

A Mechanism for Environment Integration

GEOFFREY CLEMM and LEON OSTERWEIL

University of Colorado at Boulder

This paper describes research associated with the development and evaluation of Odin—an environment integration system based on the idea that tools should be integrated around a centralized store of persistent software objects. The paper describes this idea in detail and then presents the Odin architecture, which features such notions as the typing of software objects, composing tools out of modular tool fragments, optimizing the storage and rederivation of software objects, and isolating tool interconnectivity information in a single centralized object. The paper then describes some projects that have used Odin to integrate tools on a large scale. Finally, it discusses the significance of this work and the conclusions that can be drawn about superior software environment architectures.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques; D.2.6 [**Software Engineering**]: Programming Environments; D.4.2 [**Operating Systems**]: Storage Management

General Terms: Experimentation, Languages

Additional Key Words and Phrases: Environment integration, persistent objects, software object management

1. BACKGROUND

A primary goal of software environment research is to devise superior development and maintenance processes and effectively integrate software tools to support them. The development of a large software system can easily cost between \$50 and \$400 per line [1], yet the quality of the end product is often disturbingly low. The cost of maintaining such systems over their lifetime usually far exceeds original development costs. Further, software costs have been steadily increasing over the past decade. Most observers blame the twin problems of high cost and low quality on the lack of orderly, systematic processes for developing software and the lack of software tools that effectively exploit computing power to support them. Accordingly, there has been much work on software tools and processes during the past decade. There have been some proposed software processes that stress extensive documentation of software products. Often these are not widely

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

Authors' current addresses: G. Clemm, Evolutionary Software, 55 Commonwealth Road, Watertown, MA 02172; L. Osterweil, Department of Information and Computer Science, University of California at Irvine, Irvine, CA 92717.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1990 ACM 0164-0925/90/0100-0001 \$01.50

used because they are onerous without significant tool support. Many tools are not used because they do not support specific processes, or are not well integrated with each other.

Thus, software is usually developed in a manual, ad hoc way. Even where effective software processes are defined, they are rarely adequately supported by tools, making it difficult to experimentally determine just how effective they are and whether they should be promulgated more widely. Similarly, it is difficult to definitively evaluate individual tools, as they often do not clearly support tasks within popular processes, and are thus not comparable to manual efforts or other tools. We believe that the emergence of software environments is catalyzing improvements in processes and tools by providing a common evaluative framework. This paper describes a research software system called Odin [3, 4], which facilitates the rapid and effective integration of software tools, thereby aiding experimentation and evaluation.

We believe Odin is a language and interpretation system for what DeRemer and Kron have called “programming in the large” [5]. Odin was first used to integrate a family of FORTRAN development, testing, and maintenance tools in the Toolpack project [16]. It was subsequently used to integrate tools to support software development in C [8], and then to integrate tools to support the creation of attribute grammars. Currently, Odin is being used to support a variety of large-scale tool integration applications, including a data flow analyzer generator project [14]. The Odin architecture, our experiences in exploiting it, conclusions about the Odin architecture, and principles to which these experiences have led us, are the subjects of this paper.

In Section 2 we describe the philosophical basis upon which Odin rests, Section 3 describes related work; Section 4 describes the details of Odin’s design and some implementation issues; Section 5 the language used to specify the way in which tools are to be integrated; Section 6 the language that users employ to actually use tools; Section 7 our experiences in using Odin; and Section 8 suggests likely future research directions.

2. THE ODIN TOOL INTEGRATION PHILOSOPHY

Odin was originally designed as part of the Toolpack project [16], whose aim was to integrate both existing and proposed tools for supporting the development, testing, and maintenance of FORTRAN code. Flexibility and extensibility were also key Toolpack goals. At the time (1979) there was considerable interest in software environments, but very few had been built, and there were virtually no successful environment architectural paradigms to guide our work. Thus, the design and evaluation of an innovative environment integration paradigm became another major goal of the Toolpack project.

The Odin project followed the suggestion (e.g., [2, 15, 17]) that an environment must be data centered, rather than tool centered, and adopted the philosophy that Odin-integrated environments should be devices for using tools to manage repositories of software data in such a way as to expedite responses to interactive user commands.

Accordingly, a primary design issue was determination of the contents and structure of the repository. In Odin-integrated environments, the repository

consists exactly of those software objects that are the inputs to, and outputs from, the various tools to be integrated. Thus, for example, such code-related objects as source code, object code, parse trees, symbol tables, and flow graphs populate the repository, but so do such noncode-related objects as documentation, test data, test results, and program structure representations.

Odin manages these large-grained objects by coordinating and managing the application of correspondingly large grained tools such as parsers, instrumenters, prettyprinters, and data flow analyzers. Odin views each software object as an operand, and the tools that manipulate the objects as operators that manage these objects and their relations to each other. Odin facilitates the synthesis of these tools into larger tools, while relieving users of the burden of figuring out the details of how to do this synthesis. Odin enables users to create objects that may be derivable only by a complex process involving many tools, simply by specifying the object in a precise yet terse way. The object may have been created earlier, it may have been built out of objects that have subsequently been changed, or it may never have been created before at all. In all of these cases, Odin creates an efficient derivation process for building or rebuilding the desired object and executes that process. Often, Odin makes considerable use of existing objects to speed creation of the newly requested object.

