invoked :

For each file that is created or modified by tool Z, create a new node in the derivation graph (a new object type). If a new file is being created, add a vector containing the name of this file to the new node. If an old file is modified, move the vector containing the name of this old file to the new node. Create a hyper-edge to the newly created nodes from the nodes that are annotated by vectors containing the names of the input files. Annotate this edge with the name of Z.

8.1.3. The GAG Command Script

The GAG command script that invokes the thirteen basic tools and the three auxiliary tools consists of 230 lines of Unix CSH commands (see Appendix A). The kinds of CSH commands that appear in this command script are :

#	: a comment
cp	: copy one file to another
ln	: line a file to a new name
rm	: remove (delete) a file
if ... endif	: conditionally execute a set of statements
toolname	: execute a tool (all toolnames begin with "$exec")

As indicated earlier, conditional execution constructs can be ignored. In particular, the construct

```
toolname || set continue = false
```

assigns the value "false" to the variable "continue" when "toolname" aborts, and then the outer conditionals of the form :

```
if ($continue) then
```

are used to skip later tools. The nested conditionals of this form are used to allow the re-use of information that is still valid from earlier runs. The Odin System automatically provides both of these capabilities, therefore the derivation graph can be designed with the assumption that "continue" is always "true".

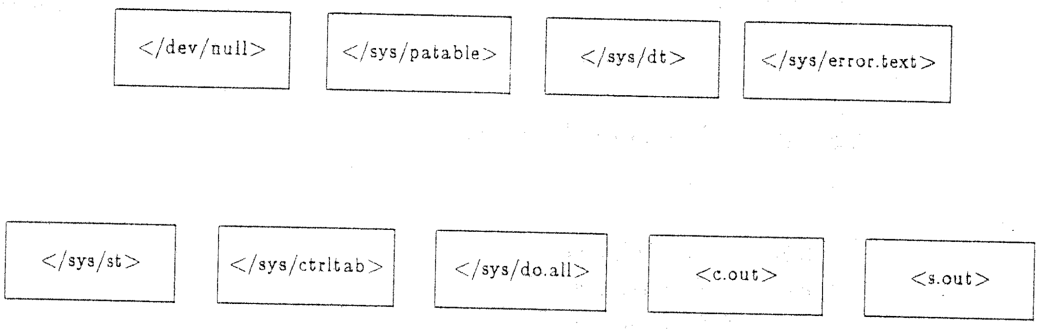
In order to demonstrate the process of producing a derivation graph, the initial derivation graph for the GAG system will be described and the modifications resulting from the first page of the GAG command script will be detailed. The complete derivation graph for the GAG command script can be found in Appendix B.

8.1.4. The GAG Derivation Graph

There are two user specified input files to the basic GAG system. The first is a file named "s.out" containing an attribute grammar written in the ALADIN language [Kastens84]. The second is a text file named "c.out" containing the list of options for this run of the GAG system. In addition to the user provided input, there are seven system provided inputs that are named "/dev/null", "/sys/st", "/sys/ctrltab", "/sys/do.all", "/sys/patable", "/sys/dt", and

"/sys/error.text". Therefore the initial derivation graph for the GAG system consists of two object type nodes and seven constant object nodes. The two object type nodes are annotated with a name vector consisting of the filename "s.out" and "c.out" respectively, while the six constant object nodes are each annotated with a name vector consisting of "/dev/null", "/sys/st", "/sys/ctrltab", "/sys/do.all", "/sys/patable", "/sys/dt", and "/sys/error.text" respectively. The resulting derivation graph appears in Figure 8.1.

This initial derivation graph is then incrementally refined based on each line of the command script. A description of this process applied to the first page of the command script follows (each line of the command script will be preceded with a line number).



Derivation Graph - Stage One
FIGURE 8.1

```
1:      # *********** execute control **********
```

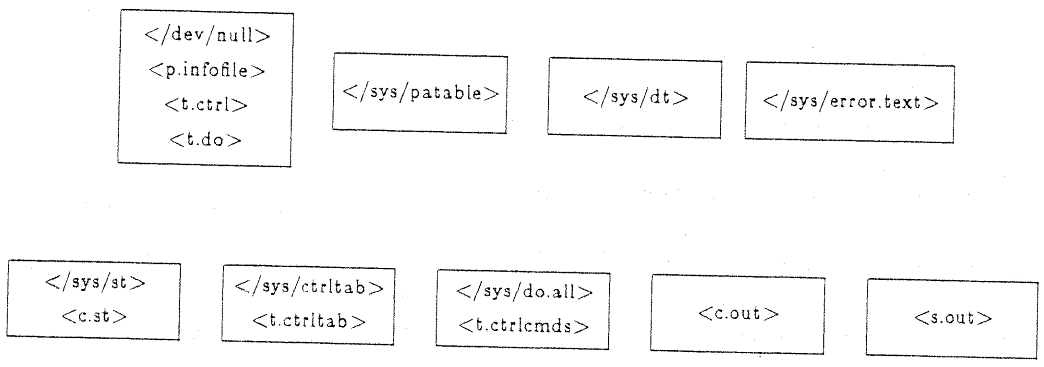
Since this line is a comment, it has no effect on the derivation graph.

```
2:      cp /sys/st      c.st
3:      cp /dev/null    p.infofile
4:      cp /sys/ctrltab         t.ctrltab
5:      cp /dev/null    t.ctrl
6:      cp /sys/do.all  t.ctrlcmds
7:      cp /dev/null    t.do
```

These copy commands have the following effects. Add a name vector containing "c.st" to the node that contains the name vector containing "/sys/st". Add a name vector containing "p.infofile" to the node that contains the name vector containing "/dev/null". Add a name vector containing "t.ctrltab" to the node that contains the name vector containing "/sys/ctrltab". Add a name vector containing "t.ctrl" to the node that contains the name vector containing "/dev/null". Add a name vector containing "t.ctrlcmds" to the node that contains the name vector containing "/sys/do.all". Add a name vector containing "t.do" to the node that contains the name vector containing "/dev/null". The resulting derivation graph appears in Figure 8.2.

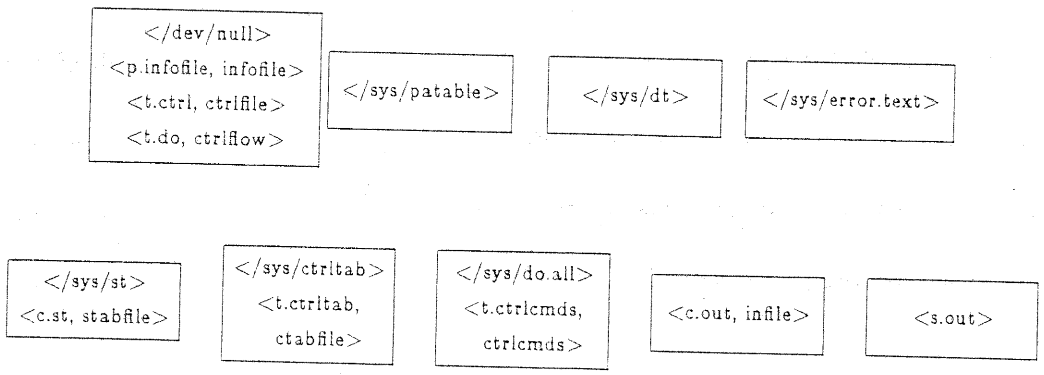
```
8:      ln p.infofile   infofile
9:      ln t.ctrltab    ctabfile
10:     ln c.st  stabfile
11:     ln t.ctrl       ctrlfile
12:     ln t.ctrlcmds   ctrlcmds
13:     ln t.do ctrlflow
14:     ln c.out        infile
```

These link commands have the following effects. Add "infofile" to the name vector containing "p.infofile". Add "ctabfile" to the name vector containing "t.ctrltab". Add "stabfile" to the name vector containing "c.st". Add "ctrlfile" to the name



Derivation Graph - Stage Two
FIGURE 8.2

vector containing "t.ctrl". Add "ctrlcmds" to the name vector containing "t.ctrlcmds". Add "ctrlflow" to the name vector containing "t.do". Add "infile" to the name vector containing "c.out". The resulting derivation graph appears in Figure 8.3.



