A Mechanism for Environment Integration

Geoffrey Clemm and Leon Osterweil

CU-CS-323a-86 March 1986

An extended technical report

Department of Computer Science
Campus Box 430
University of Colorado,
Boulder, Colorado, 80309

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

1. Background

Software environment research is directed towards establishing effective ways of integrating software tools in support of software processes such as development and maintenance. It has been observed (eg. [Boehm 83]) that the development of a large software system usually costs between $50 to $400 per line, yet the quality of the end product is often disturbingly low. Further, it is generally agreed that the cost of maintenance of software systems over their lifetime usually far exceeds original development costs. In addition, the cost of software has been steadily increasing over the past decade. Most observers place the blame for the twin problems of high cost and low quality on the lack of an orderly, systematic methodology for developing software and the lack of effective software tools capable of exploiting computing power in the software development process. There has been no lack of work in the areas of development of software methodologies and tools during the past decade. Little of this work, however, has wound up being exploited widely in practice.

A common complaint about most current software methodologies is that they are not well supported by tools. Thus, especially because they tend to stress the importance of large and complex documentation of software lifecycle products, these methodologies are usually difficult and onerous to adhere to without significant tool support. In the area of tools there is a complementary complaint that the welter of existing tools is not well integrated and coordinated. Tools usually do not support any specific rational methodology, and further are rarely well integrated with each other.

As a result, software continues to be developed with largely manual and ad hoc procedures. Even where promising development methodologies are in place, it is rare for them to be adequately supported by tools. As a result it is difficult to experimentally and empirically determine how good these methodologies are, and whether they should be promulgated more widely. It is similarly difficult to definitively evaluate individual tools, as they are often not clearly related to well-defined software tasks within existing methodological frameworks. Thus, they are usually not directly comparable to existng manual procedures or more rudimentary tools. There has been a growing concensus that the emergence of effective software environments will catalyze improvments in software methodologies and tools alike by serving as the framework for the development of entire methodologies which are well supported by comprehensive toolsets. Such environments could then also serve as experimental and evaluative testbeds.

This paper describes a software environments research project which is aimed at gaining a better understanding of the architectural principles which must underly mechanisms for the effective integration of software tools in such a way

as to facilitate and enable the sort of experimentation and evaluation that has just been described. Our approach to this problem has been to create an innovative tool integration mechanism and then use it experimentally to integrate some diverse collections of software tools. The integration mechanism is called Odin. We believe that it can fairly and profitably be viewed as a language and interpretation system for what DeRemer and Kron have called "programming in the large" [DeK 75].

Odin was first used as the basis for the effective and successful integration of a family of Fortran development, testing and maintenance tools in the Toolpack project [Osterweil 83]. It was subsequently used to effectively integrate a family of diverse tools to support software development in c [KrnRitch 78], and a family of tools to support the creation of attribute grammars. The Odin architecture, our experiences in exploiting it, and the conclusions about the Odin architecture and principles to which these experiences have led us are the subjects of this paper.

2. The Odin Tool Integration Philosophy.

The Odin environment integration mechanism originally emerged in the context of the Toolpack project [Osterweil 83], which required an effective and systematic way in which to integrate a collection of tools for supporting the development, testing and maintenance of Fortran code. A major Toolpack project goal was to take a number of existing, and a variety of proposed, mathematical software development tool capabilities and effectively integrate them in a flexible, extensible way, thereby creating a Fortran software environment. At the time (1979) much had been written about software environments, but very few had been built, and there were virtually no successful environment architectural paradigms to serve as guides to our work. Thus, the establishment and experimental evaluation of an innovative environment integration paradigm became another major goal of the Toolpack project.

We adopted as a key architectural premise the suggestion which had been made by a number of authors ([Osterweil 81], [Riddle 83], [Buxton 80]), that an environment must be data centered, rather than tool centered. Although this seemed a bit surprising at first, it was certainly consistent with previous successful experience. A common characteristic of the most successful past applications of computer systems (eg. in banking, insurance and government) is that these applications have all been data centered. Computer people have encouraged the users of these systems to think of their work as being aimed at the creation and maintenance of large central repositories of information, and this approach has been quite successful. How paradoxical that we ourselves

have been slow to adopt this paradigm. Yet our own business seems most obviously to be information centered.

The Odin project elected to take this premise quite seriously, and to adopt the philosophy that the purpose of a software environment is to create and manage the repository of information needed to effectively build and maintain software. Thus we envisioned Odin-integrated environments to be collections of tools which are best thought of as satellites around a large structured respository of software data, and a command language which is best though of as a mechanism for using the tools implicitly to maintain that data repository.

Accordingly, one of the first design issues was to determine the information that should comprise the central data repository and the organizational structure that should be imposed upon that information. We concluded that a software data repository is best thought of as a store of software objects, in which the objects are those which are the inputs to, and outputs from, the various tools to be integrated. This suggests, for example, that such entities as program source text, derived views of the text (such as parse trees, symbol tables and flow graphs), and such non-traditional items as documentation, test data, test results, and program structure representations should be the sorts of objects which populate the data repository of a software development, testing and maintenance environment.

Accordingly we designed Odin to manage a repository of these relatively large-grained software objects by coordinating and managing the application of such corresponding large grained tools as parsers, instrumentors, prettyprinters and data flow analyzers. >From the Odin perspective, each software object is profitably viewed as an operand, and the tools which manipulate the objects should then be considered to be operators capable of aiding in creating and maintaining the validity and correctness of these objects. An important feature of this approach is that it deemphasizes the importance of tools, freeing users of the need to become expert in combining and interfacing tools. In fact, as shall be described subsequently, Odin encourages the synthesis of larger tool functions out of the functions of smaller tool fragments in ways which should be of no direct concern to end users. By supporting the composition of tools out of fragments Odin has more flexibility in efficiently managing its object store. This approach also makes it easier for users to configure and adjust individual tools so as to extract the maximum utility from them, but does not oblige users to become tool experts.

The foregoing characterization should suggest that Odin can also be thought as an interpreter for a high level command language whose operands are the various software objects in the data repository and whose operators are the various tool fragments used to construct the tools in the toolset . In fact, this view is

extremely useful. Later it shall be shown that these operands can be aggregated into stuctures, that the operands are best thought of as being typed, that the set of types is extensible, and that the tool operators enforce a strong typing discipline upon the user. Further, we shall indicate that the command language processor is profitably viewed as a compiler that is decomposed into the usual compilation subphases, but which emphasizes the importance of effective optimization through such strategies as lazy evaluation.

3. Related Work.

Before proceeding to a detailed description of the Odin architecture, it seems important to compare our approach to that taken in some other projects aimed at furnishing effective support for software development and maintenance activities. As we characterize our approach as one in which tools are integrated into an object centered environment, we shall divide this section into two parts--one discussing other environment efforts, and one discussing previous work on object management systems.

3.1. Other Software Environment Projects.

As stated earlier, few environments have been in existence for very long, although there has been a flurry of activity in this area within the past couple of years. In the following subsections we divide these past approaches into intelligent editing approaches, incremental compiler approaches, database approaches, and very early integrated tool system approaches.

3.1.1. Intelligent Editing Systems

Many of the earliest systems which were called environments were centered around tools which have come to be called syntax directed editors or intelligent editors. The the Cornell Program Synthesizer [Teitelbaum 81], Gandalf [Habermann 79] and Mentor [Donzeau-Gouge 84] are examples of such systems. An intelligent editing system is aimed at supplying to the code writer all of the capabilities of a line or screen editor, plus certain additional text generation and error detection capabilities. The idea of such a system is to implement a language sensitive editor which embodies an understanding of the rules of the language in which the user is writing his or her code and to use these rules effectively to speed the text input process and to help assure the correctness of the program being created.

Clearly such a system must incorporate at least a syntax analyzer to

incrementally parse input text as it is received from the user. As the input text is successfully formed by the parser into recognizable language constructs, the intelligent editor responds by either helping the user to proceed more quickly and effectively or by advising the user of errors that must be corrected. The intelligent editor may, for example, automatically complete the construct which the user is building (eg. if the user is in the process of inputting a long keyword), prompt the user for additional input required to correctly complete the construct (eg. if the user is in the process of keying in an if__then___else), or reject as incorrect new input which is inconsistent with the pattern of constructs already keyed in and accepted by the intelligent editor.

Thus it is seen that intelligent editors integrate smoothly and effectively at least two tools--namely an editor and a parser-- behind a comfortable and friendly user interface. In fact many of these systems integrate powerful and sophisticated viewing devices as well. The intelligent editor's user interface is often through a tool which makes intelligent choices about emphasizing the structure of the program being edited by suppressing detail according to strategic rules. This process has been referred to as holophrasting.

Some intelligent editors go even farther, attempting to integrate semantic analyzers as well. This is an extension of the philosophy that the tool should detect and prevent errors as soon as they can be detected. In this case, however, the errors in question are semantic errors. Thus these systems hold and incrementally update a semantic model of the program being evolved. This can be unwieldy. Further it has been observed that often a program can fall into a semantically inconsistent state while the user is in the process of changing more than one statement to correct an error. Under such circumstances the very power of the editor may prove counterproductive, making it hard or impossible for the user to edit in needed changes.

3.1.2. Incremental Compilation Systems

Once one has considered the integration of semantic analysis into an editing and parsing tool suite, the extension of the tool suite to include full compilation and interactive execution seems to be the next logical step. Systems such as Arcturus [Taylor 84] Interlisp [Teitelman 81] and Cedar [Teitelman 84] are examples of such systems, which have been referred to as incremental compilers. Such systems are able to facilitate the process of checking evolving programs for errors by having actual test data run through the programs. These systems convey the impression of support for the code creation process through a device that presents itself to the user as a very powerful editor. In reality these systems offer far more than just editing and compilation support, however. The ability to interact with the executing program makes these systems very

powerful debugging and checkout systems as well.

Another important feature of these systems is that their user interfaces provide an exceptionally smooth and uniform appearance to users. The basic philosophy of the user interface is that it should suggest to the user that the user is more or less continually producing code in the subject language--whether the user is using that language to actually create end-product code or to control the environment itself. Thus the Interlisp user controls the actions of Interlisp by creating Lisp expressions. The Arcturus user control Arcturus by creating Ada expressions. In both cases these utterances can themselves be altered and maintained by environment tools as well. The net effect is that the users of these environments are encouraged to feel that they are always programming in the target language--whether they are programming user systems or "progamming-in-the-large" to control their environment and its tool suite.

Unfortunately it is this very focus on the programming activity which seems to most sharply restrict the prospects of being able to extend the range of these environments to cover more of the software lifecycle. Because systems such as Interlisp and Arcturus are so sharply and effectively directed towards support of coding, we are pessimistic that they could be extended smoothly to support such other activities as comprehensive testing or maintenance.

We believe it is important for the user to have an interface to an editor which facilitates the code creation process, and these systems offer that most handsomely. We also believe, however, that it is equally important for the user to have an interface to a testing system which facilitates the testing process. We are doubtful that one single user interface is likely to be effective in supporting both types of interactions, however. We believe that users engaged in the various activities entailed by software development and maintenance are guided by particular mental models adapted to these tasks. An environment's tols and data items should be aimed at the effective support of the creation and maintenance of these models. Thus, it seems likely that users would be handicapped rather than helped by having to attempt to do testing while faced with a user interface which fosters the more effective creation of code. In an important sense it seems that it is the very tightness of the integration of tool capabilities that works most tellingly against the possibility of effectively integrating markedly different types of tools into an environment. Thus our primary concern about these powerful incremental compilation systems, and our primary reason for rejecting this approach ourselves, was that it did not seem to offer the prospect of being as smoothly extensible across the expected range of tools needed by Toolpack as we believed was necessary.

3.1.3. Early Tool Collections

Clearly we have betrayed our prejudice towards designing environments which are collections of decoupled tool capabilities, centered around a data repository. We believe this approach results in systems which are more flexible and extensible, enabling users to assemble and combine tools in ways which are likely to be better adapted to their needs and idiosyncrasies, and which are better focussed on the information centered and driven paradigm of software development.

One of the first system designs to recognze and capitalize on the importance of the decoupled tool approach was the Unix (TM) operating system [Kernighan 81]. Unix users are strongly urged in the direction of thinking of the operating system as a collection of small tools which are to be assembled into larger tool capabilities in whichever ways the user may see fit. Unfortunately Unix fosters a naive and inadequate view of the complexity of the data objects and structures which tools must create and pass back and forth between each other. Unix tools communicate with each other most easily through pipes, which are conduits for single files of text information. As long as users are able to comfortably model their needs and activities in terms of text files, and as long as users are able to comfortably model their tool needs as being text manipulation chores, Unix is likely to remain an adequate environment. Thus, for example, Unix seems to provide a reasonable environment for straightforward office chores.

The needs of software developers are far more demanding, however. Software development systems, as has been seen, ordinarily encompass such tools as parsers, semantic analyzers and testing systems. These tools create and consume such files as parse trees, semantic attribute tables and graph structures of various kinds. These files are not best thought of as being text files. Further a parsing, compilation or testing system requires multiple input files and should be expected to produce multiple output files. Thus, while the Unix model of readily assembled tool fragments is a good one, the Unix model of the tools as having trivial interfaces to each other severely hampers its application to the support of software development. In addition, the Unix focus on tools and relative deemphasis of the data objects which they manipulate seems to us to direct the user's attention in the wrong direction.

3.1.4. Database Systems

There have been some systems which have attempted to place what we would consider to be a suitable emphasis on the data centered nature of software development. One excellent example is the Troll/USE system [Wasserman 83] which is quite effective in supporting the creation of relational databases but

seems rather less effective in supporting the creation of algorithmic code. Troll/USE is surely an interesting approach, but it seems to us to invite the risk of suggesting to users that they might have to abandon algorithmic coding if they are to capitalize on tools effectively. We sought to create a system which effectively conveys the importance and effectiveness of the data-centered approach to software development in the context of coding with a classical algorithmic language.

A system which adopted just this sort of approach was the Joseph system [Riddle 83]. In Joseph, the focus was the creation and maintenance of a large relational database of information about the status of the software under development. This information ranged from management information to requirements information to information about design and coding. Joseph appeared, in early use, to be hampered by the fact that the information in its database was of too small a granularity. As the size of the database grew and the need for updates to the database continued and grew with the progress of the project, it became increasingly common for even minor updates to require frustratingly large amounts of time for database updates. Joseph seems to have been an early step very much in the right direction, but it did point out the importance of selecting an appropriate granularity for a data-centered environment.

3.2. Object Oriented Software Support Systems.

As previously indicated, the approach we elected to take was to construct Odin as a system for managing large-grained objects. Here we describe some past projects in which the management of large software objects was used as the basis for effective support of software activities.

3.2.1. Earliest Efforts.

The first software object managers were 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 the 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 object managment schemes was that they did not capture the evolutionary character of software objects. Each software object, whether it was stored in a box of cards, a reel of tape, or a file on a disk, only contained the view of the object at a single point 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, thereby allowing 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.