Odin can be thought of as an interpreter for a high-level command language whose operands are the various software objects in the data repository and whose operators are tool fragments. Later, we show how these operands can be aggregated into structures, 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.

3. RELATED WORK

Odin's approach to tool and object management has some unique features, but also builds upon concepts found in other systems.

Odin builds upon the UNIX[®] view of tools, in that it is aimed at the easy concatenation of loosely coupled tool fragments into larger tools. UNIX, however, only supports simple-minded tool concatenation comfortably. UNIX tool fragments are easily combined by connecting consecutive tool fragments by a single data stream, called a pipe. Effective software development and maintenance, however, require complex tools with multiple connections. Thus Odin supports the synthesis of complex tools out of tool fragments that communicate with each other through multiple data streams.

Odin also builds upon the Make [6] model of software object management. Make automatically applies tools to assure that changes to some objects are reflected in other related objects. Odin supports this object management feature as well. With Make, however, for all but single-input/single-output tools, the user must explicitly name each individual object which is to be automatically updated and the exact sequences of tools to be used in doing so. Odin, on the other hand, enables users to define types of objects and to prescribe general procedures for automatically creating and updating instances of those types from corresponding instances of other types.

[®] UNIX is a trademark of AT&T Laboratories.

Thus, the Odin user may create a new object and then immediately request a derivation requiring the complex synthesis of many diverse tools and the creation of many intermediate objects. A Make user would have to set up directory structures naming all of these objects and would have to define the derivation process in detail. Odin automatically constructs storage structures as needed, in accordance with the derivation process that it creates from a more general specification of how tools interconnect.

The key construct in doing this is the Odin Derivation Graph, which models the way in which tools can be synthesized. The edges in this graph correspond to the tools that Odin integrates. The nodes correspond to types of objects. User requests are treated as requests for named objects, which are considered to be instances of types contained in the Derivation Graph. Graph traversal is used to determine which tools must be used to create which specific instances of which intermediate types in order to eventually satisfy the dataflow needs of the tools that produce the requested objects. Objects that have been requested before are likely to have been retained to expedite response to subsequent user requests. As with Make, however, Odin does not return retained objects to the user if the objects from which they have been built have changed. Instead, Odin automatically rederives the requested object from its changed predecessors. The Derivation Graph is a key and complex feature of Odin, and is described in considerable detail in Section 5.

Arbitrarily complex new tools (and object types) are integrated by editing models of their interactions with existing tools into the Derivation Graph. This represents a great improvement over the way in which Make exploits new tools. In Make, except when adding simple UNIX pipe-style tools, every object that is to be either directly or indirectly built using the new tool must have its Makefile altered to show precisely how the new tool is to be used.

The System Modeler [9, 10, 18], developed at Xerox for the Cedar programming environment, and Apollo's DSEE (Domain Software Engineering Environment) [11–13] both incorporate a more explicit tool and object model than does Make. For example, System Modeler explicitly captures the object flows between the Cedar editor and the Mesa compiler/linker. Both DSEE and Cedar are examples of very tightly integrated environments in which both the tools and integration mechanisms have direct knowledge of each other. Users enjoy important benefits from such tightly coupled toolsets. Such tight coupling, however, makes it correspondingly difficult to expand and alter the toolsets. In addition, it means that these tool integration mechanisms are tied to the particular tools and environments for which they have been developed. Make and Odin, on the other hand, are more general and reusable. They are not tied to any particular host environment.

The modeling capabilities of DSEE and System Modeler, moreover, seem to us to be inadequate. They, like Make, 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. Their analog of the Make system's "Makefile" is called the "System Model" in both. As a result, as in the case of Make, information about a particular tool must often be specified repeatedly in new System Models, a significant

burden for complex tools. Further, it requires the individual updating of all of these System Models when the interface to a tool is modified or when a new tool is to be incorporated into a project. In Odin, objects and tools are described separately. This enables a single tool expert to precisely specify the behavior and use of a given tool, saving each potential user the trouble of reproducing that specification every time the tool is to be used.

Odin's modeling language is also more powerful than those of the systems just described in its support for command parameters. When users specify parameters in their commands, Odin can tap the considerable flexibility built into underlying tools, yet analyze such parameterized commands accurately enough to correctly identify intermediate results of earlier parameterized commands for reuse.

Some of Odin's superior default capabilities have already been described. Although both Make and DSEE contain mechanisms for providing "default" rules for tool activation and object creation, the semantics of these rules are too simple to allow for the specification of complex tool interactions. In environments such as Toolpack, which must integrate a complex and fluctuating toolset, the need for more powerful modeling languages and default rules is critical.

Odin's object-modeling capabilities also support the specification of object aggregates—such as arrays and structures. One result is that Odin can manage the integration of tools, such as source text splitters, which produce an unpredictable number of objects as outputs. Earlier systems such as Make are unable to model such tools.