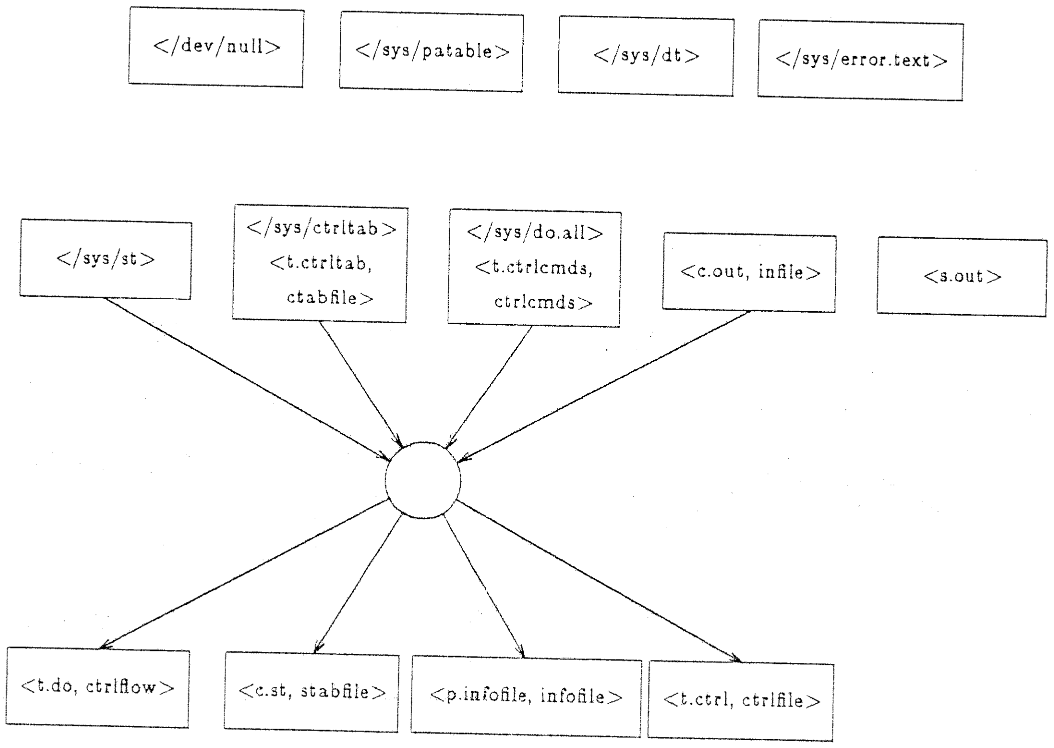
Derivation Graph - Stage Three
FIGURE 8.3

15: /exec/ctrl || set continue = false

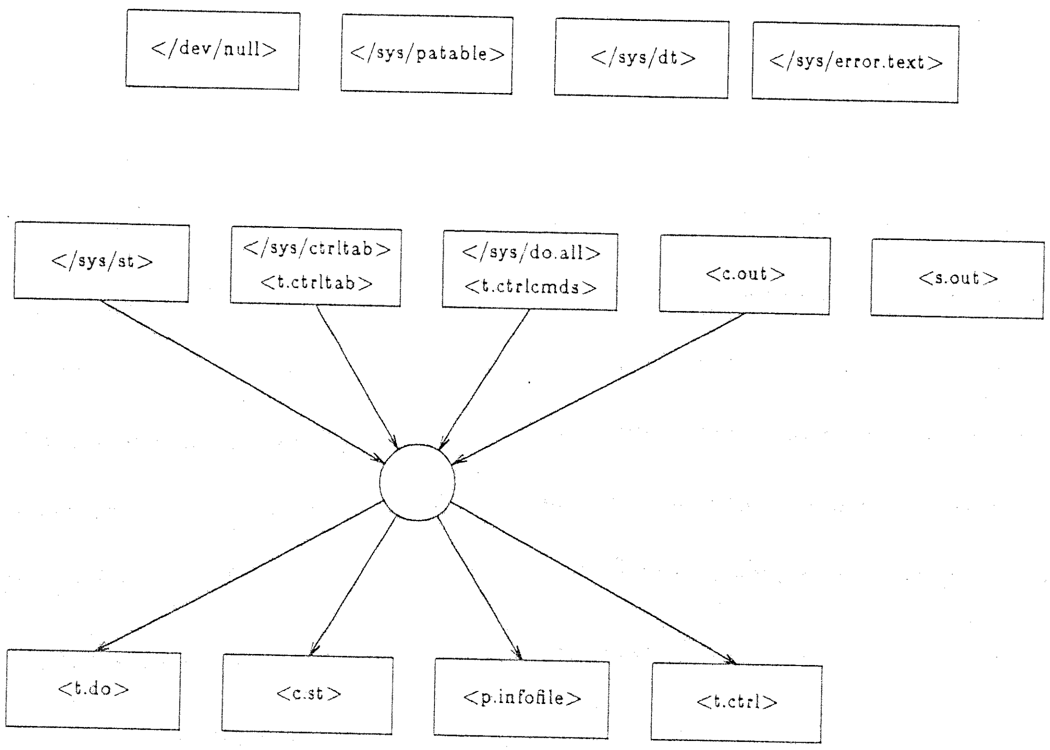
In order to determine the effect of "/exec/ctrl", either documentation must exist that describes the tool (possibly the source code itself) or the tool must be tested to determine which objects it uses as input, which new objects it creates, and which old objects it modifies. Using a combination of these two techniques, it was determined that "/exec/ctrl" uses as input the files "infile", "stabfile", "ctabfile", and "ctrlcmds". It produces no new files, but modifies "ctrlflow", "stabfile", "infofile", and "ctrlfile". This results in the addition of four new object type nodes, where the name vectors containing the filenames of the modified files are moved from the nodes that previously contained them to the appropriate new nodes. These name vectors are <t.do, ctrlflow>, <c.st, stabfile>, <p.infofile, infofile>, and <t.ctrl, ctrlfile>, respectively. Finally, a hyper-edge is created from the four input nodes to the four output nodes. The resulting derivation graph appears in Figure 8.4.

16: rm infofile ctabfile stabfile ctrlfile ctrlcmds ctrlflow infile

The seven specified filenames are removed from their respective name vectors. In each case, at least one filename remains in each name vector, therefore no name vectors are deleted. The resulting derivation graph appears in Figure 8.5.