There have been a number of new systems providing 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 [Ritchie78].

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 relying upon a simple tree structure as the basis for storing and displaying software objects encouraged the development of database systems for managing large software objects. 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.

3.2.2. Automated Object Managers.

A result of the increasing complexity of software objects is that it is increasingly infeasible for a programmer to effectively manage software objects without assistance. Instead of a few boxes of cards and the associated boxes of object code, the programmer is now faced with software systems composed of complicated hierarchies and networks of objects, where each object in turn consists of a complex set of named and numbered versions. The logical solution is 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 was 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 which must interpret them 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 with command scripts, however, 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 generally remain valid for reuse during 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, thereby inviting the risk of introducing subtle errors through incorrect 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 such systems were developed to handle a specific class of intermediate objects. For example, the System Building System [DeJong73] was designed to manage only object code objects produced from PL/I 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.

3.2.2.1. Large Software Systems

The problem of efficient management of large software objects 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 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 results 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 is a serious impediment to effective use of the Intermetrics system.

3.2.3. General Object Mangers

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

The first successful general purpose software object manager was 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 the 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 Equipment Corporation's Module Management System [DEC84b].

A central characteristic of the Make system and its variants is the use of the host file system as a repository of information about current software objects. This provides many of both the strengths and weaknesses of these systems. The principal strengths are that such systems are extremely compact and efficient - they depend on the operating system (ie. the host file system) to maintain needed information. In addition, 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 pricipal weakness of 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 a desired object managment capability depends on having more information about the software objects than is provided by the operating system, then that capability cannot be provided.

One description of how additional information could be used to support a more general software 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].

One significant problem with all of the existing software 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 a 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), the problem becomes critical.

4. Overview of the Odin Architecture

As observed earlier Odin's central architectural notion is that an environment is best thought of as a system for managing the large grained software objects which are the embodiment of the information needed in the process of creating and maintaining a software product. That being the case, it seemed natural to us to center our architecture around the objects needed to develop software rather than the functional capabilities (eg. the tools) needed to build and maintain these objects. Thus, the Odin architecture is strongly object-centered. This impression is fostered and reinforced by the command language by which the user operates Odin (and the tools which it integrates). This language has as its philosophical basis and goal the creation of objects which the user specifies in response to perceived needs for information which those objects contain.

Requested objects are generally best created by the action of tools. Thus it may seem that an object-centered command language is isomorphic to an algorithmic or functional command language. As noted above, however, there is an important difference which becomes clear when one considers that it is often easy for the user to conceive of and describe an important object which may require lengthy and complex tool operations for its creation. For example, it may be easy for a user to conceive of and describe the output from a program test run but it may be quite hard to describe the intricate steps required in order to build the program source text out of a variety of files and libraries and then execute the program. The Odin philosophy is that users should be encouraged to think of these important objects, describe them tersely, and then have Odin-integrated tools build the requested objects as painlessly and noiselessly as possible. Further, as also noted above, a focus on specification of the objects required also opens the important possibility that the automatic object

management system can orchestrate and effect important intermediate object reuse leading to significant efficiencies.

Another key aspect of the Odin philosophy is that tool use should be encouraged as much as possible, and stumbling blocks to the use of tools should be removed as effectively as possible. Thus, Odin encourages the composition of tool capabilities. In particular the Odin command language makes it quite easy to apply one tool to the output of another. From the point of view of the user, this entails thinking of the object produced by a tool as a derivation of the object or objects which were taken as input by the tool. Thus the product of a tool is an object derived by the tool, and that object can be the basis for further derivations. We believe that this is quite consistent with the mental models which tool users exploit when managing the development of software. Effective software development requires the creation and use of a variety of views and versions of the program object under development. Some of these views and versions are easily obtained by the action of readily accessible tools (eg. a compiler, prettyprinter). Some are currently most readily derived by manual or mental processes. Some are very hard to derive at all because they involve exactly the sort of composition of powerful tools which has just been described. As a result, at present, some of these mental models do not get constructed at all and some are constructed incompletely or incorrectly. Whatever models the user is able to conjure up, they are stored and managed in a haphazard, manual, and error-prone way.

Our approach to this problem is to conceive of all such models, mental or otherwise, as objects, store them in a central repository, consider all of them to have been created from each other by the action of a tool or sequences of tools (except for a few "atomic" objects, which have been externally inserted into the information repository), and to save the user the burden of constructing them and maintaining their consistency with each other as the software development or maintenance process proceeds and necessitates changes. It is important to note that the objects we are discussing here may well be objects which are only evolved by laborious processes over long periods of time (eg. days, weeks or months). Such objects, and the many intermediate objects from which they are built are usually referred to as persistent objects. Thus the ability to manage such persistent objects is required in order to effectively support the view of software development and maintenance which we have just espoused.

Thus, we designed Odin to be a system for effectively managing large-grained persistent software objects and controlling this process through a powerful and conceptually suggestive command language.

In spite of the elegant and simple user view of tool application and object generation which the foregoing architecture presents, we nevertheless expected that

there would be considerable reticence to exploit it because of residual prejudices that tools are expensive to use. Indeed, tools, especially the powerful tools that were proposed for Toolpack, are expensive. Their purpose is to derive analytic information which may be abstracted from voluminous details. This is rarely inexpensive. On the other hand our experience has shown that often a variety of powerful, high level tools all rely upon identical or simiar bodies of lower level information. These bodies of lower level information should be viewed as objects which have been derived by low-level tools (eg. parsers and lexical analyzers). Thus the Odin strategy is to maintain such low-level objects as persistent objects once they have been created as part of the operation of a high level tool in the expectation that they will be of use as the basis for the operation of another high level tool. This strategy may be viewed as an optimization process where what is being optimized is the use of intermediate information across the extent of an entire software development activity. Accordingly Odin should be viewed as an optimizing persistent object creation and management system.

Another key philosophical component of the Odin architecture is that it supports and facilitates additions and alterations to the suite of tools and object types which it integrates. Although we feel that the Odin approach is capable of supporting a superior approach to making tool support available to end users, we also realize that it is just a small step toward the ultimate goal of providing appropriate automated support for software development and maintenance. In order to reach this ultimate goal it will be necessary to provide tool support on a scale far more massive than what we have achieved in our early work. Further it will be necessary to see that different users are able to tailor their tool support systems to meet their needs as their sophistication grows and as the natures of their various jobs change.

When viewed in this larger context, it becomes clear that the creator of an environment must be worried about producing something which may turn out to be provocative for a short time, but unacceptably constraining and underpowered afterwards. This suggests that current environments must be designed to be extensible. In fact, we realized very early in the Toolpack project that tools would have to be integrated incrementally, as some were initially available and others were to be provided at various intervals during the progress of the project. Thus a key requirement on Odin was that it support tool function extensibility. This requirement has been met by Odin, as we have already created integrated toolsets in which the original suite of tools was systematically and incrementally augmented without undue disruption or trauma to users. In some cases the augmentations were considerable in size and variety. In addition a number of the added tools were ones whose inclusion was not originally anticipated or planned for. Our experience to date indicates that the addition of new tools can be so effortless that experimentation with improved or

competing tools seems quite feasible. This suggests that our system might well prove to be useful as a basis for the sort of testing and experimentation with tools and methodologies which was advocated earlier in this paper.

5. The Design of the Odin Object Manager.

In designing and building the Odin prototype we rapidly came to understand that in order to effect the user view previously described it would be necessary to think of Odin as a system that manages not only software objects, but also the dependency and derivation relations among these various objects. In fact, by the end of the project it became clear that what was developed was not just an object structure, but also an ad hoc type structure for the software objects and tools managed by Odin.

The distinction is a necessary one if Odin is to be thought of as a system for integrating arbitrary collections of tools rather than a single, fixed, given collection of tools. In the latter case, the collection of tools can be taken as fixed and immutable, and the relations of the various tools and the types of objects they manipulated can be "hard-coded." We sought to study the principles underlying the effective integration of general toolsets, however. In addition, we recognized that at least for the present any collection of tools would have to grow and change incrementally in order to remain useful to its users. Both of these circumstances dictate a need for a system in which the structure of tool and object type interrelation and interdependency is to be separately maintained and maintainable.

Thus, this section will center on the description of two structures--1)the structure of the objects which are maintained by any single, specific Odin-integrated environment and 2)the structure of the types of these objects and the tools by which these objects are created and manipulated. The distinction between these two structures and the close working relation between them will be explained in the first two subsections. Following these two subsections will be subsections discussing the two languages for the description and manipulation of these two structures.

5.1. Software Object Management in Odin.

As observed above, the goal of the Odin architecture is to encourage the user to think of his or her work as much as possible as a process of creating and accessing software objects and to carry out that work by manipulating those objects as directly as possible. Our aim is to ask the user to specify as little

information as possible about how to produce these objects and to insulate him or her as much as possible from the work entailed. In some cases, it may be that the requested object or objects are persistent objects which have been requested before, and stored for future reuse. In this case, Odin will rapidly access them and provide them. In other cases it may be that the requested objects must be built essentially from scratch, perhaps because neither they, nor anything remotely like them have ever been requested before. In this case, Odin will determine the procedure by which they can be most effectively created, execute that procedure, present the requested objects and store them away for possible future reuse.

Our assumption is that, in most cases, the requested objects will be "similar" to objects which have been requested before. This may arise because objects formerly created have been altered, for example by editing, or because the requested objects are derivations of objects requested earlier. No matter what the reason, this is the most interesting case. In this case Odin will determine how similar the requested object or objects are to objects already created and stored. Odin will then take considerable pains to assure that the requested objects are created by making the most effective use possible of information already at hand.

In each of these three cases the Odin command interpreter's strategy is the same. The strategy involves carefully naming each object in its store in such a way that the name accurately reflects the way in which the object either has been, or could be, derived from the most elementary objects in the store by a sequence of tool fragments. This fully elaborated name then suffices as a guide which can be used to effectively and efficiently search the object store. If the requested object is already there the search terminates with a pointer to it. If it is not, the search terminates at an object or objects which represent(s) some progress from the store's most elementary objects towards what has been requested. Odin then uses the fully elaborated and derived name to provide a prescription of the sequence of tool fragments which must be invoked to take the already-existing object(s) and derive them into what has been requested.

An illustration should be helpful here. Suppose that the user is interested in viewing a prettyprinted version of a program named "joe", which had previously been prepared for a dynamic debugging session through the actions of a dynamic instrumentation tool. The user would request the object:

joe : ins : fmt

namely, the object which results from starting with joe, then producing an instrumented (ins) version of that, and then producing from that a formatted (fmt) version. It should be noted that although the user's view is that this

object has been created by the sequential execution of two tools--an instrumentor and then a formatter, the Odin strategy is to encourage the implementation of each of these two tools by the sequential execution of several smaller tool fragments. Thus in Toolpack/IST, the Toolpack toolset integrated under Odin, instrumentation is accomplished by the sequence: lexical analysis, parsing, semantic analysis, and then finally instrumentation. Formatting is accomplished by the sequence: lexical analysis, parsing and then formatting. Thus the object which the user requested is actually built by the sequential execution of at least half a dozen tool fragments, most of which are unseen by, and presumably unknown to, the user.

The process by which Odin translates the user's name into this tool fragment execution sequence entails the creation of the canonical internal name of the object which the user's request has described and then a search for an optimal sequence of tool executions capable of building the described object from the objects currently on hand in the store. The organization of the store is designed to effectively support searches for derived objects, and will be described shortly. If the search reveals the presence of the object which the user has requested in the store, that object is immediately returned to the user. If joe is present in the store, but none of joe's derivatives are present, then Odin constructs a procedure for building the requested object from joe. Along the way, each of the following objects (and a number of others) are also built:

joe : lex	(joe's token list)
joe : prs	(joe's parse tree)
joe : nag	(joe's semantic attribute table)
joe : ins	(instrumented version of joe)
joe : ins : lex	(token list for instrumented version of joe)

All are stored for possible reuse at some time in the future. If "joe : ins" is present in the store (eg. because the user had previously requested the execution of the instrumented version of "joe"), then Odin would use this object a the basis for a much simpler and faster derivation procedure, namely the execution only of the lexical analyzer and the formatter using what the user would call

joe : ins

as input.

The key to understanding how the above intelligent object derivation process works is an understanding of the organization of the underlying Odin software object structure, the derivative forest. This structure is detailed next.

5.1.1. The Odin Derivative Forest.

The derivative forest is a structure that indicates which objects in the Odin information store have been produced from which other objects in the store. Odin assumes that the object store consists of two different classes of objects--atomic objects and derived objects. Atomic objects are those which have entered the store either by means of a synthesis process such as text-editing or by explicit importation from an external store such as the host file system. Atomic object are object which Odin realizes it cannot reproduce on its own. Derived objects, on the other hand, are those which have been created as the direct result of the execution of tools or tool fragments incorporated by Odin. The derivative forest organizes all objects in the file store into a collection of trees, such that each atomic object is the root of a tree consisting of all the objects which have been derived, directly or indirectly, from the root object by tool invocation sequences.

Figure 1 is an illustration of a portion of such a derivation forest. In Figure 1, note that "sam", "joe", and "bob" are all atomic objects (in this case, each is a body of source code). The tree rooted by "sam" has two subtrees, each consisting of a single node. One node, labelled "prs", represents the parse tree derived from the "sam" source code by the Toolpack/IST parser tool. The other node, labelled "ins", represents the instrumented version of "sam" which has been derived using an instrumentation tool. The tree rooted by "bob" has one subtree whose root is labelled "fmt". This indicates that a formatted version of "bob" has been derived and stored. The node representing this formatted version is, in turn, the root of a subtree which contains two other nodes-- labelled "prs" and "ins". These two nodes represent further derivations, namely the parse tree and the instrumented version of the formatted version of the original "bob" source code.

5.1.1.1. Object parameterization.

Finally, note that the tree rooted at "joe" illustrates yet another descriptive and representational feature of Odin--namely the use of parameterized object descriptions. The subtree of "joe" rooted at "ins" has two sub-subtrees, each representing a different derivation of "joe" by the instrumentor. Often there is a variety of additional information that can be associated with an object which will affect the derivatives produced from that object. In Odin, 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.