Odin can also adjust its object-management strategy dynamically, depending on the outcomes of tool executions. For example, in order to integrate "include processing," objects which "include" other objects must be processed by tools as if the text of included objects was part of the including object. In earlier systems such as Make, include processing requires the creation and explicit invocation of metatools. Odin integrates a special tool fragment that generates the list of objects included by an object, and the output of this tool determines the input dependencies of the object.

4. THE DESIGN OF ODIN

As noted above, Odin facilitates tool and object management by grouping objects into types and then specifying tool interactions in terms of the way the tools use and create instances of these types. Thus, the Odin design centers around two structures: the Derivation Graph, which specifies type and tool interconnections, and the Derivative Forest, which specifies how actual objects (instances of the types) are related to each other. This section describes these two structures, how they are used, and the languages that specify them.

4.1 Object Management in Odin

Odin encourages users to directly request needed software objects, without worrying about how the object might be created, whether it has been built before, or whether something similar might be used as a suitable starting point. Odin's job is to expedite the process of furnishing such requested objects.

Odin carefully names each object in its store so that the name accurately reflects the way the object either has been, or could be, derived from elementary

objects in the store by a sequence of tool fragments. This fully elaborated name then guides an efficient search of 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 representing some progress from the store's most elementary objects toward what has been requested. Odin then uses the fully elaborated name to prescribe the sequence of tool fragments that can take the already existing object(s) and derive them into what has been requested.

For example, if the user wants a prettyprinted version of source text “joe”, which had previously been prepared for a debugging session by an instrumentation tool, then the user would request the object:

```
joe:ins:fmt
```

namely, the object that results from starting with `joe`, then producing an instrumented (`ins`) version of it, and then producing from it a formatted (`fmt`) version. Although the user may view this as the sequential execution of two tools—an instrumenter and a formatter—each of these two is achieved 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 requested by the user is built by at least half a dozen tool fragments, most of which need not be understood by the user.

Odin builds this tool fragment execution sequence by first creating the internal name of the requested object and then searching for an expedited sequence of tool executions for building the object from what is currently in the store. The organization of the store expedites such searches. If the requested object is present, it is immediately returned to the user. If `joe` is present, but none of its derivatives are, then Odin constructs a procedure for deriving the requested object from `joe`. Along the way, 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 future reuse. If “`joe:ins`” is present in the store (e.g., the user may have previously requested the instrumented version of “`joe`”), Odin would use it as the basis for a simpler and faster derivation procedure, namely, executing only the lexical analyzer and formatter using `joe:ins` as input.

The Odin Derivative Forest is key to understanding this derivation process.

4.1.1 *The Odin Derivative Forest.* Odin's object store is structured into a forest—the Derivative Forest—which indicates how objects have been produced from each other. The object store contains two classes of objects—atomic and derived. Atomic objects are those that Odin cannot reproduce on its own. They have entered the store by text insertion or explicit importation from the host file system. Derived objects have been created by the action of Odin-integrated tool fragments. Each atomic object is the root of a tree in the derivative forest, where

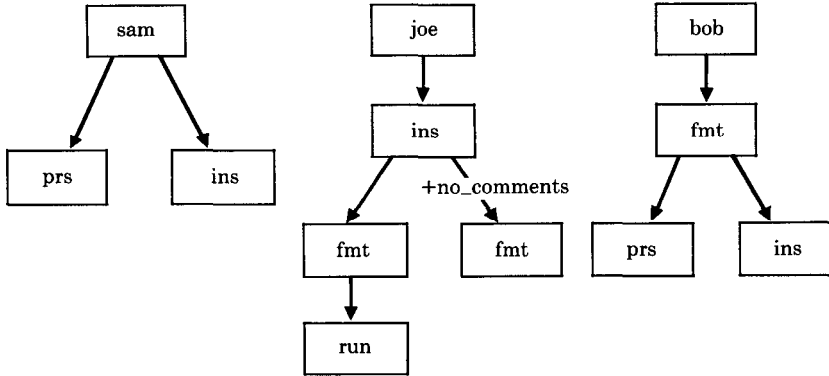


Fig. 1. An illustration of an example Odin Derivative Forest, showing both atomic and derived objects.

the tree consists of all objects derived, directly or indirectly, from the root by Odin-integrated tool fragments.

Figure 1 shows such a derivative forest, in which “sam”, “joe”, and “bob” are all atomic objects of type “source code”. The tree rooted by “sam” has two subtrees—one represents “sam”’s parse tree and the other represents an instrumented version of “sam”. The tree rooted by “bob” has one subtree whose root represents a formatted version of “bob”. It is, in turn, the root of a subtree that represents the parse tree and the instrumented version of the formatted version of the original “bob” source code.