Derivation Graph - Stage Four
FIGURE 8.4

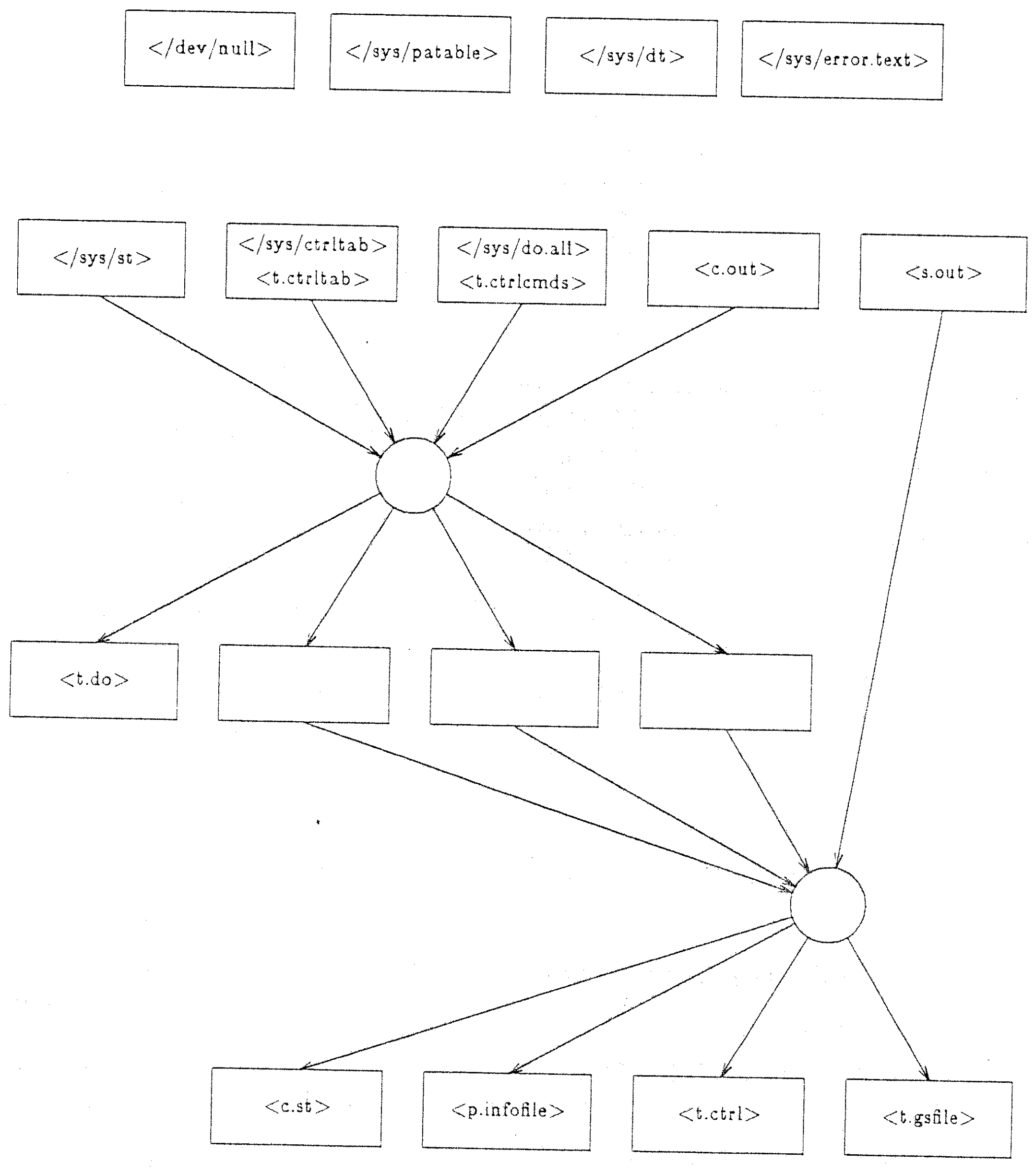


Derivation Graph - Stage Five
FIGURE 8.5

```
17:    # *********** execute scanner **********
18:    if ($continue) then
19:            cp /dev/null    t.gsfile
20:            ln p.infofile   infofile
21:            ln c.st  stabfile
22:            ln s.out        infile
23:            ln t.ctrl       ctrlfile
24:            ln t.gsfile     gsfile
25:            /exec/scanner || set continue = false
26:            rm infofile stabfile infile ctrlfile gsfile
27:            endif
```

As before, the comment has no effect on the derivation graph. The conditional statement is assumed to succeed, since the Odin system will automatically handle the situation where a tool fails. The copy command adds a vector containing "t.gsfile" to the node containing "/dev/null". The link commands add "infofile", "stabfile", "infile", "ctrlfile", and "gsfile" the name vectors containing "p.infofile", "c.st", "s.out", "t.ctrl", and "gsfile" respectively. The tool "/exec/scanner" uses as input the files "infofile", "stabfile", "infile", and "ctrlfile". It produces no new files, but modifies "infofile", "stabfile", "ctrlfile", and "gsfile". This causes the creation of four new object type nodes, and a new hyper-edge connecting the four input nodes to the four output nodes of the scanner. Finally, the remove command deletes "infofile", "stabfile", "infile", "ctrlfile", and "gsfile" from their respective name vectors. The resulting derivation graph appears in Figure 8.6.

This derivation graph can then be described in the Odin specification language (see Chapter Five). Nodes that correspond to user created input objects are declared to be ATOMIC. Nodes that correspond to constant objects are referred to directly by the Odin scripts, and therefore do not need to be declared. Each edge (a tool) is then declared as a production, where this production contains a declaration for each of the objects produced by that tool. The textual representation for the derivation graph of Figure 8.6 appears in Figure 8.7. The choice of names for the object types is arbitrary, but in general the annotations of the node that corresponds to the object type were used as a guide. If the filenames in the script seemed to be poorly chosen, simple mnemonic names were used. For example, "ala" and "opt" were thought to be more intuitive names for the two atomic types than "s.out" and "c.out". The remainder of the GAG command script is processed in an analogous fashion. The resulting derivation graph can be found in Appendix B.



Derivation Graph - Stage Six
FIGURE 8.6

```
scan <
  scan_stab "symbol table"
  scan_gs "global symbols"
  scan_i "info file"*
  scan_c "ctrl file"*
  > "lexical analysis" :
      USER scan.cmd
             : ctrl_stab
             : ala
             : ctrl_i : ctrl_c

ctrl <
  ctrl_cmds "host system commands"
  ctrl_stab "initial symbol table"
  ctrl_i "info file"*
  ctrl_c "ctrl file"*
  > "control analysis" :
      USER ctrl.cmd
             : opt

opt ATOMIC "GAG options"
ala ATOMIC "ALADIN grammar"
```

Textual Derivation Graph
FIGURE 8.7

8.2. Phase 2 - Improving Efficiency

8.2.1. Methods of Optimization

There are three main methods that can be used to improve the runtime efficiency of an existing tool system when it is integrated under the Odin system. In each of these methods the improvement in runtime efficiency is due to increased re-use of previously computed objects.

8.2.1.1. Abstraction

The first method is to introduce tools to generate intermediate objects that are abstractions of the source objects, where an abstraction is an object that can remain unchanged when an object from which it is derived changes. The Odin system understands that if an abstraction is not affected by a source level modification, then any objects previously derived from that abstraction are still valid.

For example, a tool could be written that strips out the comments from a source object containing program source. The derived object containing executable binary could then be derived from the "de-commented" abstraction. If a comment in the source object were changed, the Odin system would generate the de-commented abstraction, notice that this abstraction has not changed from the previous version, and therefore mark the executable binary from the old version as still valid.

8.2.1.2. Partitioning

The second method is to introduce a tool to automatically partition an existing object, and then apply later tools to the elements of the partition, re-using objects that are derived from elements of the partition that have not been affected by source level modifications.

For example, a source object for a programming language that provides facilities for separate compilation often can be partitioned into several smaller objects (such as procedures), each of which could be compiled separately. A tool fragment can usually be provided that will automatically perform this partitioning. When the source object is to be compiled, first it is automatically partitioned into objects that each contain a single procedure, and then each procedure is compiled separately. When the source object is modified and then is to be compiled, again it is first automatically partitioned, and only those procedures that have been modified will be recompiled. For the unmodified procedures, the previously computed compilations will still be valid.

8.2.1.3. Parameterization

The third method is to identify objects that contain default information that can be optionally modified by the user when making requests. These objects can be partitioned into "parameters". The types of parameters that are of interest to a given tool are specified in the PARAMETER list of the specification for that tool (see Chapter Five), and the values for parameters are specified by the user at run time using the parameterization operation (see Chapter Four). The benefit of parameterization is that there frequently are intermediate objects that are not affected by the specified parameters and therefore can be re-used in several different parameterized queries.