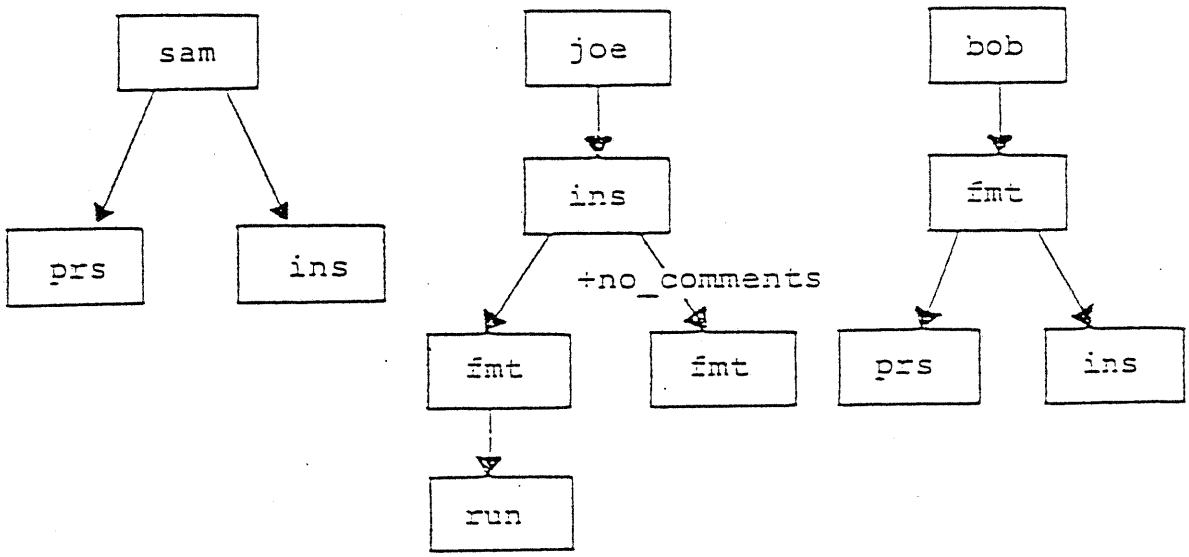

Figure 1

In the example, the user has specified explicit parameterization for one derivation of the instrumented version of "joe : ins". This instrumented version, denoted "no_comments" is presumably a version which the instrumenter will produce upon request which has no comments imbedded within it. The other instrumented version does not have an explicit parameter attached to it, but instead is parameterized by default parameterization.

A parameterized object is specified by augmenting the specification of the object by a '+' and the parameter. The paramenter specification is actually appended to the end of the specification of the object which is to be taken as input to the tool which is to interpret the parameter, and just before the specification of that tool. This is intended to underscore the fact that the parameterization is reasonably considered to be input to the tool, along with the contents of the object.

Thus, for example, the "no_comments" paramenter would be incorporated into the specification of object "joe : ins" as follows :

```
joe +no_comments : ins
```

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 "joe", then respectively

```
joe +debug=arrays
```

or

```
joe +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 "joe" using the library produced from an object called "util" would be specified as :

```
joe +lib=(util : lib) : run
```

It is important to note that the derivative forest keeps families of objects stored together in a way which is conceptually clean and logical, and also in a way that makes it easy for the Odin object manager to quickly and easily find objects which have been requested either directly or indirectly. Objects which have previously been created can be rapidly returned to the user, and instructions for the creation of objects which have not previously been created can be rapidly synthesized.

5.1.2. Alteration of Objects.

A significant complication to the above process of creating, controlling and reusing derived objects arises from considering the effect of altering objects, for example by a process such as editing. Under Odin, when an atomic object (for example source text) is modified (for example by a text editor), the resulting source text object is considered to be a new object which is a version of the original object. The user may specify a name for this new version, in which case it becomes a new atomic object and the root of a (temporarily descendantless) derivative tree. If the user does not specify a name for this new version, then it automatically replaces the object which was the original version. In this case, however, it is not safe to assume that objects which are derivations of the original version (descendants of the original object in its derivative tree) are correct derivatives of the object which is the new root of the derivative tree.

Odin assures that the potential problem will be recognized by assigning a date and time stamp to each object under its control. Whenever a derived object is found to be older than any of the objects which are ancestors in its derivative tree, that derived object is treated with suspicion. In particular, if the user requests "sam : ins" and Odin finds that "sam : ins" already exists in the file store, the requested object is not automatically returned to the user as stated above. Instead, Odin first compares the time and date stamps of "sam" and "sam : ins". If the time and date stamp on "sam" indicates that it is newer than "sam : ins" a rederivation process is begun.

It is tempting to suggest that this comparison process be avoided by the simple expedient of always deleting all derivatives of an atomic object which has been edited. Odin does not do this because some editing procedures result in superficial changes that do not alter some or all of the atomic object's derivations. Odin incorporates difference analyzers that attempt to make this determination in an attempt to aviod potentially costly rederivations which might not be necessary.

In the example just given, for instance, it may be that the user's editing of "sam" has only altered comment lines. In this case the previous scan table,

parse tree, symbol table and instrumentation are still correct derivations of the edited version of "sam". Odin will detect this and save most of the work of rederivation.

The process is as follows. First Odin recognizes that the present version of the "sam" object is new. As a result it invokes the scanner to rederive the scn_cmt and scn_tab derivatives. Having done this, it compares these new derivatives to the old derivatives. This comparison reveals that the new scn_cmt object is different, but the new scn_tab object is not. Thus Odin replaces the old scn_cmt object with the new version, but merely updates the time and date stamp on the old scn_tab object, indicating that it is a correct derivation of the new "sam" object. Further, Odin now recognizes that derivations of "sam : scn_tab", such as the prs_sym and prs_nod objects are also correct derivations of "sam" and it updates their time and date stamps as well without rederiving them. Finally Odin will recognize that "sam : ins" is also a correct derivation and will update its time and date stamp without rederiving it. Thus only the scanner tool fragment has been rerun in response to this superficial editing of "sam". Even this relatively minor rederivation, moreover, is carried out only in response to a user request for that derived object, or one of its descendants.

5.1.2.1. Trustworthiness and Validity of Objects.

The preceding discussion should have suggested that there are significant complexities involved in correctly and efficiently managing the effects of changes made to the objects of the object store. One of the most important mechanisms used to facilitate these activities is the simple expedient of attaching to each object an attribute intended to indicate how much confidence should be accorded the object. Associated with this information there is also a mechanisms for speeding the delivery of information about changes in the status of the object to all other objects which should need to know this information.

The basis for this mechanism is a status level attribute, attached to each object by Odin. The status level may be viewed as an enumerated type whose values are OK, WARNING, ERROR, NOREAD, NOFILE, and ABORT. OK is considered the highest status level and ABORT the lowest. The status of a atomic object is always OK. The status of a given derived object depends on the results of the tool fragments 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 fragment 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 are considered to be objects in the Odin object store. These objects are specified by appending the ": warn" and ": err" derivations respectively to specifications of the objects to which they apply. Thus, if the request for the object,