The tree rooted at “joe” illustrates the use of parameterized object descriptions. The “ins” subtree of “joe” has two subsubtrees, each rooted at a different instrumented derivative of “joe”. Sometimes, additional information can be associated with an object to affect the derivatives produced from that object. In Odin, this additional information is captured as a “parameter” of that object, and is specified by appending a ‘+’ and the parameter. For example a “debug” parameter could cause the compilation derivative to contain runtime checks; a “library” parameter could cause the load derivative to have undefined externals satisfied from a nondefault library; and a “format” parameter could cause all printable derivatives to be generated in line-printer format. Figure 1 specifies default parameterization for one instrumented version of “joe”, and the explicit parameterization “no_comments” for the other, as follows:

```
joe +no_comments:ins
```

Values can be assigned to parameters, and these values may be other Odin objects.

4.1.2 Alteration of Objects. The need to change stored objects (e.g., by editing them) complicates the management of derived objects. When an atomic object (e.g., source text) is modified, Odin considers the result to be a new object, namely a new version of the original object. If the user specifies a name for this new object, it becomes the root of a (temporarily descendantless) derivative tree. If not, the new object automatically replaces the original object. In this case, it

is not safe to assume that objects derived from the original are correct derivatives of the new object.

Odin assigns a date and timestamp to each object in its store. Whenever a derived object is older than any of the ancestors in its derivative tree, it is treated with suspicion. For example, if the user requests “sam:ins” and Odin finds that “sam:ins” already exists, Odin first compares the time and date-stamps of “sam” and “sam:ins”. If “sam” is newer than “sam:ins”, a rederivation process is begun.

It is tempting to suggest that this comparison process be avoided by always deleting all derivatives of an atomic object that has been edited. Odin does not do this because some editing procedures create only superficial changes that do not alter many of the object’s derivatives. Odin incorporates difference analyzers to make this determination and to avoid costly and unnecessary rederivations.

In the example just given, for instance, editing “sam” may only alter comments. If so, the previous scan table, parse tree, symbol table, and instrumentation are still correct derivatives of the edited version of “sam”. Odin detects this and saves most of the work of rederivation. Once Odin recognizes that the present version of “sam” is new, it rederives the `scn_cmt` and `scn_tab` derivatives, compares the new derivatives to the old ones, and finds that the new `scn_cmt` object is different, but the new `scn_tab` object is not. Odin then replaces the old `scn_cmt` object with the new version, but merely updates the time and datestamp on the old `scn_tab` object, indicating that it is a correct derivative of the new “sam” object. Derivatives of “sam:scn_tab”, such as the `prs_sym` and `prs_nod` objects, are also correct derivatives of “sam”, and need not be rederived. Odin recognizes that “sam:ins” is also a correct derivative and updates its time and datestamp without rederiving it. Thus only the scanner 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.

4.1.2.1 Trustworthiness and Validity of Objects. Odin attaches to each object a status-level attribute indicating how much confidence the object should be accorded. The status level is an enumerated type whose values are OK, WARNING, ERROR, NOREAD, NOFILE, and ABORT. Odin assigns an ordering to these values, with OK being the highest and ABORT the lowest. The status level of an atomic object is always OK. The status level of a derived object is the least of the status levels of the objects derived by the tool fragments used to produce the object. An object’s status level is indicated whenever it is requested, unless the status level is OK. 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 abort status are the contents of the object

```
joe:run:err
```

When key objects have been changed and now have reduced status levels, Odin broadcasts this news to derived objects to which such changes are expected to be particularly significant. Such derivatives are said to be related to such key higher level objects by a “sentinel” relation. The existence of a sentinel relation between 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 create a network of constraints among the objects in Odin’s store, as the store is being built up. This network facilitates the early, effective, and effortless detection of changes to higher level objects which cause errors in key derivatives 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.txt”. If these two objects are marked as being “sentinels,” then every modification to “thesis.txt” will effect an automatic rederivation of both. If either rederivation causes an error or warning, the user will be notified. Assume that the “:spell” object is a list of spelling errors and has status ERROR if the list is nonempty, and assume the “:run” object creates an executable and has status ERROR if any error messages are generated in attempting to compile and run. If “thesis.txt” is modified, Odin will automatically rederive those objects, automatically checking that the “thesis.txt” object is spelled correctly and that it is acceptable to the “prog.c” program. Thus Odin enables the user to specify automatic checking to determine when certain changes have undesirable effects.

4.1.3 *Compound Objects*. Odin objects may be either simple or compound. Simple objects have no internal fine structure visible to Odin. Compound objects are lists or structures of Odin objects, and they arise in two ways—through the action of tools or through explicit construction by users.

4.1.3.1 *Tool-Generated Compound Objects*. A source text object consisting of more than one compilation unit is a compound object. It may arise as output from another tool, from editing, or it may be input from the host environment. Odin must initially treat such objects as simple objects, but can treat them as compound objects after the user invokes a special tool (e.g., a “splitter”) to detect the fine structure. These tools associate a key with each component of the compound object to enable users to extract the components.

For example, suppose “joe:output” is generated by executing “joe”, and is compound because joe’s execution creates multiple output streams. The tool that derives “:output” objects is expected to associate with each output stream a meaningful key, such as the filename associated with the stream, to enable users to access these component objects.