For example, in a system that has tools for parsing programs and for pretty-printing parse trees, formatting instructions can be embedded in comments in the source code. These instructions do not affect the parse, but are passed on to the pretty-printer through the parse tree. If these formatting instructions were removed from the source code and placed in parameters, the same parse tree could be used for several different parameterized pretty-printing requests.

8.2.2. Optimizing the GAG Specification

The first method, abstraction, is the most difficult to apply to existing tool fragments. Extensive knowledge of both the data structures being produced and the expected usage of the system is usually necessary before significant abstractions could be generated. In some cases though, a data structure is passed to a tool when that tool does not in fact make use of any information in that data structure. In these cases, a simple form of abstraction consists of eliminating the superfluous inputs. In the GAG system, such a situation arises with a sequence of "info" files. The purpose of an "info" file is to collect from each pass the messages containing information for the user. Each tool takes as input the "info" file from the preceding tool, appends any messages that it generates, and passes the list on to the next tool. This list is finally given to the "protocol" tool that generates a readable version of the messages. The result of this approach is that whenever a tool generates new messages, it appears that all later tools must be rerun since one of their inputs, namely the list of messages, has changed. Since a tool never uses the information in this list, but only appends new information to it, there is no need for this list to be an input to any tool except for the "protocol" tool. In addition, since the Odin system has an internal tool, ERROR, that collects together the error messages from a series of tools, a simple "protocol" phase can be done to produce readable messages immediately following each GAG tool invocation, and then the ERROR tool would collect the appropriate messages together to produce a full error report.

The second method, partitioning, is applicable if there exists significant segmentation at the source level that is reflected in the intermediate data objects.

In the case of the GAG system, we were not able to identify any significant partitioning that would not involve extensive modification to the existing tools.

The third method, parameterization, is usually applicable with a minimal amount of effort. In most tool systems, some set of flags or options are provided to modify the behavior of the tool system. In the Odin specification, each type of flag or option can be specified as a distinct parameter type, and the specification of each tool would be extended to specify which parameters are of interest to that tool.

In the original GAG system, the options to all of the tool fragments are stored together in a single file. This file is processed by the first tool fragment to produce a "control file" that is passed to each of the succeeding tool fragments. Each tool fragment then extracts the values of options that are of interest. Since a large number of options to the GAG system only affect the results of the later tool fragments, specifying each kind of option as a separate parameter type in the derivation graph can provide significant increases in re-use of previously computed objects. For example, if a user makes several requests that differ only in the options passed to the cross reference tool, the Odin system would re-use all analysis and simply rerun the cross reference tool with the various parameters.

In order to allow the Odin System to control the distribution of options, each option type is assigned a specific parameter name. Since each GAG option has the syntax :

```
option_name option_value ;
```

the simplest way to assign a parameter names is to use the GAG option_name as the parameter name. Each tool specification in the derivation graph for GAG would then be extended with a "PARAMETERS" input line (see Chapter Five) that would contain the list of parameter names whose values are of interest to that tool.

8.2.2.1. Optimization Example

The optimizations described in the previous section can be applied to each of the tools specified in the derivation graph for GAG. Since each of these tool specifications will be modified in a similar fashion, we will consider just one of the tools, the "expand" tool.

In the original derivation graph, the "expand" tool was specified as :

```
expand <
  expand_sem "expanded semantics"
  expand_sx "expanded symbols"
  expand_def "expanded definitions"
  expand_i "info file"*
  expand_c "ctrl file"*
  > "ALADIN shorthand expansion" :
       USER expand.cmd
            : seman_sem
            : seman_sx
            : seman_def
            : seman_i : seman_c
```

In the optimized derivation graph, the "seman_c" and "expand_c" objects, which contain the option information, would be replaced with a PARAMETERS specification. In addition, the "seman_i" and "expand_i" objects, which pass along the "info" messages, would be eliminated. A new "scan_stab" input is necessary to make the messages produced by the expand tool readable. These messages would then be collected for display to the user by the Odin System's internal ERROR and WARNING tools. The modified specification for the "expand" tool would then be :

```
expand <
  expand_sem "expanded semantics"
  expand_sx "expanded symbols"
  expand_def "expanded definitions"
  > "ALADIN shorthand expansion" :
       USER expand.cmd
            : seman_sem
            : seman_sx
            : seman_def
            : scan_stab
            : PARAMETERS < expand interface >
```

After the derivation graph was optimized, specifications for the auxiliary tool fragments that generated a scanner, generated a parser, compiled the resulting analyzer, and tested the analyzer, were added. The complete modified derivation graph for the GAG system appears in Appendix C.

8.3. User Experience with GAG in Odin

The main purpose of the GAG experiment was to test the flexibility of the Odin specification language. Since both the unoptimized and optimized Odin specifications captured the full GAG system without modification to the Odin System, the experiment was successful. In addition, since no modifications to the GAG tools were required, the only effort expended was the translation of the GAG command script into an Odin Derivation Graph. This translation took place over the course of a week, with the majority of the time devoted to experimenting with possible optimizations to the specification.

With GAG fully specified in a Derivation Graph, it would then be interesting to determine whether the Odin-embedded GAG system is an improvement over the stand-alone GAG system. The only significant difference to a GAG user between the Odin-embedded GAG and the stand-alone GAG is efficiency. In the unoptimized form of the Odin specification, very few objects could be re-used, and therefore very little execution time efficiency improvement over the stand-alone system could be noticed. In the optimized form of the Odin specification though, the efficiency improvements were considerable. For example, a request for an alternative analyzer with modified "codegen" parameters required only 10% of the time required for this request from the stand-alone system.

A measurement of the average expected improvement in efficiency for a normal user of the GAG system could not be made, because the only local user of the GAG system at the time of the experiment was involved in modifying the GAG system, not in using it to produce analyzers. The presense of this kind of user, though, did allow a qualitative measurement of the benefits of using the Odin System from the viewpoint of a tool builder. A report concerning the use of GAG embedded in Odin [Gray85] indicated that the use of an object manager like Odin is essential to properly control the development of a complex system like the GAG. "The non-Odin version was such an uncontrollable mess. ... I think anyone needing GAG would use Odin/GAG over the old way" [Gray85]. As a result of this experience, the research group studying GAG at the University of Colorado will be using Odin to support further development of the GAG system.

CHAPTER IX

CONCLUSIONS

9.1. Contributions

9.1.1. Process View vs. Object View

The focus of many software environments is on the processes or tools that are provided by the environment. This focus encourages a user to concentrate on the processes being performed in the environment - which processes are available, how they are invoked, and how they are controlled once they have been invoked. An alternative focus is on the objects or data in the environments - both the objects manipulated directly by the user via editors, and derived objects generated by tools from other objects.

For some tools, the process viewpoint is the appropriate one. These tools are the interactive tools that maintain a continuous dialogue with the user. For example, in a browsing tool it is the process of gathering information that is of paramount interest to the user, and therefore it is this process which should be the focus of the user. In an editing tool the process of modifying information is combined with the information gathering of a browser, and again, this process should be the focus of the user.

For many tools though, the process viewpoint is inappropriate. These tools are non-interactive tools that function without user intervention once the input objects have been provided. For these tools, the data produced by the tool is what is of interest, not how the tools produce this data. For example, when test data regression analysis is being performed, what is of interest is the output data resulting from running the modified programs on existing test data - not how this output data is computed. In particular, the user has no need of following the details of program compilation, linking to produce executables, setting up a runtime environment, and capturing the appropriate output data. These details add complexity to the use of the environment without adding a corresponding amount of utility. This dissertation demonstrates that with the appropriate underlying mechanisms, a software environment can support and encourage the object viewpoint, thereby eliminating much of the complexity that results from inappropriate use of the process viewpoint.