```
joe : run
```

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

```
joe : run : err
```

The error object is always a subset of the warnings object. 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.

One important way in which Odin makes use of status level attributes is by broadcasting the news that key objects have been changed to derived objects to which such changes are expected to be particularly significant. Such derivations are said to be related to such key higher level objects by a "sentinel" relation. The existence of a sentinel relation between such pairs of objects effects the automatic rederivation of derived objects and the automatic reporting of sufficiently low status level of any such rederivations. Sentinel relations are used to construct a network of constraints among the objects in Odin's store, as the store itself is being built up. The purpose of this network is to facilitate the early, effective and effortless detection of changes to a higher level object which cause errors in key derivations of those objects.

For example, suppose

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

are two Odin objects each of which is derived from the object "thesis.text". If these objects are related to "thesis.text" by sentinels, then whenever a modification to "thesis.text" causes the status level of a derived object to become ERROR or less, Odin will generate an error message indicating the

objects for which this has happened. In the above examples, 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 "thesis.txt" is modified, the presence of the sentinel relation between this atomic object and each of the two derived objects will cause Odin to automatically rederive those objects and automatically check that the "thesis.txt" object is spelled correctly, and that it is acceptable to the "prog.c" program.

5.1.3. Compound Objects.

The objects managed by Odin may be either compund objects or simple objects. The previous discussions of objects implicitly assumed that the objects under discussion were simple objects--that is that the objects had no internal fine structure that was visible or of interest to the object managment system. In fact some objects may have such a fine structure, in which case they are called compound objects.

Compound objects may arise in essentially two ways--either through the action of tools or through explicit construction by users. When compound objects are created by tools they are generally treated as monoliths by the Odin object manager. When they are created by users Odin generally has much more flexibility in handling them and is often able to achieve significant efficiencies.

5.1.3.1. Tool Generated Compound Objects.

The most common way in which tools effect the insertion of a compound object into Odin's object store is when a source text synthsizer such as an editor produces more than one compilation unit, or when an object input utility tool draws such an object in from the host system's file store. In either case, Odin is made aware of the internal structure of this object by a tool which has special knowledge of the original object and is capable of detecting fine structure in objects of this type (an example of such a tool would be a source text "splitter" which can identify separate compilation units in a source text object).

Odin employs other tools to enable users to extract components from such compound objects as well. To enable this, Odin associates a "key" with every Odin object. A atomic object is given a key based on the object's name. A derived object is given a default key equal to the key of the atomic object from which it was derived. For example, the key of the atomic object "joe" will, barring unusual circumstances, also be "joe." The key of "joe : run" would also be "joe".

In case a derived object is a compound object, the key for each element of the derived compound object is generated by a tool that derives names for the compound object. For example, suppose "joe : output" specifies the output object generated when executing the object "joe", and this derived object is a compound object because the nature of joe is that it generates more than one output object. The most useful and more likely situation is that the creator of the tool which derives ":output" objects has also supplied another special purpose tool which extracts from such compound objects especially meaningful names to be used as keys to those objects. For example, if the tool is one which produces formatted versions of input source text procedures, then the tool which creates key names would most likely be a tool which finds the names of all of the individual procedures.

Once a set of keys representing the components of a compound object has been determined, they can be used to extract these components. The objects within a compound object having a certain key can be specified by appending to the name of the compound object an at-sign ('@') and the key. For example, suppose that running "joe : output" produces three output objects whose key values are "DATA", "source.list", and "source.errors". These three objects could be specified as the three Odin objects,

joe : output @DATA
joe : output @source.list
joe : output @source.errors

5.1.3.2. User-Created Compound Objects.

The other principal way in which compound objects enter the Odin object store is through their explicit creation by users. The capability upon which this rests is the ability to to define objects which consist of pointers to other objects. Such objects have type "ref". For example, in Figure 2 the objects "sally" and "jane" are "ref" objects. The object "sally" consists of pointers to the three objects-- "sam", "joe" and "bob". The object called "jane" consists of pointers to "bob" and "tim".

Once compound objects have entered the Odin object store, other compound objects can be created from them by the action of tools. Thus, the application of an Odin tool to an input compound object generally results in the creation of an output compound object which can invariably be thought of as the composition of the various objects resulting from the application of the tool to each of the components of the original input compound object. For example if the user requests the object

sally : ins

Odin produces a compound object consisting of pointers to

sam:ins joe : ins bob : ins

In order to do this, of course, Odin first has to be sure that the "scn_tab", "prs_sym" and "prs_nod" objects are created for each of "sam", "joe" and "bob". All of these derived objects are stored as descendents of the "sam", "joe" and "bob" "f77" atomic objects. The object "sally : ins" is itself stored as an object which has been derived from the atomic object "sally". It consists of pointers to the derived objects

sam : ins joe : ins bob : ins

which reside in their respective derivative trees. Clearly if any or all of these individual "ins" objects had been created previously, Odin would recognize this and not go through the process of recreating them by the reapplication of tool fragments.

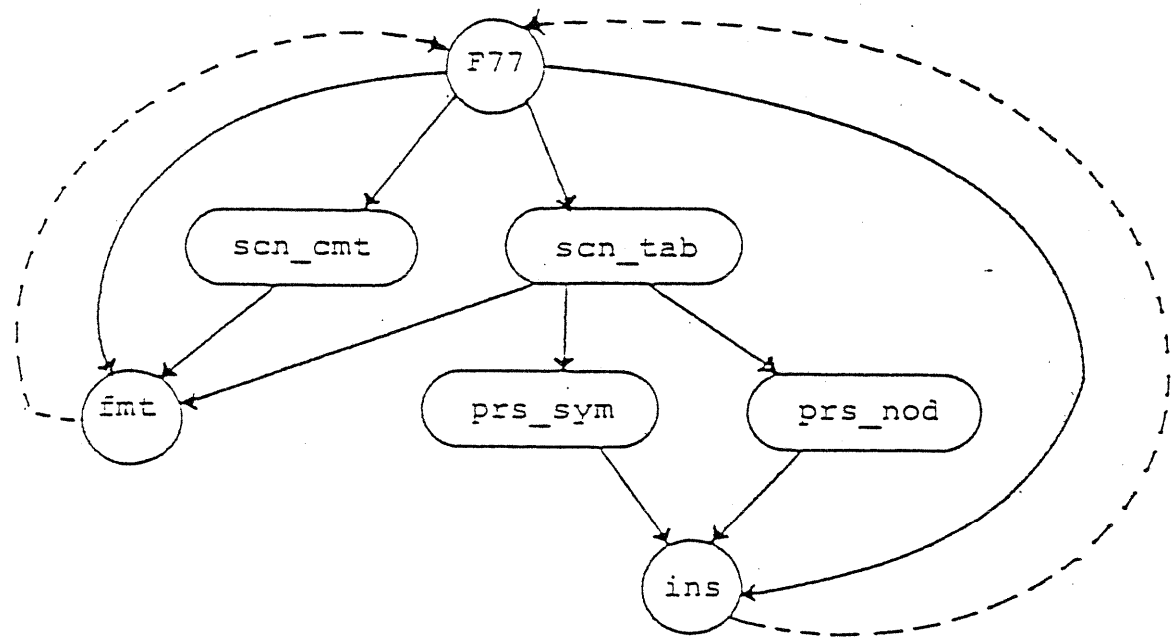
F77
scn_cmt
scn_tab
fmt
prs_sym
prs_nod
ins
——————> Derivation Edge
--------> Cast Edge

Figure 2

In addition, these derived objects, once created, are available for reuse in the context of responding to user requests made indirectly through other "ref" files. In the example shown in Figure 2, suppose the user requests the object "jane : ins" after requesting "sally : ins". As the request for "sally : ins" has resulted in the creation of "bob : ins", Odin needs only to create a pointer to that existing object and create the "tim : ins" object in order to satisfy the user's request for "jane : ins". Thus Odin supports the ability to aggregate objects in overlapping ways, without incurring inefficiencies due to needless repetition of work.

These "ref" objects can be used to define hierarchies of objects, by having a "ref" object contain other "ref" objects (as long as such definitions do not become recursive). This capability has proven quite useful, as support libraries have been stored as "ref" files, which have then, in turn, been included in higher level "ref" files which also incorporate pointers to clusters of source code objects comprising various functional pieces of a program. Pointers to these "ref" files are, in turn, included in still higher level "ref" files corresponding to, perhaps, major functional pieces of the program. Finally, the entire program is itself represented by one "ref" object consisting of pointers to high level "ref" objects representing major program constituents.

In constructing this hierarchy, the user need not be unduly concerned with assuring that the constituent objects are mutually disjoint, as no inefficiencies in tool application would result from this. Instead the hierarchical structure is free to reflect the program's logical structure. Such tool applications become very easy to request, as the user simply applies needed tools to the highest level "ref" object. In the case of a program which is under maintenance, and for which such tool applications have been done in the past, it is often the case that the many of the objects which the user is indirectly requesting have already been created. In this case these objects are not rederived. Only objects which have not previously been created, or which have been made obsolete by alterations to the atomic objects from which they had previously been created, need be created. In the usual maintenance scenario, the determination of which objects need to be recreated is a painstaking and perilous one. Most users elect to avoid it and its risks by doing massive rederivations, often needlessly duplicating considerable previous work. Through Odin the user is assured that only those objects which must be rederived will be rederived. The user simply makes one terse command. This relatively painless and extremely powerful mechanism for the control of derived objects has proven to be one of the most powerful and appealing features of Odin-integrated toolsets.

There is also no requirement that the objects to which the pointers in a "ref" object point be of the same type. Thus, we see that "ref" objects enable the creation of logical records and stuctures. This gives the Odin command language the ability to manipulate complex structures of objects of

heterogeneous types quite simply.

5.2. The Odin Dependency Graph.

The foregoing discussion of how users can employ Odin to manage complex tool and object configurations intelligently has now prepared us to discuss the way in which a different but related structure, namely the Dependency Graph, is used as the basis for this process.

The previous subsection has indicated that Odin is designed to help the user to use powerful and complex tools to create large and possibly intricate structures of software objects. Wherever possible users are shielded from the need to understand and master complex tool interactions. Further, users are also prevented from attempting to misuse tools, for example by applying a tool to an inappropriate argument object (eg. attempting to use a text editor on a flowgraph). The mechanisms for guiding proper application of tools and for forestalling inappropriate tool use is the same--a primitive typing mechanism.

The objects which the Odin user creates and organizes by means of the previously described Derivative Forest are best considered to be instances of data types, and tool fragments are considered to be operators which transform instances of the various types into instances of other types. The tool fragments (operators) integrated under Odin accept as inputs only operands of prespecified types. Odin is responsible for furnishing only objects of those types. This may involve casting inappropriately typed objects where necessary and possible. Where such casting is not possible, Odin is responsible for advising the user and attempting to provide help in forming an acceptable request.

The focus of the Odin typing and casting mechanism is the Odin Dependency Graph. This is the structure which Odin uses in order to retain a record of which tool fragments are currently incorporated into the system, which types of objects they produce and require, and the way in which these various tool fragments can be synthesized and concatenated to effect the higher level tool functionality which users are able to request. The nodes of the Dependency Graph represent the range of possible types of objects in the object store and the edges represent ways in which objects of one type can be transformed into new objects (perhaps of different types).

These transformations can be achieved in two different ways-- by casting and by derivation. Existing objects can be retyped by casting them from their present type to a new type. This is often useful when Odin determines that some existing object is the needed input to a particular tool fragment, but that the object is not of the type required by the tool fragment. Casting the object

from its current type to a new type does not alter the contents of the object, but retyping it enables different tool fragments to use it as input.

For example, in Toolpack/IST the scanner tool fragment accepts as input source text files. These files have type "f77" (Fortran 77 text). On the other hand, the output of the formatter tool (fmt) and the instrumentation tool (ins) are both source text, but are typed "fmt" and "ins" in order to facilitate the work of the Odin command interpreter when it is looking in the object store to see if these objects exist. Clearly objects of type "fmt" and "ins" should be considered legal inputs to the scanner. All that is needed is to convert their type (but not their contents) to type "f77". This is indicated by specifying that it is possible to cast objects of type "fmt" to type "f77" and objects of type "ins" to type "f77".

New objects can be created from existing objects by the action of tool fragments. In this case, the new objects are said to be derived from the existing objects. In the Odin Dependency Graph the nodes are the various types objects of which can be managed by Odin. The edges represent the various derivation and casting relations which can be effected either by, or in support of, the various tool fragments which Odin manages.

Figure 3 shows a part of the Dependency Graph which represents the tool fragments and object types managed by Odin to form Toolpack. In this figure cast relations are shown as dotted edges and derivation relations are represented by solid edges. This example is sufficient to support a more careful and satisfying explanation of how the Odin command interpreter is able to effectively reuse previously created persistent objects as was indicated earlier in this paper.

Earlier we indicated that the user could effect the creation of the formatted version of the instrumented version of a body of source text named joe by typing the command

joe : ins : fmt

This command would create a Derivative tree which is a supertree of the middle tree of Figure 1. Initially suppose that the middle tree of the Derivative Forest of Figure 1 consists only of the root, namely the atomic source text object, "joe", when the user gives the command. In this case, Odin first examines the Derivative Tree to see if file "joe : ins : fmt" has already been created. This search quickly terminates with a negative result as "joe" has no descendents at all. Odin next determines that "joe : ins : fmt" can not be built until "joe : ins" is built and concentrates on determining how to construct "joe : ins"

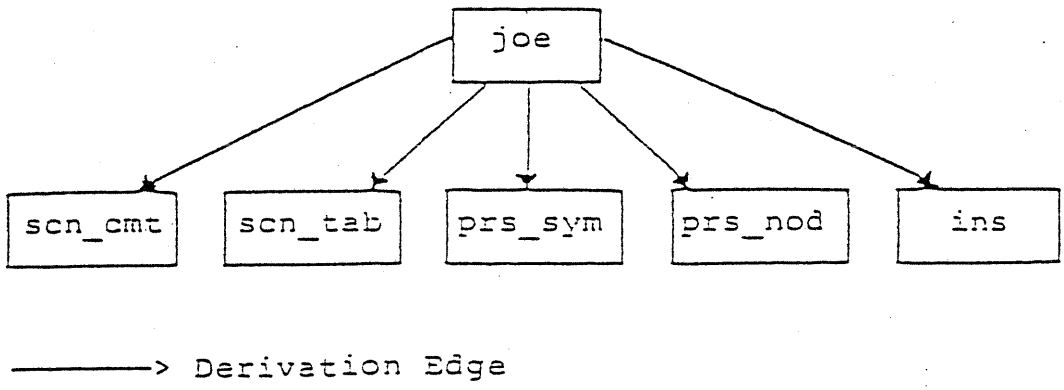

Figure 3

first. Odin consults the Dependency Graph to determine this. The Dependency Graph in Figure 3 shows that an object of type "ins" (such as "joe : ins") is derived from an object of type "f77", an object of type "prs_sym" (parser symbol table), and an object of type "prs_nod" (parse tree). An associated table (not shown here) indicates that the tool fragment which is able to produce both parser symbol tables and parse trees is the parser. Thus Odin infers that the creation of "joe : ins" requires the application of the parser.

Odin next determines the input objects required by the parse to see that they are present. In this case Odin discovers that the parser requires as input an object of type "scn_tab" (scanner table). This object is not present either (or else it would be shown as a descendent of the atomic object, "joe"). Thus "joe : scn_tab" must also be built. The associated table (not shown here) indicates that the scanner table is built by the lexical analysis (scanner) tool fragment using an "f77" source text object as input. Thus Odin infers that the scanner tool fragment must be invoked before the parser fragment. Odin searches to see if the input to the scanner is present. In this case, the Dependency Graph indicates that the input to the scanner must be an object of type "f77" and "joe" is an object of that type. Thus Odin now has determined that "joe : ins" can be built from the existing objects by first invoking the scanner using "joe" as input, and then invoking the parser, using "joe", "joe : prs_sym" and "joe : prs_nod" as inputs.

Now Odin is able to direct its attention to the task of creating "joe : ins : fmt". Odin now is able to assume that "joe : ins" has already been created, along with the various other objects necessary for the creation of "joe : ins". Specifically, when this process has terminated the Derivative Tree rooted at "joe" will appear as shown in Figure 4. The following objects will at that time all be direct descendants of "joe" --

joe : scn_cmt	(comments imbedded in joe source text)
joe : scn_tab	(joe's scanner table, sometimes referred to as the token list)
joe : prs_sym	(joe's symbol table)
joe : prs_nod	(joe's parse tree)
joe : ins	(instrumented version of joe)

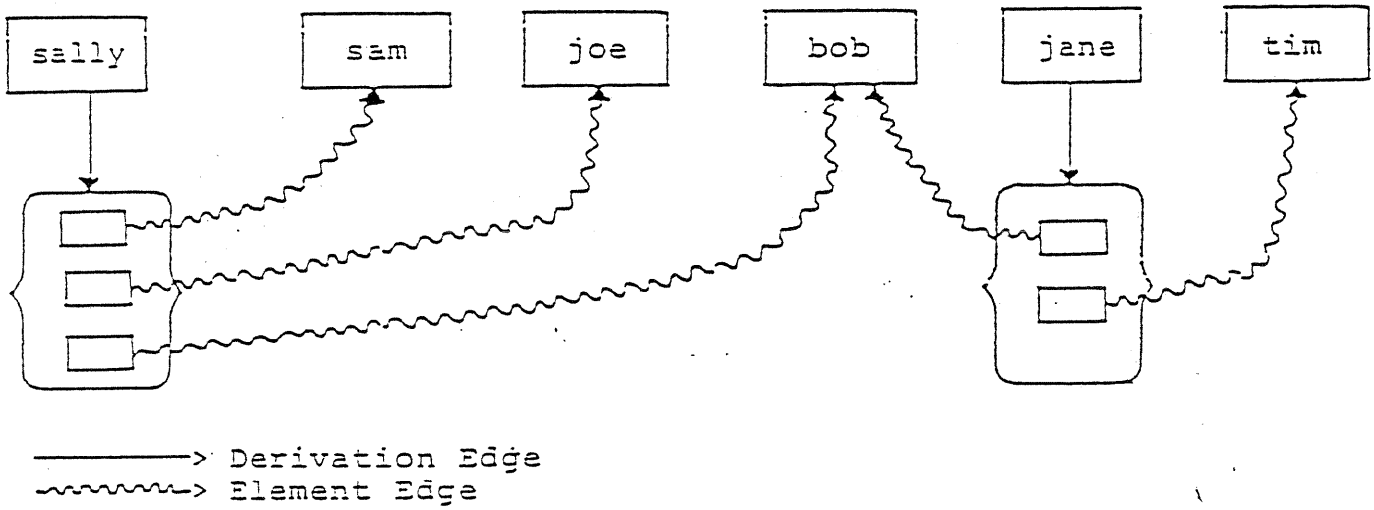

Figure 4

The desired object, "joe : ins : fmt", must be created as a node of a subtree rooted at "joe : ins", but it is unclear whether or not Odin can create the desired object immediately. Odin now examines the Dependency Graph and notes that an object of type "fmt" can be derived from an object of type "f77" and an object of type "scn_tab" by using the formatting tool fragment. As "joe : ins" is not of either type, some work must be done. In particular the object of type "scn_tab" can be created by the scanner from an object of type "f77", indicating that Odin will have to invoke the scanner on some source text. Unfortunately, "joe : ins" is not of type "f77". Here, however, Odin determines that "joe : ins" is of type "ins" and that objects of type "ins" can be cast to objects of type "f77". Thus Odin determines that the scanner can in fact be applied directly to "joe : ins", to create "joe : ins : scn_tab". That object, along with "joe : ins" (cast to type "f77"), are sufficient inputs to enable the formatter to execute, thereby producing the final object "joe : ins : fmt".

It should now be clear that this whole process would be significantly expedited if "joe : ins" were already in the object store at the time the user requested "joe : ins : fmt". In this case, Odin's initial search for "joe : ins" would have been successful, as "joe : ins" would be a direct descendant of "joe". Odin would then have determined that only the scanner and formatter would have had to be invoked in order to build the requested object.

5.3. Extensibility.

This is an opportune time to indicate the manner in which Odin serves as an excellent vehicle for facilitating the flexibility and extensibility of toolsets which it integrates. The basis for the extensibility of Odin is the Dependency Graph just described. This graph indicates the way in which objects of any type can be built from other objects, perhaps of a variety of types. It is important to note that this Dependency Graph is accessed and maintained by the Odin command interpreter and is not accessible to the various tool fragments themselves. Further, the various tool support libraries through which the tool fragments access objects force the tool fragments to access only needed objects and isolate the tool fragments from any direct contact with other tool fragments. Thus tool fragments have no knowledge of the sequence in which they may be called, and are prevented from establishing reliance, explicit or implicit, upon other tool fragments. As a result any tool fragment can always be replaced by another provided that the replacement produces objects of the same types as that produced by the original, and draws upon objects created by the other tool fragments in the toolset. This effects a great deal of flexibility in upgrading or correcting existing tool fragments.

Beyond that, this architectural device also makes it relatively straightforward

to integrate a new tool fragment into an existing toolset. The new tool fragment must first be characterized in terms of the object types it requires as input and produces as output. Its input object types must all be types already produced by existing tool fragments. Its output types may be either partially or totally new. If some output object types are new, then new nodes must be inserted into the Dependency Graph to represent them. Edges connecting the input types to the output types--new and old--must then be inserted into the Dependency Graph as well. Once this has been done Odin has complete knowledge of when to invoke the new tool fragment, and how to optimize the creation of objects which it produces.

In Odin the Dependency Graph is specified through the use of a Specification Language and is instantiated by means of a processor for translating such specifications into the actual graphs. The language is described in some detail in a later section of this paper. Specifications in this language are relatively straightforward to alter by the use of a text editor. Thus the process of altering or extendng an Odin-integrated toolset centers on making alterations to the Dependency Graph and supplying the new tool fragment(s) in an appropriate fashion.

Alterations to the Dependency Graph can currently be effected by using a text editor to modify the graph specification and then rerunning the tool used to construct the Dependency Graph from its specification. Supplying the new tool fragment is easily done by placing an executable version of it someplace where the Odin command interpreter can effect its execution.

Supplying the tool fragment in an appropriate fashion should entail more than this, however. As observed earlier Odin's object store contains typed objects, where the type structure is specified by the Dependency Graph. In order for this typing structure to most effectively support the flexibility needed in a prototype integrated toolset, the implementations of the various types should be concealed from the tools which use them. Thus, it is most preferable for types to be defined as abstract data types (ADT's) in terms of clusters of accessing functions. When types are defined in this way, their implementation structures are more effectively hidden from using tools, and can therefore be modified transparently to those tools. Thus, for example, the internal representation of a data type might be altered to make accesses more efficient, to reduce storage costs, or to enable the incorporation of tool fragments which create instances of the type more efficiently. All of these sorts of changes to the toolset are made possible if the type is defined in terms of accessing cluster functions. Thus it is most preferable that new tool fragments which create new data types be supplied not simply as a single executable capable of generating instances of that type, but rather in the form of a collection of executables.

The collection should include a cluster of accessing functions representing the primitive accessing capabilities for the new type, as well as the executable which creates instances of the type. The executable should be designed so that it creates instances of the new type by invoking the appropriate type accessing primitives, and so that any uses which it makes of objects of the new type are made through appropriate primitives as well. The cluster of accessing primitives, of course, must be made available to writers of future tools which intend to build upon objects of the new type. Ideally Odin should take a positive role in enforcing the use of accessing primitives by all tools and tool fragments. Presently this enforcement is carried out informally in the Odin-integrated toolsets that have been constructed to date.

Our experience to date indicates that Odin is indeed a vehicle for readily extending and modifying integrated toolsets. We have succeeded in incrmentally incorporating dozens of tools and tool fragments into Toolpack/IST--some which we produced ourselves and some which we captured from host environments. New tool fragments have been incorporated into Toolpack/IST in as little as five minutes. In Toolpack we have attempted to treat object types as data abstractions, but this has been done on a largely voluntary basis, due in large measure to the fact that many important Toolpack tools were captured from host environments and could not be altered to sharpen the boundaries between ADT accessing functions and functional tool capabilities.

6. The Odin Request Language

As observed earlier, the goal of a software environment should be to serve as a positive, painless aid to the user in effectively gaining access to software objects which contain needed information. To that end, the role of the language which the user is to employ should be to effect that access quickly and painlessly. Accordingly, the language which is provided as the user's vehicle for creating and accessing software objects--called the Request Language--is an imperative object oriented language. This section presents a very brief overview of the Request Language. Further details can be found in [Clemm 86] and [ClemOst 86].

The Request Language offers the user the use of two different Basic Commands--Display and Transfer--and a variety of Utility Commands.

6.1. The Display Command.

The display command prints out an Odin object to the current standard output device, normally a terminal screen. This command is implicitly invoked, being

automatically invoked whenever an Odin object is specified. The key to understanding how objects are specified in the Odin command language is to recall that all objects are considered to be the outputs from tools and tool fragments. Thus each object is specified by naming the root object in the derivative tree in which it is contained and by appending to that name the names of key nodes in the derivative tree which are needed to unambiguously specify the requested object. The names of these appended nodes are separated from each other and from the name of the derivative tree by a punctuator, currently the colon (:).

Thus, if "joe" is the name of a derivative tree corresponding to a source text object, then if the user specifies

joe

Odin will return the source text comprising the object named joe.

If the user specifies

joe : fmt

Odin will return a version of the atomic object "joe" which has been processed by the tool fragment named "fmt" (in Toolpack/IST this is a formatting tool).

If the user specifies

joe : ins

Odin will return a version of the atomic subroutine joe which has been processed by "ins" (in Toolpack/IST this is the dynamic instrumentation tool fragment). Further the user can specify

joe : ins : fmt

in which case Odin will return an object which has been derived from the atomic version of "joe" by first instrumenting it for dynamic analysis and then formatting that derived version. The user is free to specify arbitrarily many successive derivations by Odin-integrated tools and tool fragments in this way.

Clearly these derivations could not have been created until and unless an interpreter system for the Request Language had previously effected the construction of lexical analyses and parses of various versions of "joe". As noted earlier, the user does not need to know any details of what was done or why or when these views were created. On the other hand if the user wishes to examine these views for any reason, it is easy to do so. The naming conventions to be used

are just those described above. For example, if the user wishes to see the token list prouced by lexical analysis of the atomic version of "joe" he or she need only specify it as

joe : scn_tab

As a result of this command, the interactive user would see the token list scroll by on the terminal screen.

In these examples the objects which the user is able to specify are the straightforward outputs of relatively simple tool capabilities. Much more complex derivation processes can be concealed behind the Odin command language conventions and formats. For example, if joe were an executable main program then the user could specify

joe : rin

and Odin would effect the execution of "joe", displaying back to the user the output of the run (which in this case is assumed to be interactive). Thus this specification effects the compilation, loading and actual execution of the program represented by the source text of the atomic version of "joe". If the user instead desired to see the results of executing the version of "joe" which have been instrumented, then the user would only have to type

joe : irn

In this case, Odin would effect not only the instrumentation of the atomic version of "joe", but also its compilation, its loading and its execution. All details of how this had been done would be concealed from the user.

Earlier it was also noted that complicated tools such as an instrumentor or a formatter can be configured to perform differently by the use of parameterized object specifications. >From the point of view of the Request Language, these different versions are denoted by specifying the names of the parameter specifications or specification objects just after the names of the objects which are input to the configurable tool. Option specifications are separated from object names by a special punctuator, currently the plus sign (+). Thus, assuming that the user has previously created an object called "newtopts" containing a specific set of instrumentor options, the version of "joe" obtained by instrumenting it according to the dictates of "newtopts" is specified by

joe +newtopts : ins

Finally it should be noted that in all of the above discussion it was not

necessary to have assumed that "joe" was the name of the derivative tree of a source text object. It could equally well have been the name of a compound object, such as a "ref" object. Thus, supposing that "sally" had been defined to be a "ref" object consisting of pointers to the objects named "sam", "joe" and "bob", then

sally : fmt

would be the specification of the object consisting of pointers to the objects which are the formatted derivations of "sam", "joe" and "bob".

sally +newtopts : ins

would be the specification of the object consisting of pointers to the objects which are the derivations of "sam", "joe" and "bob" obtained by using the instrumentor according to the dictates of the "newtopts" parameter specification object.

sally

is the specification of the "ref" object and would *not* return the concatenation of the source text of "sam", "joe" and "bob", but, rather the ref file itself:

sam
joe
bob

In order to obtain the concatenated source texts for these files, the user would ask for the object created by the action of "cmpd", a special purpose Odin unary operator defined on "ref" objects. Thus the user would specify

sally : cmpd

to create the concatenated source text of atomic versions of the source text objects, "joe", "sam", and "bob".

6.2. The Transfer Command.

The basic form of the transfer command copies the contents of one Odin object into another Odin object. The second object must be a atomic object. 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,

joe > tom

would put a copy of the contents of "joe" into "tom".

joe : run : err > joe.err

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

An extended form of the transfer command places an object as input to a host system command. This allows the use of host system "editors" or "viewers". In this form of the transfer command, appended to the object is a right-angle-bracket ('>'), a colon (':'), and the name of the host system command. For example,

joe > : vi

would invoke the host system editor "vi" on the file "joe", while

joe : run : err > : more

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

In case the colon and host system command name is omitted, a default host system command is invoked. The name of the default host system command can be specified by the user through mechanisms described later in this section. For example, if the default host system command is "vi", then the following two commands are equivalent :

joe >
joe > : vi

6.3. Utility Commands.

6.3.1. Command Script Commands.

An Odin command script is 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" contains a list of Odin commands,

< script.odin

would invoke all the commands in script.odin.

6.3.2. History Commands.

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 display the history list.

6.3.3. Help Commands.

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.

6.3.3.1. Syntax Help.

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 ('?'). At that point the user is able to get further information about any of the listed topics by typing in the name of the topic, followed by a question-mark.

6.3.3.2. Atomic Type Help

A list of all object types currently being managed by Odin's object store can be automatically extracted from the Odin Dependency Graph and presented to the user, simply by typing "?:". This feature of Odin's help system is automatically maintained and kept consistent with the actual system specification.

6.3.3.3. Derived Type Help.

If the type of a desired derivation has been forgotten or a list of the types which might possibly be derived from a particular specified object, a question mark ('?') can be put in place of the derivation name, and the Odin System will

respond with a list of the types that could possible be derived. Thus, for example