Users specify the components of a compound object by appending an at-sign (‘@’) and the key. For example, if “joe:output” consists of three components whose keys are “DATA”, “source.list”, and “source.errors”, then they are specified as

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

4.1.3.2 *User-Created Compound Objects.* Users create compound objects directly by using pointers—Odin objects of type “ref”. For example, in Figure 2, “sally” and “jane” are “ref” objects, where “sally” consists of pointers to the three objects: “sam”, “joe”, and “bob”, and “jane” consists of pointers to “bob” and “tim”.

A compound object can be an input to a tool, in which case the output is usually the compound object that is the composition of the objects produced by applying the tool to each of the components of the input object. For example, if the user requests

```
sally:ins
```

Odin produces a compound object consisting of pointers to

```
sam:ins   joe:ins   bob:ins
```

To do this, Odin assures that the “scn_tab”, “prs_sym”, and “prs_nod” objects are created for each of “sam”, “joe”, and “bob”. These are all stored as descendants of the “sam”, “joe”, and “bob” “f77” atomic objects. The object “sally:ins” is itself stored as a descendant in the derivative tree of “sally”. It consists of pointers to

```
sam:ins   joe:ins   bob:ins
```

which reside in their respective derivative trees. Odin recognizes if any or all of these individual “ins” objects had been created previously, and does not rederive them. Similarly, these derived objects are available for reuse in satisfying subsequent user requests made indirectly through other “ref” files. In the example shown in Figure 2, if the user requests the object “jane:ins” after requesting “sally:ins”, Odin simply sets a pointer to “bob:ins”, but will derive the “tim:ins” to satisfy the request. Thus, Odin manages overlapping aggregate objects without needless inefficient repetition of work.

Odin “ref” objects are also used to define object hierarchies. This enables the modeling of programs as hierarchies of procedures. At lower levels, procedures must be modeled as overlapping sets of support procedures and libraries. In constructing this hierarchy, the user need not be concerned with assuring that the constituent objects are mutually disjoint, as no inefficiencies in tool application result from this. Instead, the hierarchical structure is free to reflect the program’s logical structure.

Applying tools to such structures is easy and natural, as the user simply applies needed tools to the highest level “ref” object. Odin is often able to quickly satisfy such requests, because very few objects need be recreated. For example, when a program is being maintained, many of the objects which the user indirectly requests have probably recently been created and need not be rederived. In the usual maintenance scenario, determining which objects need to be recreated is painstaking and perilous. Most users avoid it and its risks by doing massive rederivations, often needlessly duplicating considerable previous work. Odin assures that only necessary rederivations will occur. Users simply issue one terse command at an appropriately high conceptual level.

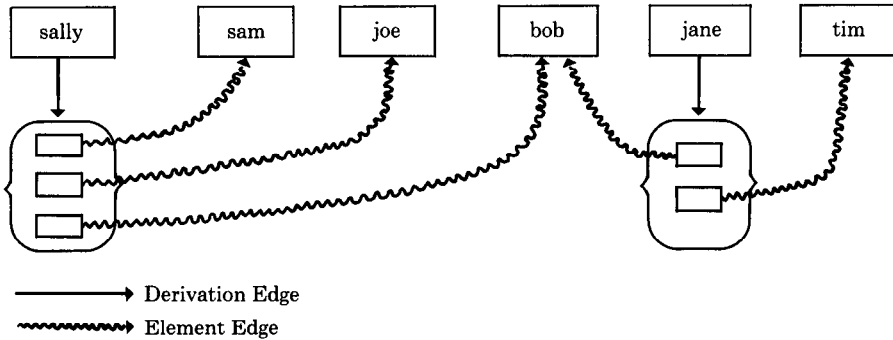


Fig. 2. An illustration of an example Compound Object, showing both derivation edges and element edges.

As the pointers in a compound object may point to objects of different types (even other compound objects), this construct may be used to build arbitrary graphs of objects.

5. THE ODIN SPECIFICATION LANGUAGE AND DERIVATION GRAPH

Odin also incorporates a specification language that enables tool integrators to specify how the various tool fragments and object types relate to each other. The language is designed to facilitate the integration of existing tools or sets of tools 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.

For example, a host-system compiler can be incorporated into an Odin-integrated environment by describing the input/output behavior of the compiler using the Odin specification language, but without having to alter the compiler. Odin itself provides a variety of internal tools to make this possible. For example, one Odin internal tool interprets an object containing a list of object names as a “collection of objects,” so that this collection of objects can be treated as a single object by a user of Odin. Odin ensures that a request to run a tool on this collection invokes the tool on each of the elements in the collection.

Another internal tool builds the Odin Derivation Graph. This tool uses the specifications of all tools and object types that are expressed in the Odin specification language to construct a single graph representing all of the relations among these tools and types.

Details of the Odin specification language are provided in [4]. A brief summary of salient features of this language is provided here.

All specifications of tools written in the Odin specification language consist of the name of the tool and a description of its input/output behavior. For example, a simple formatter is described as follows:

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

where “fmt” is the type of the result of applying a C code formatter, “formatted version of C code” describes this object, “pol_c.cmd” is the name of the formatting tool, and “c” names the type of object suitable as input to the formatter. Note that this tool specification also serves as a specification of the types with which the tool deals. Thus the totality of all tool specifications is also a specification of all of the types of objects integrated by Odin.