9.1.2. Specification Language

A specification language for tools is one of the central underlying features that must be provided in order for a software environment to support an object view of the software environment. If the user is no longer required to specify which processes are to be invoked in order to produce a desired object, there must be an alternative means whereby the environment can determine this information. A specification language that describes the available tools and the kinds of objects produced by these tools provides this alternative. In this dissertation, an extended production system language called the Odin

Specification Language has been developed to serve this function for a software environment. This specification language has been used to describe existing collection of tools, such as those provided with the Berkeley Unix operating system, and tool systems under development, such as those of the Toolpack project. The dissertation provides an example of the construction of an Odin Specification for a non-trivial tool collection (the GAG Attribute Grammar System), and describes how the Odin Specification and the tool system can be modified for improved runtime efficiency.

9.1.3. Query Language

In order to allow a user to specify objects rather than processes that create objects, a query language must be provided with which a user can request any software object of interest. The Odin System provides such a query language that is based on the orthogonal combination of three basic principles : derivation, parameterization, and selection. Derivation is used to request a modified form of the input object; parameterization is used to extend an object with additional information; and selection is used to access a specific part of a larger object. The actual kinds of modification, addition, and selection that can be applied in a given tool system are determined from the Odin Specification for that tool system.

9.2. Future Research

9.2.1. Graphic Interface for Interactive Queries

Once an object manager such as the Odin System is in place, the user can focus on the creative processes of browsing and editing (as opposed to compiling and loading). One way to support these processes is to provide a sophisticated graphical interface to the object manager that displays the relationships between objects in order to help guide a user during the browsing process. Since the Odin System maintains a spanning tree for the complex graph of object relationships, it would be feasible to use this underlying tree structure to guide the graphical display of objects in the Odin System. With this information presented on a bit-mapped graphics display, a pointing device such as a mouse could be used to negotiate through the graph of objects, with menus providing the appropriate guidance. The message passing syntax of the Odin Query language was specifically designed to allow such a menu-driven traversal. For example, the query

```
test.c :fmt :output
```

could be generated through a graphic interface by first pointing at the object test.c. A menu of messages that could be sent to "test.c" would then pop up, from which the user could select ":fmt". This selection results in the cursor being positioned at the object "test.c :fmt", which would then cause a new menu of messages to pop up, one of which would be ":output". When the user selects ":output", the cursor would be positioned at the object "test.c :fmt :output". Requests for selections and parameterizations of objects would then be performed in an analogous fashion, with menus popping up at the appropriate times to indicate the set of possible messages. In the current implementation, these menus can be explicitly requested through the Odin Help facility, therefore providing this graphical interface would not require extensive modification to the existing Odin implementation.

9.2.2. Alternative and Constrained Derivation Paths

The Odin System determines from the Derivation Graph a single derivation path from a given kind of object to another kind of object. For example, the path from a "program" object to an "output" object might consist of compiling the "program" object, loading the "compiled" object, and then running the "loaded" object. This is in contrast to the more common use of production systems where the interpreter explores all possible paths from a given object to another object, until at least one "successful" path is found.

One reason that the Odin System does not explore alternative paths is that exploring a given path is usually an expensive process. For example, suppose there were three distinct compilers that could be used, where each compiler could handle a slightly different variant of the same basic language. If a program has an error in it, a user would probably not be willing to wait until the system has tried each of the three compilers before being informed of an error.

Another reason for not exploring alternative paths is that in practice, few of the alternative paths in a software environment are reliably equivalent. Again using the example of the three compilers, in spite of all beliefs or hopes to the contrary, it is likely that some program compiled by one compiler will execute differently than that same program compiled by another compiler. A user might modify the program and find that it produces unexpected results, not because of the user modifications, but because a different compiler was used.

In spite of these problems, it still would be interesting to explore the use of alternative paths, especially in the context of a distributed environment. In such an environment, a given task can often be performed on some remote machine rather than the local machine. It might be appronriate for these alternatives to be represented as alternative paths through a Derivation Graph, and have the object manager choose an appropriate path.

An extension to the Odin Derivation Graph that might alleviate the problems with exploring alternative paths would be to introduce the notion of a set of "constraints" that apply to a derivation in the Derivation Graph. A constraint would describe the situations under which a given derivation is applicable. Assuming that these constraints are cheaper to compute than the derivation itself, they could be used to drive the choice of which path to use, and thereby lessen the expense of exploring alternative paths. In addition, these constraints could be specifically designed to eliminate paths that do not produce equivalent results.

9.2.3. Object Granularity The Odin System was designed for the manipulation of comparatively large objects such as procedures, modules, chapters, or larger objects that are composed from these components, such as programs or books. An interesting question is whether an object manager such as the Odin System is appropriate for smaller objects such as individual tokens or words.

9.2.4. Co-operating Object Managers

When the Odin System is used on a network of machines that do not share a common file system, it is necessary for each machine to maintain a local object store. With the current implementation of the Odin System, the robustness and local control benefits that accrue from maintaining separate object stores motivate the existence of separate object stores for separate projects even when a common file system exists. The robustness benefits occur because damage to the object store of one project does not carry over to another project with a separate object store. The local control benefits occur because one project can customize the Derivation Graph and the various Odin System parameters such as allowed file space usage without affecting those of another project. These separate projects would still like to share information in various ways, and therefore the question arises of how to provide co-operation among these autonomous object managers. Currently, sharing is provided through the use of tool fragments that request objects from other object stores. It would be interesting to determine if improved sharing could be achieved by directly supporting the sharing in the object manager itself.

BIBLIOGRAPHY

[Anderson79]
Anderson, R. B., Proving Programs Correct, John Wiley and Sons, 1979.

[Avakian82]
Avakian, A., S. Haradhvala, Julian Horn, and B. Knobe, "The Design of an Integrated Support Software System," Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, pp. 308-317, June 1982.

[Bazelmans85]
Bazelmans, R., "Evolution of Configuration Management," SIGSOFT Software Engineering Notes, vol. 10, no. 5, pp. 37-46, October 1985.

[Cargill79]
Cargill, T. A., "A View of Source Text for Diversely Configurable Software," Technical Report CS-79-28, Department of Computer Science, University of Waterloo, 1979.

[Chapin74]
Chapin, N., "A New Format for Flowcharts," Software - Practice and Experience, vol. 4, no. 4, pp. 341-357, 1974.

[Cristofor80]
Cristofor, E., T. A. Wendt, and B. C. Wonsiewicz, "Source Control + Tools = Stable Systems," Proceedings of the Fourth Computer Software and Applications Conference, pp. 527-532, October 1980.

[Bratman75]
Bratman, H. and T. Court, "The Software Factory," Computer, vol. 8-5, May 1975.

[Cooprider79]
Cooprider, L., "The Representation of Families of Software Systems," Doctoral Dissertation, Computer Science Department, Carnegie-Mellon University, April 1979.

[CDC76]
"Modify Reference Manual," Publication Number 60281700, Control Data

Corporation, 1976.

[DEC84a]
CMS Code Management System, Digital Equipment Corporation, 1984.

[DEC84b]
MMS Module Management System, Digital Equipment Corporation, 1984.

[DeJong73]
DeJong, S. P., "The System Building System," Technical Report RC 4486, Thomas J. Watson Research Center, 1973.

[Erikson84]
Erikson, V. B. and J. F. Pelligrin, "Build - A Software Construction Tool," AT&T Bell Laboratories Technical Journal, vol. 63, no. 6, pp. 1049-1059, August 1984.

[Feldman79]
Feldman, S. I., "MAKE - A Program for Maintaining Computer Programs," Software - Practice and Experience, vol. 9, pp. 255-265, 1979.