```
joe : 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 :

```
joe : fmt : obj
joe : fmt : fmt
joe : fmt : xref
joe : fmt : run
```

6.3.3.4. Parameter Type Help.

Similarly, a question mark can be used in place of a parameter, in which case Odin will respond with a list of the possible parameters that could appear at that position. For example,

```
joe : fmt + ?
```

would generate the following message :

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

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

```
joe : fmt +id
joe : fmt +lib
joe : fmt +debug
```

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

```
joe : fmt +id=run5
joe : 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,

joe : fmt + ? : obj

would generate the following message :

Possible Parameters : debug

This states that the following would be a legal object:

joe : fmt +debug : obj

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

6.3.4. Odin System Parameter Commands.

The functioning of the Odin system is regulated in some very important ways by the settings of a variety of system parameters. To facilitate user understanding and control of this functioning Odin makes these system parameters accessible to users. Some parameters are read-only and some are user-modifiable. In this subsection we briefly indicate some of these parameters. The interested reader should consult [Clemm 86] or [ClemOst 86] for further details.

6.3.4.1. The ErrFile System Parameter.

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 parameter.

6.3.4.1.1. The Editor System Parameter.

The Editor parameter specifies the name of the default host system tool to be used for the abbreviated form of the transfer command.

6.3.4.1.2. The HelpLevel System Parameter.

The HelpLevel parameter specifies the degree of detail provided when the user asks for help.

6.3.4.1.3. The History System Parameter.

The History parameter specifies how much history information the user is to receive.

6.3.4.1.4. The LogFile and LogLevel System Parameters.

The "log" contains a brief description of each of the tools that were invoked to satisfy the request for an object.
These two parameters are used to control the way in which that log is maintained.

6.3.4.1.5. The Size, MaxSize and MinSize System Parameters.

These parameters are used to regulate the size of the cache in which Odin stores computed objects.
Odin deletes and recreates objects as needed to keep the cache filled.

6.3.4.1.6. The Sentinel System Parameter.

This parameter specifies whether or not sentinels are to be actively used to help in automatically propagating alterations to objects which Odin maintains.

7. The Odin Specification Language.

As discussed earlier in this paper, the structure of the object types and tools which are integrated by Odin is expressed in the Dependency Graph, which is defined by a specification expressed in a language which we refer to as the Specification Language. The specification language is designed to allow the integration of existing tools or under 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 could 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 (compound) 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 must be expressed in text form and incorporated into the text object which is the specification of the Dependency Graph. As we shall see, however, this body of text describes more than just the Dependency Graph. Basically, a tool 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 type of the object produced by a C code formatter, the string in quotes on the first line constitutes an official description of this object type, the name following the keyword USER on the second line is the name of the tool which is currently being defined to be a producer of objects of this type, and c names the type of object which is suitable as input to the tool being defined.

In general, the input/output behavior of a tool can be far more complex than this simple example, but this basic model of naming the output of a tool, naming the procedure that invokes the tool, and then describing the input to the tool, is always followed. In this section we very briefly summarize the salient aspects of this specification language. More detail can be found in [Clemm 86]

and [ClemOst 86].

7.1. Comments

Comments can be placed anywhere within the text of a specification expressed with the specification language. A comment is initiated with the sharp character ('#') and is terminated by the end-of-line character.

7.2. Atomic Object Types

Every object type being included in the specification must be assigned a unique "atomic object type". Each atomic object type is declared in the specification by stating the name of the atomic object type followed by the keyword ATOMIC and a text string stating the official 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"
```

7.3. Derived Object Types.

Every object type that is produced by a tool described in the specification must be given a unique "derived object type" and defined by a specification. A definition 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 that tool.

7.3.1. Derived Object Structures.

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. Some examples of the different kinds of outputs that a tool might generate are: 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. These are specified as follows:

7.3.1.1. Simple Derived Object Types.

A simple derived object type specification consists of the name of the derived

simple 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.

7.3.1.2. Reference Derived Object Types.

A object with a "reference" derived object type is an object that refers to another object. There are two kinds of reference derived object types - pointer reference and name reference.

An object with a "pointer reference" derived object type contains the actual name of the object being referred to. The specification of such an object is like a simple derived object type specification except that a caret ('^') is inserted before the 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".

An object with a "name reference" derived object type contains an Odin query specification of an object. A name reference derived object type specification is like a pointer reference derived object type specification except that an at-sign ('@') is placed immediately following the name of the object type which is being referred to. For example, in:

f_main ^ fcast@ "scanner default main program" :

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

7.3.1.3. Compound Derived Object.

A "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. There are two kinds of compound derived object types--compound reference type and compound source

type.

7.3.1.3.1. Compound Reference Derived Object.

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".

7.3.1.3.2. Compound Source Derived Object.

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".

7.3.1.4. 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.

7.3.2. Tool Input Specification.

In order to produce objects as specified, one or more input objects are needed by the tool that creates such objects. 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.

7.3.3. Specification of Tools.

The purpose of a tool specification is to furnish the name of the process which 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.

7.3.3.1. Internal Tools.

Odin furnishes a repertoire of internal tools to facilitate the more effective assimilation of user tool capabilities, and to assure that the more sensitive and complex manipulations of Odin's key data structures are shielded from users. An internal tool can be specified for use in a derived object specification simply by stating the keyword for that internal tool.

Following are some examples of commonly used internal tools:

7.3.3.1.1. STRUCT.

A tool that produces a composite object from a text object containing a sequence of Odin object specifications, one per line.

7.3.3.1.2. COMPOUND.

A tool that produces a compound pointer reference object from a compound name reference object.

7.3.3.1.3. COLLECT.

A tool that produces a single compound reference object from a set of input objects by constructing a compound reference object whose elements are the input objects.

7.3.3.1.4. FLATTEN.

A tool that produces a flat compound object from a nested compound object.

7.3.3.1.5. HOMOMORPHISM.

A tool that produces a compound object from another compound object by applying the derivation following the HOMOMORPHISM keyword to each element of the input compound object.

7.3.3.1.6. KEY.

A tool that generates an object containing the key of the input object. This is the key that would be used by the Odin selection operator.

7.3.3.1.7. CAT

A tool that produces a simple object from a compound object by concatenating together the contents of all simple objects that are elements of the compound object.

7.3.3.2. External Tools.

An external tool is defined in a derived object specification with the use of the keyword USER followed by the name of the external tool. For example, the external tool "cc" is defined by the following specification.

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

7.4. Linking Object Types.

For a variety of reasons it is often important to be able to relate one or more objects types defined in an Odin specification to be related to each other in certain specific ways. Odin provide some mechanisms for doing this. We briefly indicate why this is useful and how it can be done. Additional details can be found in [Clemm 86] and [ClemOst 86].

7.4.1. Joining Tool Outputs.

Sometimes 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. This relations is established by defining a linking object type.