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.

5.1 Object Types

Every type of object that is to be made available for editing directly by the user is given a unique “atomic object type.” Every type of object that is produced by some computer program or tool is given a unique “derived object type.” A description of a derived object type consists of a description of the structure of the derived object followed by a description of the tool that produces the derived object and a description of the inputs needed by the tool.

5.1.1 Derived Object Structure. Due to the great variety in the output behavior of tools, it is necessary to provide a flexible language for describing the various possible kinds of derived object types. Examples of different kinds of outputs that a tool might generate 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.

5.1.1.1 Simple Derived Object. Some common simple object types are assembler code generated from a higher level language, executable binary, cross-reference listings, and error reports. A simple object type is analogous to a basic variable type in a programming language, such as Boolean, character, or integer. Odin allows a user to introduce an arbitrary number of such basic types.

An example of a simple type declaration for a linking loader is

```
exe “executable binary”:  
  USER ld.cmd  
  :c.o
```

In this example, the type “exe” is declared as a simple type, produced by the tool “ld.cmd” from input of type “c.o”.

5.1.1.2 Compound Derived Object. An object that is of type “compound derived object” consists of a set of objects, each of which is of the same object type, called the “element object type” or is another compound derived object of the given type. A compound object that contains only objects of the element object type is called a “flat compound object”—one that also contains other compound objects is called a “nested compound object.” A flat compound object is analogous to an array in a programming language—a nested compound object is analogous to a tree.

A tool must be specified as producing a compound object type when it produces an arbitrary number of objects of the same type, or when it produces references to an arbitrary number of objects.

An example of a compound type declaration is given by this specification for a tool that scans a C source code file for included files:

```
c_ref(h) "included files":
  USER c_ref.cmd
  :c
```

In this example, "c_ref" is declared as a compound type whose elements are of type "h".

5.1.1.3 Composite Derived Object. An object whose type is a "composite" derived object consists of a fixed number of objects whose types may be different. This is analogous to a record or structure type in a programming language. Odin considers most tools that produce multiple outputs to be tools that produce a single composite object as output. The members of a composite object type can be compound or simple object types.

An example of this is the following composite type declaration for a C compiler:

```
cc {
  c.o "object code produced by the C compiler"
  c.list "listing produced by the C compiler"
  } "C compiler output":
  USER cc.cmd
  :c
  :c_ref
```

In this example, the compiler produces two outputs: the object code and a listing file.

5.1.2 Inputs. More than one input object is often needed by a tool in order to produce its output. 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.

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

An example of a tool that allows such parameters as inputs is one that checks out a particular revision of a file from a file that contains a tree of revisions in a compact form. A possible Odin specification for such a tool is

```
c "checked out version of a C file":
  USER co.cmd
  :c,v
  :PARAMETERS (date revision who state)
```

In this example, "c,v" is the type of file expected as input, "date", "revision", "who", and "state" are the optional parameter types, and "c" is the type of file produced as output.

5.2 Casting

We consider the objects that Odin manages to be instances of types, and the tool fragments that Odin invokes to be operators on the type instances. Odin assures that tool fragment operands are always objects of the proper types. This involves casting inappropriately typed objects where necessary and advising the user when casting is impossible.

The focus of this typing and casting mechanism is the Odin Derivation Graph—the structure that, as noted earlier, is built from the tool specifications in order to record which tool fragments are currently incorporated, which types of objects they produce and require, and the way in which these various tool fragments can be synthesized and concatenated. The nodes of the Derivation Graph represent the types of the objects in the object store, and the edges represent ways in which new objects can be created from existing objects. New objects can be created either by casting or derivation, each of which is represented by a different type of Derivation Graph edge.

Casting is done when Odin determines that an 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. If the nodes representing these types are connected in the Derivation Graph by a “cast” edge, then Odin will retype the object to enable the needed tool fragment to use it. This does not alter the contents of the object. For example, in Toolpack/IST, inputs to the scanner must be of type “source text.” The outputs of the formatter (fmt) and the instrumenter (ins) are both source text, but are typed “fmt” and “ins”, so that these different objects can be distinguished. Objects of type “fmt” and “ins” are made legal inputs to the scanner by casting their types to “source text.” The Derivation Graph reflects this by including two cast arcs, one to “source text” from “fmt” and one to “source text” from “ins.”

The Derivation Graph also contains derivation arcs to represent how tools can derive new objects from existing ones. Each derivation arc represents a tool fragment which Odin can invoke to derive an object of the type indicated by its head node. The tail of a derivation arc represents the type of the object needed as input to this tool fragment. Several derivation arcs may be needed in order to completely characterize the behavior of a single tool fragment.