[Glasser78]
Glasser, A. L., "The Evolution of a Source Code Control System," Software Engineering Notes, vol. 3, no. 5, pp. 122-125, ACM, November 1978.

[Goldberg83a]
Goldberg, A., Smalltalk-80: The Interactive Programming Environment, Addison-Wesley, 1983.

[Goldberg83b]
Goldberg, A., Smalltalk-80: The Language and its Implementation, Addison-Wesley, 1983.

[Gray85]
Gray, B., "Experiences with GAG and Odin," Private Communication, December, 1985.

[Habermann79]
Habermann, A. N., "A Software Development Control System," Computer Science Dept. Technical Report, Carnegie-Mellon University, Pittsburgh, 1979.

[Huff81]
Huff, K., "A Database Model for Effective Configuration Management in the

Programming Environment," Proceedings Fifth International Conference on Software Engineering, pp. 54-61, March 1981.

[Ince83]
Ince, D. C., "A Compatibility Software Tool for Use With Separately Compiled Languages," SIGPLAN Notices, vol. 18, no. 9, pp. 31-34, Setptember 1983.

[Kastens82]
Kastens, U., B. Hutt, and E. Zimmermann, GAG: A Practical Compiler Generator, Springer, 1982.

[Kastens84]
Kastens, Uwe, "The GAG-System - A Tool for Compiler Construction," in Methods and Tools for Compiler Construction, ed. B. Lorho, pp. 165-181, Cambridge University Press, 1984.

[Kernighan78]
Kernighan, B. W. and D. M. Ritchie, The C Programming Language, Prentice-Hall, Englewoods Cliffs, New Jersey, 1978.

[Lampson83]
Lampson, B. and E. Schmidt, "Practical Use of a Polymorphic Applicative Language," Proceedings of the Tenth POPL Conference, January 1983.

[Lampson83]
Lampson, B. and E. Schmidt, "Organizing Software in a Distributed Environment," SIGPLAN Notices, vol. 18, no. 6, June 1983.

[Leblang84]
Leblang, D. B. and R. P. Chase, "Computer-Aided Sortware Engineering in a Distributed Workstation Environment," SIGPLAN/SIGSOFT Symposium on Practical Software Development Environments, ACM, April 1984.

[Leblang85a]
Leblang, D. B. and G. D. McLean, "Configuration Management for Large-Scale Software Development Efforts," Workshop on Software Engineering for Programming-in-the-Large, Harwichport, June 1985.

[Leblang85b]
Leblang, D. B., R. P. Chase, and G. D. McLean, "The DOMAIN Software Engineering Environment for Large Scale Software Development Efforts." Proceedings of the IEEE Conference on Workstations, San Jose, November 1985.

[Linton84]
Linton, M. A., "Implementing Relational Views of Programs," SIGPLAN/SIGSOFT Symposium on Practical Software Development Environments, ACM, April 1984.

[Metzner77]
Metzner, J. R. and B. H. Barnes, Decision Table Languages and Systems, Academic Press, New York, 1977.

[Miller79]
Miller, E., Tutorial: Automated Tools for Software Engineering, IEEE Computer Society Press, 1979.

[Montalbano74]
Montalbano, M., Decision Tables, Science Research Associates, Palo Alto, 1974.

[Myers78]
Myers, G., Composite Structured Design, Van Nostrand, 1978.

[Nassi73]
Nassi, I. and B. Shneiderman, "Flowchart Techniques for Structured Programming," SIGPLAN Notices, vol. 8, no. 8, pp. 12-26, August 1973.

[Osterweil82]
Osterweil, L. J., '"Toolpack - An Experimental Software Development Environment Research Project," Proceedings 6th International Conference on Software Engineering, pp. 166- 175, Tokyo, 1982.

[Ramamoorthy76]
Ramamoorthy, C. V., S. F. Ho, and W. T. Chen, "On the Automated Generation of Program Test Data," IEEE Transactions on Software Engineering, vol. 2, no. 4, pp. 293-300, December 1976.

[Ritchie74]
Ritchie, D. M. and K. Thompson, "The UNIX Time-Sharing System," Communications of the ACM, vol. 17, no. 7, pp. 364-375, July 1974.

[Rochkind75]
Rochkind, M. J., "The Source Code Control System," IEEE Transactions on Software Engineering, vol. SE-1, no. 4, pp. 364-370, December 1975.

[Rudmik82]
Rudmik, A. and B. Moore, "An Efficient Separate Compilation Strategy for

Very Large Programs," Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, pp. 301-307, June 1982.

[Schmidt82]
Schmidt, E. E., "Controlling Large Software Development in a Distributed Environment," Doctoral Dissertation, University of California, Berkeley, December 1982.

[Teitelman81]
Teitelman, W. and L. Masinter, "The Interlisp Programming Environment," Computer, vol. 14, no. 4, pp. 25-34, April 1981.

[Thall83]
Thall, R., "Large-Scale Software Development with the Ada Language System," Proceedings of ACM Computer Science Conference, February 1983.

[Tichy82]
Tichy, W. F., "Design, Implementation, Evaluation of a Revision Control System," Proceedings Sixth International IEEE Conference on Software Engineering, pp. 58-67, September 1982.

[White77]
White, J. R. and R. K. Anderson, "Supporting the Structured Development of Complex PL/I Software Systems," Software - Practice and Experience, no. 7, pp. 279-293, 1977.

[Yourdon78]
Yourdon, E. and L. L. Constantine, Structured Design: Fundamentals of a Discipline of Computer Program and System Design, Yourdon Press, New York, 1978.