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, eg. "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"
```

Note that the keyword DERIVED is used to specify that obj is a linking object type. 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.

7.4.2. Equivalence Links.

It is sometimes important to indicate that two types are interchangeable and can be used in identical ways. This is done by defining an equivalence link. A equivalence link is created by specifying the "to" object type followed by an arrow ('<=') followed by the "from" object type.

7.4.3. Cast Links.

Sometimes an object type that is derived from a given input object type can be used in the same way that the input object type could be used. For example, the output of a formatter can be used in all the ways that the original object could be used--it can even be formatted again. This can be expressed as part of the specification of the derivation graph by specifying a cast link from the derived object type to the given input object type.

8. An Odin Implementation.

An implementation of the Odin system has been in use at the University of Colorado at Boulder since 1983. In this section we briefly summarize the way in which this implementation which was built atop the Berkeley Unix operating system. This section is very much abbreviated from the treatment of this

subject which can be found in [Clemm 86] and from the summary which can be found in [ClemOst 86].

In the University of Colorado implementation of Odin, the objects manipulated by the Odin system are files in a Unix file system. The implementation provides concurrent multi-user access in either batch or interactive modes of operation. Atomic objects are arbitrary files that were created either through a special Odin internal accessing function, called Manipulate, 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. Only the Odin System has write access to this directory.

Derivation and manipulation of Odin objects are achieved by executing accessing functions which are implemented through the creation of a command script that is given to the Unix operating system call, "system()", for execution. A skeleton for the command script for each tool is created when the tool is specified and 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 file with macro names replaced with actual file names. This command file is then given to the Unix system for execution.

Typing of atomic objects is indicated by an 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.

The Key of an atomic object is the last segment of the host system file name for that object. It consists of the string preceding the extension, without the trailing period.

8.1. The Organization of the Object Store.

All information concerning the objects in the Odin system resides in a data store that is implemented as a single Unix file called the INFO file. This data store is structured as a network of nodes.

8.1.1. Object Nodes.

In the Odin data store there is one "Object Node" for each object known by the

Odin system. An object is either an atomic object or a derived object. 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.

8.1.1.1. 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

8.1.1.1.1. Host Names for Derived Objects.

For atomic objects, the object host name corresponds to the object node name. For derived objects, though, a host name must be created since the object is

created by the Odin system rather than by the user.

Earlier implementations of the Odin system attempted to derive a host system name for a derived object in a way analogous to that for atomic objects. For example, the derived object named

root-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 mapping the names for derived objects. However obscure the 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.

The solution to the collision problem used in the current implementation is for the user to specify a special directory in which all derived objects should be placed. This solution 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 used to locate the object in a short fat directory tree.

8.1.1.2. 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").

8.1.1.3. 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. The objects "test.c :inc", "test.c :key", and "test.c :o" have "test.c" as their base object. 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".

8.1.1.4. 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, 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 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.

8.1.1.5. 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 has been described.

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. 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.

8.1.1.6. 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).

8.1.1.6.1. 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.

8.1.1.6.2. Verification Dates.

There are two "verification dates" associated with an object. The "primary verification date" indicates 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 object is valid.

8.1.2. 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.

8.1.2.1. 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 5 and 6. Figure 5 contains the Source Lists and Figure 6 contains the Output Lists.

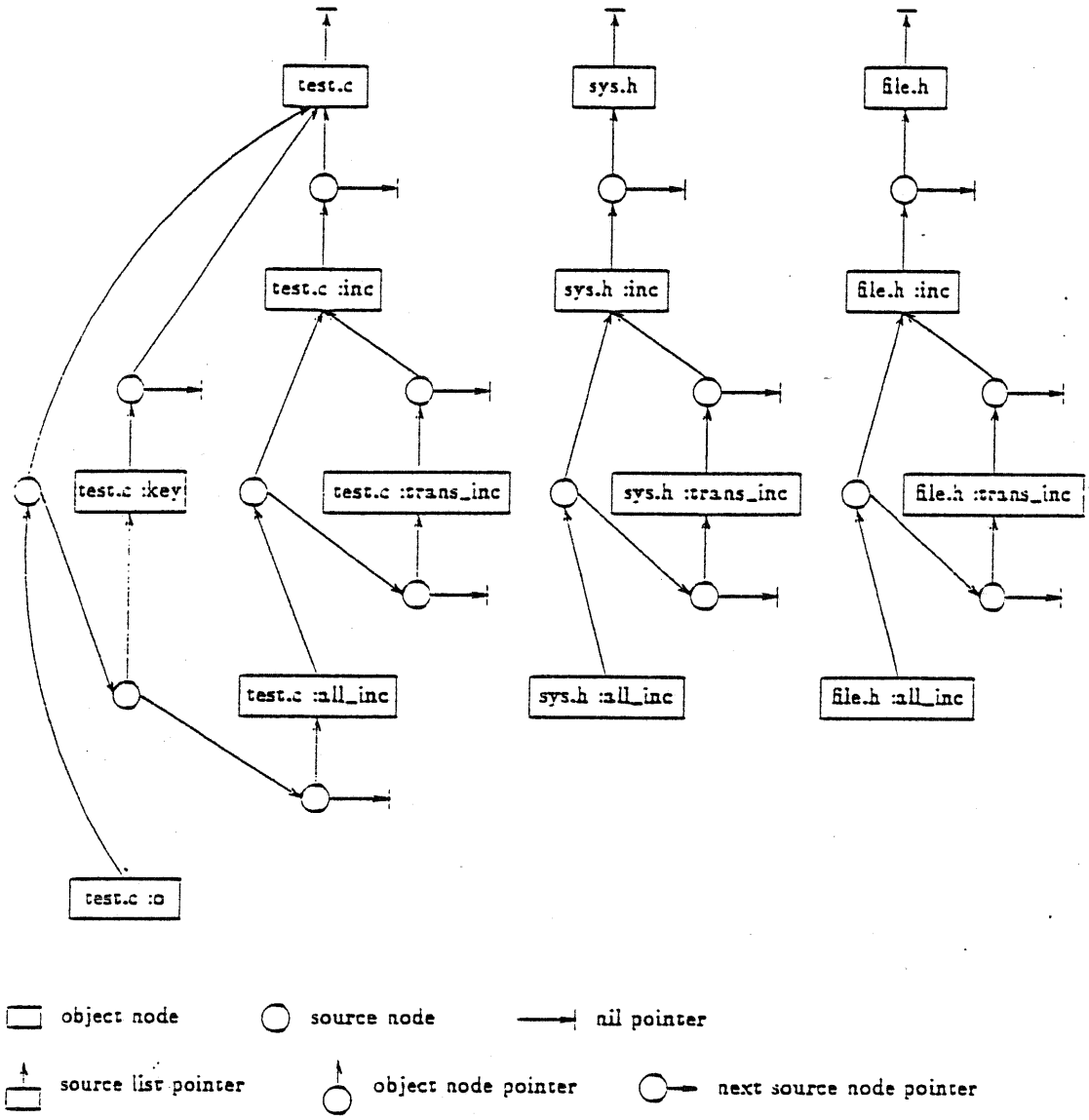
test.c
sys.h
file.h
test.c :inc
sys.h :inc
file.h :inc
test.c :key
test.c :trans_inc
sys.h :trans_inc
file.h :trans_inc
test.c :all_inc
sys.h :all_inc
file.h :all_inc
test.c :o
object node
source node
nil pointer
source list pointer
object node pointer
next source node pointer

Figure 5

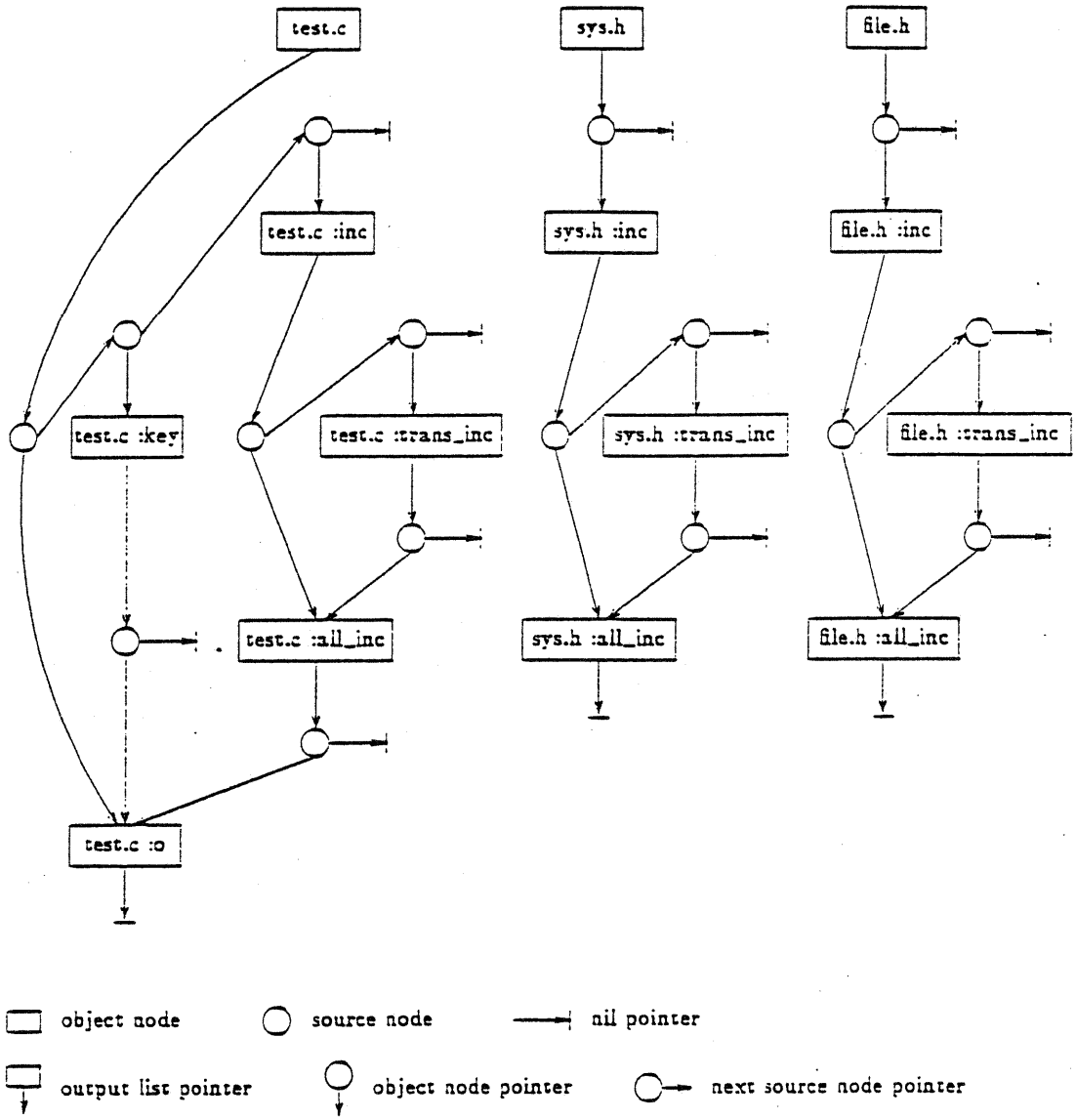
test.c
sys.h
file.h
test.c :inc
sys.h :inc
file.h :inc
test.c :key
test.c :trans_inc
sys.h :trans_inc
file.h :trans_inc
test.c :all_inc
sys.h :all_inc
file.h :all_inc
test.c :o
object node
source node
nil pointer
output list pointer
object node pointer
next source node pointer

Figure 6

8.1.3. 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.

8.1.3.1. 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 and 8. Figure 7 contains the Element Lists and Figure 8 contains the Compound Lists.

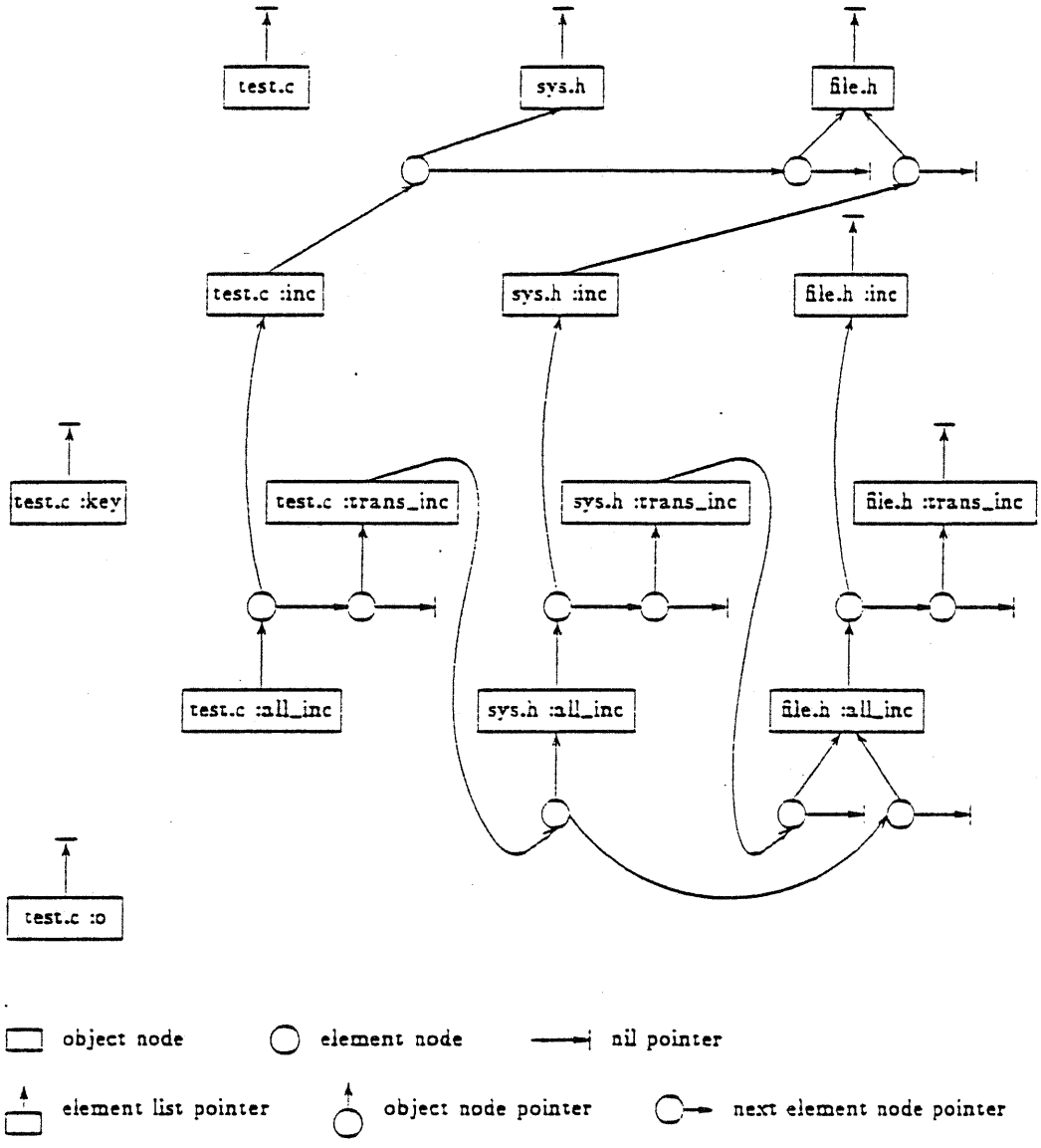
test.c
sys.h
file.h
test.c :inc
sys.h :inc
file.h :inc
test.c :key
test.c :trans_inc
sys.h :trans_inc
file.h :trans_inc
test.c :all_inc
sys.h :all_inc
file.h :all_inc
test.c :o
object node
element node
nil pointer
element list pointer
object node pointer
next element node pointer

Figure 7

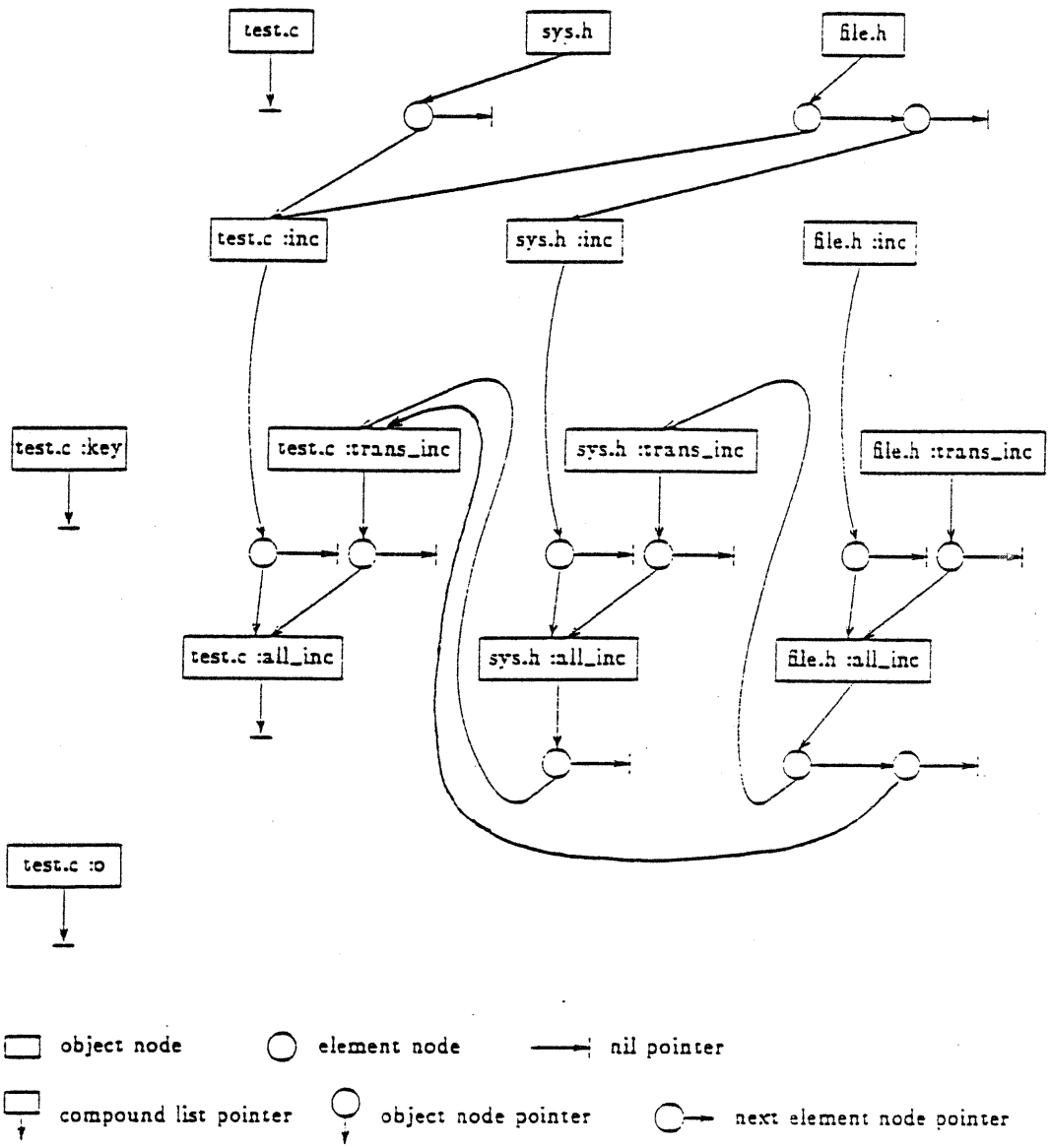
test.c
sys.h
file.h
test.c :inc
sys.h :inc
file.h :inc
test.c :key
test.c :trans_inc
sys.h :trans_inc
file.h :trans_inc
test.c :all_inc
sys.h :all_inc
file.h :all_inc
test.c :o
object node
element node
nil pointer
compound list pointer
object node pointer
next element node pointer

Figure 8

8.1.4. 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.

8.2. Concurrent Access.

To ensure correct usage of the Odin database (the INFO file) 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.

8.3. 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.

8.4. The Specification Language Compiler.

The specification language compiler translates the user specification into a sequence of tables designed for efficient interpretation by the Odin interpreter. In addition to the straightforward mapping from the symbolic specification to

internal data structures, some preprocessing of the user specification is performed.

8.4.1. Disambiguation of Queries.

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

The motivation for this decision is that users making requests to the Odin system are not expected to understand the tool fragments being 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.

8.4.2. 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 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
```