Figure 3 shows a part of the Derivation Graph that represents the tool fragments and object types managed by Odin to form Toolpack/IST. Cast relations are shown as dotted arcs and derivation relations are represented by solid arcs. We now use this figure to show how Odin creates the formatted version of the instrumented version of source text object “joe”. If the middle tree of the Derivative Forest of Figure 1 initially consists of the root only, namely atomic object “joe”, then when the user gives the command, Odin first examines the Derivative Tree to see if “joe:ins:fmt” has already been created. As “joe” has no descendants at all, Odin next determines that “joe:ins:fmt” cannot be built until “joe:ins” is built, and tries to construct “joe:ins”. Odin consults the Derivation Graph and sees 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). The Derivation Graph indicates that the parser produces both parser symbol tables and parse trees.

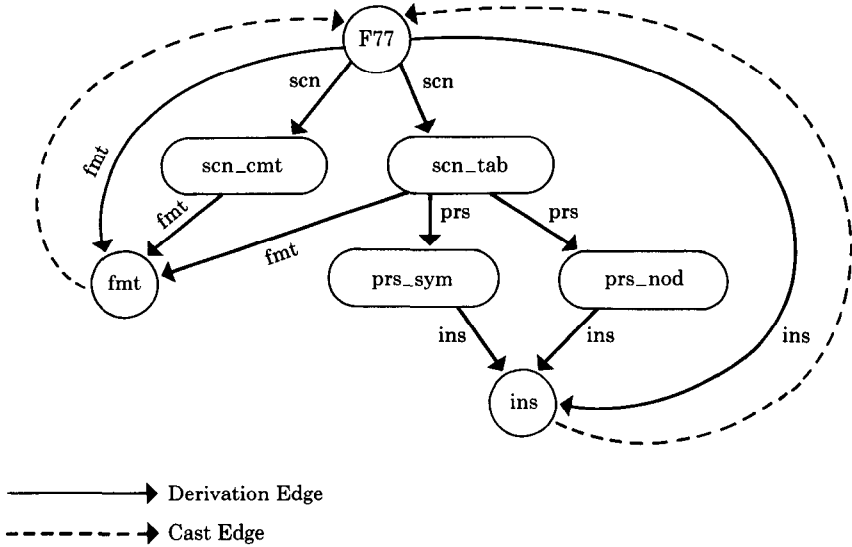


Fig. 3. An illustration of a portion of a typical Odin Dependency Graph, used to represent the relations between tool fragments and object types. This illustration is taken from the Dependency Graph used in integrating Toolpack/IST, and shows the use of both Derivation Edges and Cast Edges.

Odin next determines that the parser requires an object of type “scn_tab” (scanner table) as input. This object is not present either, and thus “joe:scn_tab” must also be built. The Derivation Graph shows 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 must be invoked before the parser. Odin sees that “joe” is an object of type “f77”, and now has determined that “joe:ins” can be built from existing objects by first invoking the scanner using “joe” as input, then invoking the parser using “joe:scn_tab” as input, and finally invoking the instrumenter using “joe”, “joe:prs_sym”, and “joe:prs_nod” as inputs.

Now Odin must create “joe:ins:fmt”, assuming that “joe:ins” and various intermediate objects are already created. Odin assumes that the Derivative Tree rooted at “joe” will appear as shown in Figure 4, and that the following objects will all be direct descendants of “joe”:

- joe:scn_cmt (comments embedded 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)

Odin must determine how the desired object, “joe:ins:fmt”, is to be created as a node of a subtree rooted at “joe:ins”. Odin examines the Derivation Graph and finds that an object of type “fmt” can be derived from an object of type “f77”, an object of type “scn_cmt” and an object of type “scn_tab” by using the formatting tool fragment. As “joe:ins” is not of any of these types, Odin must

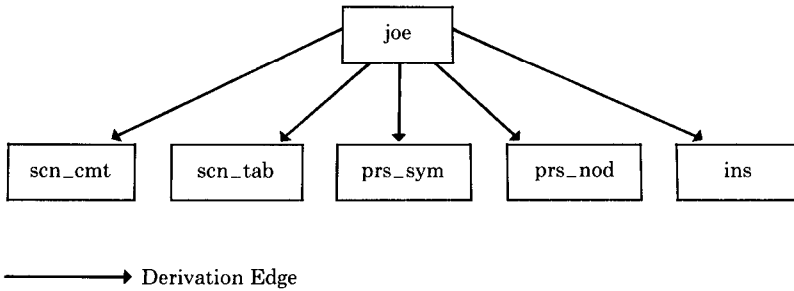


Fig. 4. An illustration of a typical Derivative Tree, showing derived objects of potential utility in further object derivation.

do some intermediate work. 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 a cast of “joe:ins”, to create “joe:ins:scn_tab” and “joe:ins:scn:cmt.” Those objects, along with “joe:ins” (cast to type “f77”), are sufficient inputs to enable the formatter to execute and produce the final object, “joe:ins:fmt”.

This 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, “joe:ins” would be a direct descendant of “joe”, and Odin would have invoked only the scanner and formatter to build the requested object.