APPENDIX A

GAG Command Script

```
# *********** execute control **********
cp /sys/st      c.st
cp /dev/null    p.infofile
cp /sys/ctrltab t.ctrltab
cp /dev/null    t.ctrl
cp /sys/do.all  t.ctrlcmds
cp /dev/null    t.do
ln p.infofile  infofile
ln t.ctrltab   ctabfile
ln c.st        stabfile
ln t.ctrl      ctrlfile
ln t.ctrlcmds  ctrlcmds
ln t.do        ctrlflow
ln c.out       infile
/exec/ctrl || set continue = false
rm infofile ctabfile stabfile ctrlfile ctrlcmds ctrlflow infile

# *********** execute scanner **********
if ($continue) then
   cp /dev/null  t.gsfile
   ln p.infofile infofile
   ln c.st       stabfile
   ln s.out      infile
   ln t.ctrl     ctrlfile
   ln t.gsfile   gsfile
   /exec/scanner || set continue = false
   rm infofile stabfile infile ctrlfile gsfile
   endif
```

```
# *********** execute parser ***********
if ($continue) then
   cp /dev/null t.pafile
   cp /sys/patable zerdat
   ln p.infofile  infofile
   ln t.ctrl      ctrlfile
   ln t.gsfile    gsfile
   ln t.pafile    paout
   /exec/parser || set continue = false
   rm infofile ctrlfile gsfile paout zerdat
   endif


# *********** execute makedef ***********
if ($continue) then
   cp /sys/dt p.dt
   ln p.infofile  infofile
   ln p.dt        deffile
   ln t.ctrl      ctrlfile
   ln t.pafile    pafile
   /exec/makedef || set continue = false
   rm infofile deffile ctrlfile pafile
   endif


# *********** execute semantic ***********
if ($continue) then
   cp /dev/null p.semfile
   cp /dev/null p.sxfile
   cp /dev/null p.treefile
   cp /dev/null s.xreffile
   ln p.pafile    pafile
   ln p.infofile  infofile
   ln p.dt        deffile
   ln p.semfile   semfile
   ln p.sxfile    sxfile
   ln p.treefile  treefile
   ln s.xreffile  xreffile
   ln t.ctrl      ctrlfile
   /exec/semantic | set continue = false
   rm pafile infofile deffile semfile sxfile treefile xreffile ctrlfile
   if ($continue) then
      cp p.dt p.dt-an
      endif
   endif
```

```
# *********** execute expand ***********
if ($continue) then
  cp p.semfile t.semfile
  ln p.infofile  infofile
  ln p.dt        deffile
  ln t.ctrl      ctrlfile
  ln t.semfile   semping
  ln p.sxfile    sxfile
  ln p.semfile   sempong
  /exec/expand || set continue = false
  rm infofile deffile ctrlfile semping sxfile sempong
  if ($continue) then
    cp p.dt p.dt-dp
    cp p.semfile p.sm-dp
    endif
  endif


# *********** execute chainelm *********
if ($continue) then
  cp /dev/null s.elimout
  cp p.sxfile t.sxfile
  cp p.semfile t.semfile
  ln p.dt        deffile
  ln p.infofile  infofile
  ln t.ctrl      ctrlfile
  ln p.treefile  treefile
  ln s.elimout   elimout
  ln t.sxfile    sxin
  ln p.sxfile    sxout
  ln t.semfile   semin
  ln p.semfile   semout
  /exec/chainelim || set continue = false
  rm deffile infofile ctrlfile treefile elimout sxin sxout semin semout
  if ($continue) then
    cp p.dt p.dt-dp
    cp p.semfile p.sm-dp
    endif
  endif
```

```
# *********** execute order ***********
if ($continue) then
  cp p.dt-dp p.dt
  cp p.sm-dp p.semfile
  cp /dev/null p.vsfile
  cp /dev/null s.grout
  cp /dev/null s.deptrace
  ln p.infofile  infofile
  ln p.dt        deffile
  ln t.ctrl      ctrlfile
  ln t.semfile   semping
  ln p.sxfile    sxfile
  ln p.semfile   sempong
  ln p.vsfile    vsfile
  ln s.grout     grout
  ln s.deptrace  deptrace
  /exec/order || set continue = false
  rm infofile deffile ctrlfile semping sxfile sempong vsfile grout deptrace
  if ($continue) then
    cp p.vsfile p.vs-dp
    cp t.semfile p.sm-dp
    endif
  endif


# *********** execute optim ***********
if ($continue) then
  cp /dev/null s.optout
  cp p.dt-dp p.dt
  cp p.sm-dp p.semfile
  cp p.vs-dp p.vsfile
  ln p.infofile  infofile
  ln p.dt        deffile
  ln t.ctrl      ctrlfile
  ln t.semfile   semping
  ln p.sxfile    sxfile
  ln p.vsfile    vsfile
  ln s.optout    optout
  /exec/optim || set continue = false
  rm infofile deffile ctrlfile semping sxfile vsfile optout
  endif
```

```
# *********** execute vstrans ***********
if ($continue) then
   cp /dev/null p.vsctrl
   cp /dev/null p.vsfile
   cp /dev/null c.gentab
   ln p.infofile  infofile
   ln t.ctrl      ctrlfile
   ln p.vsctrl    vsctrl
   ln p.vsfile    vsfile
   ln p.semfile   sempong
   ln c.gentab    gentab
   /exec/vstrans || set continue = false
   rm infofile ctrlfile vsctrl vsfile sempong gentab
   endif


# *********** execute transsyn ***********
if ($continue) then
   cp /dev/null g.pgs.syntax
   cp /dev/null c.scantab
   ln p.infofile  infofile
   ln c.st        stabfile
   ln p.dt        deffile
   ln p.treefile  treefile
   ln t.ctrl      ctrlfile
   ln g.pgs.syntax pasout
   ln c.scantab   scantab
   /exec/transsyn || set continue = false
   rm infofile stabfile deffile treefile ctrlfile pasout scantab
   endif


# *********** execute transdef ***********
if ($continue) then
   cp /dev/null g.defout pasout
   ln p.infofile  infofile
   ln c.st        stabfile
   ln p.dt        deffile
   ln t.ctrl      ctrlfile
   ln g.defout    pasout
   /exec/transdef || set continue = false
   rm infofile stabfile deffile ctrlfile pasout
   endif
```

```
# *********** execute transact ***********
if ($continue) then
   ln p.infofile  infofile
   ln c.st        stabfile
   ln p.dt        deffile
   ln t.ctrl      ctrlfile
   ln p.vsctrl    vsctrl
   ln p.treefile  treefile
   ln p.semfile   semfile
   ln g.actout    pasout
   /exec/transact || set continue = false
   rm infofile stabfile deffile ctrlfile vsctrl treefile semfile pasout
   endif


# *********** supports routines ******
# *********** execute xref ***********
if ($continue) then
   cp /dev/null s.xref
   ln c.st        stabfile
   ln p.dt-an     deffile
   ln s.xreffile  xreffile
   ln t.ctrl      ctrlfile
   ln s.xref      outfile
   /exec/xref || set continue = false
   rm stabfile deffile xreffile ctrlfile outfile
   endif


# *********** execute findpath ***********
if ($continue) then
   cp /dev/null s.findout
   ln p.dt        deffile
   ln t.ctrl      ctrlfile
   ln p.sm-dp     semping
   ln p.sxfile    sxfile
   ln s.deptrace  deptrace
   ln s.findout   grout
   /exec/find || set continue = false
   rm deffile ctrlfile semping sxfile deptrace grout
   endif
```

```
# *********** execute protocol ***********
cp /dev/null s.prot
cp /sys/error.text error.text
ln t.ctrl     ctrlfile
ln s.out      infile
ln error.text msgtexts
ln c.st       stabfile
ln s.prot     prot
ln p.infofile msgfile
/exec/prot
rm ctrlfile infile msgtexts stabfile prot msgfile
```