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. The compiler and linking loader are not interested or affected by the "stdin" parameter.

9. Experiences with Odin.

In this section we describe two major tool integration activities which used Odin. Odin has been used in a number of other tool integration projects, but the two described here seem particularly noteworthy and exemplary.

9.1. The Toolpack Project.

As was stated earlier, much of the impetus for creating Odin came from the

need to create a tool integration system for the Toolpack project. Toolpack required a tool integtration system capable of effectively integrating a large collection of large and powerful tools, and doing so in a way which would be amenable to the steady alteration and augmentation of the toolset. In these respects it appears that the needs of Toolpack are not too dissimilar to the needs of most modern environments.

Odin has been used successfully to integrate such a collection of tools into the Toolpack/IST system. The tools which have been integrated range in power from simple text manipulators to complex data flow analysis tools. The toolset includes tools which have been produced at the University of Colorado, tools produced elsewhere, and tools furnished by vendors only in executable form. Some tools have very simple connectivity and interaction with the rest of the toolset, while others (eg. the parser) must be interfaced with a large variety of other tools. Some tools (eg. the editors) are highly interactive, while others are batch oriented.

This broad diversity of tools has been successfully incorporated into Toolpack/IST. In order to do so it has from time to time been necessary to make changes in Odin's capabilities. Most of these changes have been made to the Specification Language, as we have found that unexpectedly complex and powerful descriptive mechanisms are necessary in order to model the range of capabilities and interactions among tools actually in use and needed by users. At present we find that Odin's current capabilities seem adequate to support the quick and easy installation of tools. In some cases, new tools have been successfully incorporated in as little as five minutes.

Toolpack/IST is a prototype system which is not currently supported or distributed elsewhere. Its architecture and set of initial tool fragments has formed the basis of the Toolpack/1 system which is currently being distributed by Nag Ltd., Oxford, England. The development paths of these two systems have already begun to diverge. In this section we describe experiences with, and inferences drawn from, Toolpack/IST.

Although Toolpack/IST was originally designed and implemented as a collection of tools to support the backend phases of development and maintenance of Fortran programs, it has steadily emerged as an environment capable of supporting other languages as well. In particular, as experience with Odin and the early versions of Toolpack/IST grew it became clear that there was no reason to shrink from incorporating tools for other languages such as c. In fact, as Odin itself is written in c, the first user of Odin was G. Clemm, who was developing Odin. Clemm rapidly discovered that simply by integrating the c compiler, a text editor and a very small number of other modest tools under Odin a surprisingly and gratifyingly powerful environment was created.

Much of the perceived power of Odin-integrated toolsets seems to arise from its object orientation. Thus the user of even a modest collection of tools can easily identify small but important objects which may be produced only as the result of long and complex tool applications. For example, the object containing all errors arising from compiling and loading a large collection of procedures into an executable is such an object. We rapidly discovered that, once all source code for Odin had been assembled into a structure comprising the entire Odin program, changes and additions could readily be made and evaluated. This process turned out to be a small iterative loop entailing the use of a text editor to make changes and additions, and then a request to persue the error object produced by the loader. Odin intelligently and efficiently determined which source code procedures had to be recompiled (only those which had been changed), invoked the loader (only if the new compilations has effected changes in the object code), and assembled all error messages from both the compiler and loader into a single object which was then presented to the user. The net effect was a feeling of operating on large, high level objects easily and efficiently.

We have been impressed by the ability of Odin to meet the initial needs of integrating the Toolpack toolset, and have also been pleased at the way in which the concepts upon which Odin is based have proved to be sufficiently general and robust to support capabilities beyond what was originally envisioned. Specifically, it seems that Odin is able to support software development in more than one language, is able to integrate existing tools as well as new ones, any may be more effective in supporting large software development than smaller software development.

In the next subsection we describe experiences in applying Odin to the integration of a class of tools that is quite different from the sorts of tools which were our initial targets.

9.2. The Integration of the GAG Toolset.

In March of 1985, after the Odin system had been successfully used to integrate the Toolpack tool system and most common Unix tools into Toolpack/IST, 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 Odin.

The system selected for this test was the GAG attribute grammar system

[Kastens 82]. 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.

Details of the way in which GAG was integrated under Odin and specifics of our experiences in doing so can be found in [Clemm 86]. Here we simply summarize this work.

We were pleased to find that much of the work done in the GAG procedure files was directed towards managment, control and intertool communications issues which we had taken as major foci of our Odin work. For example, determining the nature and severity of errors occuring during the processing of one tool, and deciding how to proceed in view of such errors was a significant problem addressed in the previous GAG system. Facilities for handling these sorts of problems exist in Odin and are described earlier in this paper.

As a result of the similarlity of the capabilities needed by GAG and supplied by Odin, we found that integration of GAG under Odin was not too difficult. The major problem, unsurprisingly, was our need to familiarize ourselves with the GAG system components, tools and procedures. Once these were understood we were able to rather straightforwardly map them onto Odin object types and tool fragments, and to express needed procedures in terms of Dependency Graph paths.

The result of this straightforward process was a version of GAG that ran under Odin, but seemed to offer few execution speed or space efficiencies. We did conclude that the user view presented by the Odin-integrated GAG system was cleaner and simpler, but must admit that this is a subjective appraisal to some extent.

The second phase of this activity was more interesting and rewarding. During this phase, we scrutinized the structure of the GAG system and attempted to detect ways in which the philosophy and approaches of the Odin system might be more effectively applied to integrating GAG, in the hope that some significant efficiencies might be gained.

There are three main methods that would seem useful in improving 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.

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.

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.

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 (this was described in a section of the earlier discussion of the Specification Language), and the values for parameters are specified by the user at run time using the parameterization operation (described during the discussion of the Request Language). 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.

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. Some instances of this were found in the original version of the GAG system, and eliminated in the Odin-integrated version

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 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 reuse 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.

We made effective use of the notion of object parameterization in the Odin-integrated version of GAG. Significant benefits from doing so were observed. This experience helps strengthen our opinion that it is useful and natural to consider software objects to be labelled by the activities which have created them. Clearly this labelling should include a specification of the tools which created them, but in addition, this experience indicates that the labelling should also reflect any adaptations made to the tool functions which have created them.

The effects of the optimizations described above varied considerably, as might be expected, depending upon the nature of the requests made by users. In some cases, however, the savings from reuse were so considerable that speedups of a factor of ten were measured. In other cases no improvements were noticed at all. Savings of storage space are far harder to evaluate. The Odin philosophy of supplying any specifiable object to a user in response to a request, regardless of whether the object is in store or not, and regardless of whether it has ever been created or not, implies that the space occupied by the objects in store can be prespecified with the only visible effect being longer times needed to recreate objects when less storage is made available. In the GAG experiment, we did not attempt to vary the amount of storage made available. Had we done so, we

would doubtlessly have been able to demonstrate that the Odin-integrated system could run successfully in less space than the original system occupied. Execution speed might have suffered, although this would certainly have depended upon the sequences of requests made by the user.

Thus there can be no absolute claims for reduction in the amount of storage needed. Certainly, any amount of storage sufficient to support an Odin-integrated GAG procedure, can also be made sufficient to support execution of the same GAG procedure not using Odin. This, on the other hand, would generally require lengthy, painstaking and error-prone alteration of the GAG procedure file. Odin's ability to effectively support tool execution in varying amounts of space automatically is probably its greatest contribution to storage efficiency. In the end, this makes it possible for users to make time vs. space tradeoff decisions and expect them to be effectively supported by Odin.

We believe that this research project has left us with a far clearer picture of the value of various principles and architectures for integrating software tools into effective environments. In the broadest terms, we now see the Odin research effort as having been directed towards studying the use of an object management system to integrate tools. Within this context we made, implemented and evaluated some specific architectural choices. Specifically we chose to integrate smaller tool fragments, but manage relatively large grained objects. We chose to base our strategy for managing objects upon the notion of organizing them by means of two primary relations--hierarchy and derivation. We chose to facilitate flexibility and extensibility and did so by materializing the underlying structure of object types and tool fragments, making this structure essentially an object itself.

In the following subsections we summarize the conclusions to which we have come in evaluating each of these architectural directions and decisions.

9.3. Object Orientation.

We believe that our experiences with using Odin have shown that centering an environment around an object store is a very powerful and effective way of integrating the capabilities of the environment. In Toolpack/IST we used this approach to render the functional capabilities of a wide variety of tools far more easily and economically accessible to users. This experience was reinforced in the integration of the GAG toolset, where we found that the identification of the underlying objects in need of creation and management also

served to sharpen understanding of the relations among the GAG tools, leading to more efficiency in applying them.

At the very least, this project has demonstrated to us that some prior system in which tools themselves were made the center of attention erred in not giving at least equal attention to the data objects which are the products of (and inputs to) those tools. This is because we identified a number of situations in which it is quite natural and easy to name a desired object which can, nevertheless, only be constructed by a long and complex chain of applications of tools.

9.4. Tool Granularity.

Our decision to supply the functional effect of larger tools by concatenating smaller tool fragments followed logically from our more fundamental decision to center our environments around objects rather than tools. As a consequence of studying the object structure of our environments we quickly realized that most tool functions use and create a surprising variety of data objects. Examination of these objects quickly led to the discovery of an identifiable functional structure of many common tools. In the case of a tool such as a compiler, this structure (eg. lexical analyzer, syntactic analyzer, semantic analyzer, optimizer, code generator) was already quite well known.

In a broader sense, this predisposition towards viewing tools as aggregates of tool fragments is really nothing more, nor less than a desire to determine the modular decomposition of tools, and then identify commonly used modules. As such, it seems that there should be no need to further justify this architectural decision.

9.5. Object Granularity.

Our decision to construct Toolpack/IST and to reintegrate GAG around relatively large grained objects was dictated by two or three key factors. Before addressing them, however, it should be observed that Odin itself is not inherently a manager of large object or small objects--it is simply a manager of objects. In fact, we even experimented briefly with using Odin to manage objects which were integers by tool fragments which were simply arithmetic operators. The result was an amusing, though inefficient computational system.

That experiment, however, exemplified the overhead costs which Odin incurs in order to manage objects effectively. Thus it becomes increasingly expensive to have Odin manage objects as those objects need to be accessed frequently. Further, as the size of the software project being supported grows, and the

amount of data to be managed increases, the ability to directly access all data objects becomes increasingly expensive. This increasingly suggests that larger data objects which are aggregates of smaller objects be managed. Thus, in Toolpack/IST we chose, for example, to manage entire lexical strings rather than tokens, and entire symbol tables rather than individual symbols. This enabled users to access the large-grained objects easily, but was not help to them in accessing the smaller objects. For this, special purpose tools had to be produced. These tools inevitably presented a non-uniform appearance to users. Worse still, this approach interfered with our ability to show relations among the smaller-grained objects which were constituents of different large-grained objects.