APPENDIX B

GAG Derivation Graph

```
prot "gag listing" :
        USER prot.cmd
                : scan_stab
                : ala
                : "sys/error.text"
                : xact_i : xact_c


xref "cross-reference listing" :
        USER xref.cmd
                : seman_xref
                : seman_def
                : scan_stab
                : seman_c


pfind "attribute dependency analysis" :
        USER pfind.cmd
                : optim_sx
                : optim_def
                : order_dep
                : order_sem
                : optim_i : optim_c


xact <
  xact.pp "source code for visit sequences in propp format"
  xact_i "info file"*
  xact_c "ctrl file"*
  > "visit sequence translation" :
        USER xact.cmd
                : vsx_ctrl
                : optim_sem
                : xdef_def
                : chelim_tree
                : scan_stab
                : xdef_i : xdef_c
```

```
xdef <
  xdef.pp "source code for ALADIN definitions in propp format"
  xdef_def "definitions after definition translation"
  xdef_i "info file"*
  xdef_c "ctrl file"*
  > "translation of ALADIN definitions" :
       USER xdef.cmd
              : xsyn_def
              : scan_stab
              : xsyn_i : xsyn_c


xsyn <
  xsyn.grm "pgs grammar"
  xsyn_def "definitions after syntax extraction"
  xsyn_scan "scanner tables"
  xsyn_i "info file"*
  xsyn_c "ctrl file"*
  > "context free grammar extraction" :
       USER xsyn.cmd
              : optim_def
              : chelim_tree
              : scan_stab
              : vsx_i : vsx_c


vsx <
  vsx_tab "visit sequence tables"
  vsx_ctrl "visit sequence table interpreter"
  vsx_i "info file"*
  vsx_c "ctrl file"*
  > "visit sequence translation" :
       USER vsx.cmd
              : optim_vs
              : optim_sem
              : optim_i : optim_c
```

```
optim <
  optim_out "optimizer messages"
  optim_vs "optimized visit sequences"
  optim_sem "optimized semantics"
  optim_sx "optimized symbols"
  optim_def "optimized ALADIN definitions"
  optim_i "info file"*
  optim_c "ctrl file"*
  > "attribute optimization" :
       USER optim.cmd
              : order_vs
              : order_sem
              : order_sx
              : order_def
              : order_i : order_c


order <
  grout "graph output"
  order_dep "dependency trace"
  order_vs "visit sequence"
  order_sem "ordered semantics"
  order_sx "ordered symbols"
  order_def "ordered definitions"
  order_i "info file"*
  order_c "ctrl file"*
  > "attribute dependency analysis" :
       USER order.cmd
              : chelim_sem
              : chelim_sx
              : chelim_def
              : chelim_i : chelim_c
```

```
chelim <
  chelim_out "chain elimination messages"
  chelim_tree "tree after chain elimination"
  chelim_sem "semantics after chain elimination"
  chelim_sx "symbols after chain elimination"
  chelim_def "definitons after chain elimination"
  chelim_i "info file"*
  chelim_c "ctrl file"*
  > "simple chain elimination" :
       USER chelim.cmd
               : expand_sem
               : expand_sx
               : expand_def
               : seman_tree
               : expand_i : expand_c


expand <
  expand_sem "expanded semantics"
  expand_sx "expanded symbols"
  expand_def "expanded definitions"
  expand_i "info file"*
  expand_c "ctrl file"*
  > "ALADIN shorthand expansion" :
       USER expand.cmd
               : seman_sem
               : seman_sx
               : seman_def
               : seman_i : seman_c


seman <
  seman_xref "cross reference information"
  seman_tree "tree"
  seman_sem "semantics"
  seman_sx "symbols"
  seman_def "semantic definitions"
  seman_i "info file"*
  seman_c "ctrl file"*
  > "semantic analysis" :
       USER seman.cmd
               : makdef_def
               : makdef_pa
               : makdef_i : makdef_c
```

```
makdef <
  makdef_def "definitions"
  makdef_pa "parser actions"
  makdef_i "info file"*
  makdef_c "ctrl file"*
  > "make definitions" :
       USER makdef.cmd
              : parse_pa
              : parse_i : parse_c


parse <
  parse_pa "parser actions"
  parse_i "info file"*
  parse_c "ctrl file"*
  > "syntactic analysis" :
       USER parse.cmd
              : scan_gs
              : scan_i : scan_c


scan <
  scan_stab "symbol table"
  scan_gs "global symbols"
  scan_i "info file"*
  scan_c "ctrl file"*
  > "lexical analysis" :
       USER scan.cmd
              : ctrl_stab
              : ala
              : ctrl_i : ctrl_c


ctrl <
  ctrl_cmds "host system commands"
  ctrl_stab "initial symbol table"
  ctrl_i "info file"*
  ctrl_c "ctrl file"*
  > "control analysis" :
       USER ctrl.cmd
              : opt

opt ATOMIC "GAG options"

ala ATOMIC "ALADIN grammar"
```

APPENDIX C

GAG Optimized Derivation Graph

```
# The GAG System

       gag.run <
         gr_msgs "messages from gag generated analyzer run"
         gr_out "standard output from gag generated analyzer run"
         > "gag generated analyzer run" :
              USER gagrun.cmd
                     : ala.exe
                     : vsx_tab
                     : pgs_tab
                     : xsyn_scan
                     : PARAMETERS < (input) tty id >


       ala.exe "gag generated analyzer executable" :
              USER pas.cmd
                     : analyzer.p


       analyzer.p "gag generated analyzer" :
              USER anal.cmd
                     : xdef.pp
                     : xact.pp
                     : pgs.pp
                     : PARAMETERS < (lexer) >


       pgs <
         pgs.pp "parser"
         pgs_tab "parser tables"
         pgs_lst "pgs listing"
         pgs_bnf "bnf version of pgs grammar"
         > "pgs parser generator" :
              USER pgs.cmd
                     : mod.grm
```

```
mod.grm "modified parser grammar" :
        USER mod_grm.cmd
                : xsyn.grm
                : PARAMETERS < (grm) (sed) >


lexer "lexical analyzer" :
        USER lex.cmd
                : lex


xref "cross-reference listing" :
        USER xref.cmd
                : seman_xref
                : seman_def
                : scan_stab
                : PARAMETERS < xref >


pfind "attribute dependency analysis" :
        USER pfind.cmd
                : optim_sx
                : optim_def
                : order_dep
                : order_sem
                : scan_stab
                : PARAMETERS < deptrace >


xact.pp "source code for visit sequences in propp format" :
        USER xact.cmd
                : vsx_ctrl
                : optim_sem
                : xdef_def
                : chelim_tree
                : scan_stab
                : PARAMETERS < expand optimize codegen >


xdef <
  xdef.pp "source code for ALADIN definitions in propp format"
  xdef_def "definitions after definition translation"
  > "translation of ALADIN definitions" :
        USER xdef.cmd
                : xsyn_def
                : scan_stab
                : PARAMETERS < expand optimize codegen >
```

```
xsyn <
  xsyn.grm "pgs grammar"
  xsyn_def "definitions after syntax extraction"
  xsyn_scan "scanner tables"
  > "context free grammar extraction" :
        USER xsyn.cmd
                  : optim_def
                  : chelim_tree
                  : scan_stab
                  : PARAMETERS < expand optimize >


vsx <
  vsx_tab "visit sequence tables"
  vsx_ctrl "visit sequence table interpreter"
  > "visit sequence translation" :
        USER vsx.cmd
                  : optim_vs
                  : optim_sem
                  : scan_stab
                  : PARAMETERS < expand optimize >


optim <
  optim_out "optimizer messages"
  optim_vs "optimized visit sequences"
  optim_sem "optimized semantics"
  optim_sx "optimized symbols"
  optim_def "optimized ALADIN definitions"
  > "attribute optimization" :
        USER optim.cmd
                  : order_vs
                  : order_sem
                  : order_sx
                  : order_def
                  : scan_stab
                  : PARAMETERS < expand optimize >
```

```
order <
  grout "graph output"
  order_dep "dependency trace"
  order_vs "visit sequence"
  order_sem "ordered semantics"
  order_sx "ordered symbols"
  order_def "ordered definitions"
  > "attribute dependency analysis" :
       USER order.cmd
               : chelim_sem
               : chelim_sx
               : chelim_def
               : scan_stab
               : PARAMETERS < eliminate class tree visit graph deptrace
                              arrange >


chelim <
  chelim_out "chain elimination messages"
  chelim_tree "tree after chain elimination"
  chelim_sem "semantics after chain elimination"
  chelim_sx "symbols after chain elimination"
  chelim_def "definitons after chain elimination"
  > "simple chain elimination" :
       USER chelim.cmd
               : expand_sem
               : expand_sx
               : expand_def
               : seman_tree
               : scan_stab


expand <
  expand_sem "expanded semantics"
  expand_sx "expanded symbols"
  expand_def "expanded definitions"
  > "ALADIN shorthand expansion" :
       USER expand.cmd
               : seman_sem
               : seman_sx
               : seman_def
               : scan_stab
               : PARAMETERS < expand interface >
```

```
        seman <
          seman_xref "cross reference information"
          seman_tree "tree"
          seman_sem "semantics"
          seman_sx "symbols"
          seman_def "semantic definitions"
          > "semantic analysis" :
                USER seman.cmd
                        : makdef_def
                        : makdef_pa
                        : scan_stab

        makdef <
          makdef_def "definitions"
          makdef_pa "parser actions"
          > "make definitions" :
                USER makdef.cmd
                        : parse_pa
                        : scan_stab

        parse_pa "parser actions" :
                USER parse.cmd
                        : scan_gs
                        : scan_stab

        scan <
          scan_stab "symbol table"
          scan_gs "global symbols"
          > "lexical analysis" :
                USER scan.cmd
                        : ala

ala ATOMIC "ALADIN grammar"
```