We believe that our work has focussed attention on the need to manage large-grained objects, and the fact that this is not at all straightforward. Our work has not, however, obviated the need for effective management of small objects as well. We believe that an object management system capable of efficiently and effectively managing a broad spectrum of types and sizes of objects is needed. Odin is capable of doing this management, but, in its current prototype form is likely to be unacceptably inefficient in doing so.

9.6. Organization of the Object Store.

We have just indicated that one of the motivations for managing large grained objects is to keep down the number of objects which must be directly accessible and therefore directly managed. The sheer volume of objects can be a serious impediment to effective access. More of an impediment, however, is the need to represent complex interrelations among these objects.

We sought to develop an organizational structure for our object store which would be adequate to represent the relations among our objects, but not so complex as to pose serious efficiency problems. We rejected the idea of using a full relational database to manage our software objects, fearing that this would encourage the specification of more relations than needed, and render too inefficient the task of reflecting changes in objects by having to propagate them widely around the database.

Instead we chose to use hierarchy and derivation as the only two relations for organizing the objects in our store. These two relations proved to be effective in supporting the integration of the toolsets described in this paper. We believe that both of these relations are essential organizing agents in any object store that is to support an environment effectively. Hierarchy enables the efficient and effective manipulation of large and complex software objects, and enables the user to think and work at varying levels of abstraction as needed.

Derivation is essential in keeping track of which objects can and cannot be spooled and purged, and which must be altered or updated in response to changes in others.

Thus, we believe that these two relations are a minimal set of relations needed in an environment object store. The suspicion persists, however, that it will be useful to maintain other relations, especially when the range of sizes of objects maintained in the object store is expanded.

9.7. Flexibility and Extensibility.

We are quite satisfied that the architecture we have proposed and implemented in Odin is supportive of the need to alter and extend the functional capabilities of an environment. The key to being able to do this is the isolation in a single structure of all information about how objects of the various types are created from each other by the action of tools. In Odin this information is contained in the Dependency Graph. Tool fragments have no knowledge of how they relate to each other, thereby avoiding the necessity of modifying existing tools in order add new tools and making it possible to alter a tool without having to also alter any other tools as well.

In an important sense, the body of information describing tool and object type relations--in Odin the Dependency Graph--should be viewed as an object. It is the object used to schedule the coordination of tool interaction, and is the object which must be modified in order to make any changes in the tool or type structure of the environment being supported. These processes are all facilitated in Odin by the creation of a language with which to describe the Dependency Graph and tools to support such functions as the compilation of a dependency graph description into a graph, the viewing of such descriptions, and the alteration of such descriptions.

Our experiences in integration diverse tool and object types and in rapidly supporting alterations and extensions of the toolsets integrated by Odin strongly support our view that it is important to materialize the type and tool interaction structure of an environment as an object.

10. Future Research Directions.

Our experience with Odin has convinced us that the basic ideas underlying it are sound and form an effective basis for the integration of software environments. Having completed this research, however, it is now far easier to see

ways in which these basic ideas can and should be extended. Future research is needed to pursue these ideas to these logical conclusions.

10.1. Varying grain sizes.

We now believe that an environment must be effective in managing objects whose sizes range from very large to very small. Although Odin has been proven effective in managing large objects, we are convinced of the importance of also managing the smaller objects of which they are composed. The problem of efficient management of the very large collection of objects created by a large software project is a serious one that would have to be addressed. Further, the problem of designing and implementing an interface which was both uniform and effective in furnishing users access to these very different sorts of objects also seems to be a serious one in need of further research.

10.2. User Interface.

This project has also sharpened our appreciation of the problems of providing users suitable access to the resources managed by an environment. As has been noted above, our primary focus in the Odin project was on providing capabilities for managing objects which were primarily large grained. The access we provided to these objects was through Odin's clean and uniform command language. We found, however, that most users spent most of their time dealing with the smaller objects contained in these larger objects. Access to these smaller objects was provided through viewer tools which were created for each object type. These viewers were generally custom built and not standardized. As a result users were confronted with a very non-uniform interface to the smaller objects with which they spent most of their time.

It is now clear that viewer tools must be treated in the same way as other tools in a well designed environment--namely composed out of smaller, modular tool fragments. Active research is needed to address the problem of identifying a suitable set of modular viewing tool fragments and demonstrating that they can be implemented in such a way as to support effective composition into special purpose viewers for the range of object which a general purpose environment must integrate.

In addition, it is clear that viewers are needed to assist users in understanding and exploiting the environment command execution process itself. The help facilities provided by Odin are a very primitive beginning in the process of helping users understand what Odin does and how it works. Structures such as the Derivative Forest and the Dependecy Graph are as central to the user's

understanding of the software objects being created and managed as they are to Odin's ability to manage them effectively. A key part of an environment's user interface must be a facility for depicting the object store and showing the user how it is being changed by command executions. It is unclear how best to provide this user view, and how it might relate to the viewing capabilities needed for the finer-grained

10.3. Environment Command Languages as Programming Languages.

We believe that Odin should be thought of as a language for describing, creating and manipulating software objects. On the other hand, we now understand that the collection of linguistic capabilities incorporated into Odin has not been studied as carefully as it might have. An important future direction of this research is to round out the Odin command language into a cleaner and more orthogonal language.

The Odin object typing mechanism seems clearly in need of such study. Odin currently furnishes a mechanism for creating instances of existing types and for extending the current type structure. Although we have stressed our belief that Odin objects should be thought of as instances of abstract data types, it is clear that we have not implemented mechanisms for enforcing the sort of strict discipline in object access that is a goal of using data abstractions. It is possible to supply a cluster of accessing primitives that are to be used by Odin tools in accessing instances of a defined type, but Odin does not prevent the accessing of such instances by other means.

Further, Odin combines the structure of object types and tool fragments into one structure--the Dependency Graph. While we are convinced that the Dependency Graph is a useful structure for expressing the way in which the various tool fragments interrelate, we are less certain that it is an adequate vehicle for expressing the type structure. We need to explore the use of subtyping mechanisms, for example, as a device for adding further structure to the object store. This raises the possibility of exploiting inheritance as a means for simplifiying the problem of providing powerful and uniform sets of accessing primitives for Odin object types.

In addition, we note that the Odin command language provides virtually no explicit mechanisms for altering control flow. Clearly a great deal of control flow is done implicitly. Compound objects are processed by implicit loops, and object creation is done conditionally based upon error status flags, and detection of significant alterations made to related objects. While it seems to us to be clearly useful to have these control flow operations concealed from the user, we believe that it is still necessary to give the user some amount of control over

processing sequence. Certainly, the absence of explicit control flow mechanisms gives the Odin command language a lopsided appearance.

Pursuing this line further, we are led to consider the sorts of programs which users would write in such a command language. Such programs would be tantamount to procedures for expressing either parts of or entire software processes. This raises the interesting possibility that an extension of the Odin command language could be used to express software processes as algorithms for creating and manipulating software objects through software tools. The Odin system would then correspond to a system for compiling and interpreting such procedural descriptions.

11. Acknowledgments.

The work described here has been strongly influenced by Stuart I. Feldman, formerly of Bell Telephone Laboratories, now of Bell Communications Research. Feldman's Make processor served as an important starting point for our own work. Numerous conversations with Feldman served to sharpen our insights and mold this work.

In addition our research colleagues at the University of Colorado, Boulder were a constant source of challenging and important feedback and comment.

Finally, we gratefully acknowledge the support of the National Science Foundation and the US Department of Energy.

References

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

[Baker 77]
B.S. Baker, "An Algorithm for Structuring Flowgraphs", JACM 24 #1 pp. 98-120 (January 1977).

[Bazelmans 85]
R. Bazelmans, "Evolution of Configuration Managment," SIGSOFT Software Engineering Notes, Vol. 10, pp.37-46, Oct. 1985.

[Boehm 83]
Boehm, B. W. and T. . Standish, Software Technology in the 1990's : Using an Evolutionary Paradigm, Computer (16,11) November 1983.

[Buxton 80]
J.N. Buxton, V. Stenning, "Requirements for Ada programming support environments", Stoneman, DOD (February 1980).

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

[CDC 76]
"Modify Reference Manual," Pub. # 60281700, Control Data Corporation, 1976.

[Clemm 84]
G.M. Clemm, "ODIN - An Extensible Software Environment", University of Colorado, Dept. of Computer Science Technical Report CU-CS-262-84, 1984.

[Clemm 86]
G.M. Clemm, "The Odin System--An Object Manager for Extensible Software Environments," Ph.D. Thesis, Department of Computer Sci., Univ. of Colorado at Boulder, 1986.

[ClemOst 86]
G.M. Clemm and L.J. Osterweil, "An Introduction and User's Guide to the Odin Extensible Software Environments Integration System," University of Colorado at Boulder, Dept. of Computer Science Tech. Report, April 1986.

[Cooprider 79]
L. Cooprider, "The Representation of Families of Software Systems," Ph.D. Thesis, Department of Computer Sci., Carnegie-Mellon Univ.,1979.

[Cristofor 80]
E. Cristofor, T.A. Wendt and B.C. Wonsiewicz, "Source Control + Tools = Stable Systems," Proc. 4th Computer Software and Applications Conf.,

pp. 527-532, Oct. 1980.

[DEC 84a]
CMS Code Management System, Digital Equipment Corp., 1984.

[DEC 84b]
MMS Module Management System, Digital Equipment Corp., 1984.

[DeJong 73]
S.P. DeJong, "The System Building System," Tech.Rpt. #RC 4486, IBM Thomas J. Watson Res. Center, Yorktown Hts., NY, 1973.

[DeK 75]
F. DeRemer and H. Kron, "Programming-in-the-Large Versus Programming-in-the-Small"", IEEE Transactions on Software Engineering, SE-2, No. 2, pp.80-86, June 1976.

[Donzeau-Gouge 84]
V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang, , "Programming Environments Based on Structured Editors: The MENTOR Experience", in Interactive Programming Environments, Barstow et. al. (eds.) pp. 128-140, 1984.

[Erikson 84]
V.B. Erikson and J.F.Pelligrin, "Build--A Software Construction Tool," AT&T Bell Laboratories Tech. Journal, v.63, pp.1049-1059, Aug. 1984.

[Feldman 79]
S.I.Feldman, "Make--A Program for Maintaining Computer Programs," Software--Practice and Experience, v. 9, pp.255-265, 1979.

[Glasser 78]
A.L. Glasser, "The Evolution of a Source Code Control System," SIGSOFT Software Engineering Notes, v. 3, pp.122-125, Nov. 1978.

[Habermann 79]
A.N. Habermann, "An Overview of the Gandalf Project", Carnegie-Mellon University Computer Science Research Review 1978-1979, 1979.

[Huff 81]
K. Huff, "A Database Model for Effective Configuration Management in the Programming Environment," Proc. 5th Intl. Conf. on Software Eng., pp. 54-61, 1981.

[Kastens 82]
U.Kastens, B.Hutt, E. Zimmerman, GAG: A Practical Compiler Generator, Springer, 1982.

[Lampson 83a]
B. Lampson and E. Schmidt, "Practical Use of a Polymorphic Applicative Language," Proc. 10th POPL Conf., 1983.

[Lampson 83b]
B. Lampson and E. Schmidt, "Organizing Software in a Distributed Environment," SIGPLAN Notices, v 18, June 1983.

[Leblang 84]

D.B. Leblang and R.P. Chase, "Computer Aided Software Engineering in a Distributed Workstation Environment," SIGPLAN/SIGSOFT Symposium on Practical Software Development Environments, Pittsburgh PA, April 1984.

[Leblang 85a]
D.B. Leblang and G.D. McLean, "Configuration Management for Large Scale Software Development Efforts," Workshop on Software Engineering for Programming-in-the-Large, Harwichport, June 1985.

[Leblang 85b]
D.B. Leblang, R.P.Chase and G.D. McLean, "The DOMAIN Software Engineering Environment for Large Scale Software Development Efforts," Proc. IEEE Conf. on Workstations, San Jose CA, Nov. 1985.

[Kernighan 81]
B.W. Kernighan and J.R. Mashey, "The Unix Programming Environment", Computer, April 1981, pp. 12-24.

[Linton 84]
M.A. Linton, "Implementing Relational Views of Programs," SIGPLAN/ SIGSOFT Symp. on Practical Software Development Environments, Pittsburgh PA, April 1984.

[Osterweil 76]
L.J. Osterweil and L.D. Fosdick, "DAVE - A Validation, Error Detectionn, and Documentation System for FORTRAN Programs", SP&E 6, pp. 473-486 (Sept. 1976)

[Osterweil 81]
L. J. Osterweil, "Software Environment Research Directins for the Next Five Years", COmputer 14 pp.35-43, (April 1981).

[Riddle 83]
W.E. Riddle, "The Evolutionary Approach to Building the Joseph Software Development Environment", Proc. IEEE Softfair - Software Development Tools, Techniques, and Alternatives, pp. 317-325, 1983.

[Ritchie 74]
D.M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," CACM, v 17, pp.364-375, July 1974.

[Rochkind 75]
M.J. Rochkind, "The Source Code Control System," IEEE Trans. on Software Eng., SE-1, pp. 364-370, Dec. 1975.

[Rudmik 82]
A. Rudmik and B. Moore, "An Efficient Separate Compilation Strategy for Very Large Programs," Proc. SIGPLAN 82 Symp. on Compiler Construction, pp. 301-307, June 1982.

[Ryder 74]
B.G. Ryder, "The PFORT Verifier", Software - Practice and Experience", 4, pp. 359-377, (1974).

[Schmidt 82]
E.E. Schmidt, "Controlling Large Software Development in a Distributed Environment," Ph.D. Thesis, Computer Science Division, EECS Dept.,

Univ. of Calif, Berkeley, Dec. 1982.

[Taylor 84]
R.N. Taylor and T.A. Standish, "Steps to an Advanced Ada Programming Environment", Proc. 7th Int. Conference on Software Engineering, pp. 116-125, 1984.

[Teitelbaum 81]
T. Teitelbaum and T. Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment". CACM 24 (September 1981) 563-573.

[Teitelman 81]
W. Teitelman and L. Masinter, "The Interlisp Programming Environment", Computer, pp.25-33, 1981.

[Teitelman 84]
W. Teitelman, "A Tour Through Cedar", 7th Int. Conf. on Soft. Eng., pp.181-195, 1984.

[Thall 83]
R. Thall, "Large-Scale Software Development with the Ada Language System," Proc. ACM Computer Science Conf., Feb. 1983.

[Tichy 82]
W.F. Tichy, "Design, Implementation, Evaluation of a Revision Control System," Proc. 6th Intl. Conf. on Software Engineering, pp.58-67, Sept. 1982.

[Wasserman 83]
A.I. Wasserman, "The Unified Support Environment : Tool Support for the User Software Engineering Methodology", Proc. IEEE Softfair - Software Development Tools, Techniques, and Alternatives, pp. 145-153, 1983.

[White 77]
J.R. White, R.K. Anderson, "Supporting the Structured Development of Complex PL/I Software Systems," Software--Practice and Experience, v. 7, pp. 279-293, 1977.