5.3 Extensibility

The Derivation Graph is the basis for the extensibility of Odin-integrated toolsets. The Derivation Graph is accessed and maintained by the Odin command interpreter and is not accessible to the various tool fragments themselves. Tool fragments have no knowledge of the sequence in which they may be called. They never directly invoke each other. Odin invokes them assuring correct flows of needed objects between tool fragments. As a result, a tool fragment can always be replaced by another, provided that the replacement produces objects of the same types and draws upon objects of the same types created by the other tool fragments in the toolset. This enables considerable flexibility in replacing tool fragments, and makes it straightforward to integrate new tool fragments. New tool fragments are characterized by their input and output object types. Input object types must all already exist, but output types may be either partially or totally new. New output types must be represented by new nodes in the Derivation Graph. Once edges connecting the input types to the output types—new and old—have been inserted into the Derivation Graph, Odin has complete knowledge of when to invoke the new tool fragment and how to optimize the creation of objects that it produces.

Tool and object type specifications can be altered by use of a simple text editor. Thus altering or extending an Odin-integrated toolset entails making alterations to the Derivation Graph through text-editing changes made to specifications written in the Specification Language, running the Derivation Graph generator, and placing the new tool fragment(s) in an appropriately public location.

In cases where new or altered tools create new types of output objects, it is also desirable that complete accessing function clusters for the new types be made available. 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. Tool writers can treat objects of these new types as abstractions, leading to the flexibility needed in developing and migrating a prototype integrated toolset. Thus, new tool fragments that create new data types should 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 for manipulating the instances of the type, while hiding its internal implementation. 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.

5.4 Experiences with the Odin Specification Language

Although the Odin System has been in use for several years, both in research projects and in classroom assignments, during that time only a half dozen or so people have participated in the development of an Odin Derivation Graph. The reason for this is that, unlike other weaker system-modeling languages that require changes to the system model when the objects in the system change, an Odin Derivation Graph requires modification only when the tools change. Thus, only the tool builders become involved with the specification language.

After polling the small community of Odin tool builders, the predominant criticism is that the beginning tool builder needs more help from the Derivation Graph compiler (better error messages with hints as to how to proceed). On the positive side, once the first Derivation Graph is written, it is comparatively easy to modify. In addition, the flexibility and efficiency of the Odin interpreter provides important support for the production of a unified software environment from a disjoint set of software tools.

Our experience to date indicates that Odin is indeed a vehicle for readily extending and modifying integrated toolsets. We have succeeded in incrementally incorporating dozens of tools and tool fragments into Toolpack/IST—some of which we produced ourselves and some of which were captured from host environments. The current Toolpack/IST Derivation Graph contains over 60 user tools and tool fragments and 50 instances of Odin internal tools. New tool fragments have been incorporated into Toolpack/IST in as little as a few minutes. We have attempted to treat object types as data abstractions, but this has been largely voluntary, because many important Toolpack tools were captured from host environments and could not be altered to sharpen the boundaries between abstract data type accessing functions and functional tool capabilities. Toolpack/IST was used to maintain itself. This was a considerable job, as Toolpack/IST consists of 150,000 lines of source code, with 15 megabytes of derived data.

6. THE ODIN REQUEST LANGUAGE

The Odin Request Language is imperative and object oriented. It contains two basic commands, Display and Transfer, and a variety of utility commands. The

utility commands support such capabilities as command scripts, history maintenance, and system parameter manipulation. They are described fully in [4], but are not discussed here.

6.1 The Display Command

The display command prints an Odin object to the current standard output device, normally a terminal screen, and is implicitly invoked whenever an Odin object is requested. An object is requested by naming the root of its derivative tree and then appending the names of derivative tree nodes needed to unambiguously specify a path to the requested object. The appended nodes are separated from each other and the root by a colon (:).

For example, if “joe” is a root source text object, and the user requests

```
joe
```

Odin will print the source text object joe.

If the user requests

```
joe:fmt
joe:ins
```

Odin will print the objects derived from “joe” by the “fmt” tool and the “ins” tool (in Toolpack/IST these are the formatter and instrumenter, respectively). Further, the user can specify

```
joe:ins:fmt
```

in which case Odin will print the object derived by first instrumenting “joe” and then formatting the instrumenter output. The user can specify arbitrarily many successive derivations by Odin-integrated tool fragments in this way.

Clearly, these derivatives are created only after the Odin interpreter had previously built lexical analyses and parses of the various versions of “joe”. The user can also view these intermediate objects by specifying the keywords used to describe their types. For example, to see the token list produced by lexically analyzing “joe”, the user specifies

```
joe:scn_tab
```

Some very complex derivation processes are concealed behind the Odin command language. For example, if joe were an executable main program, then the user could specify

```
joe:rln
```

The keyword “rln” is the name of a tool fragment that executes “joe”, accepting input interactively from the user and displaying output interactively as well. Thus, “rln” indirectly effects the compilation, loading, and interactive execution of the program represented by the “joe” source text.

As described earlier, objects are parameterized by appending a plus sign (+) and parameters after the name of the object. Parameters can be names of atomic objects, or names of compound objects, such as a “ref” objects. Thus, supposing “sally” is a “ref” object pointing to “sam”, “joe”, and “bob”, then

```
sally:fmt
```

would specify the object pointing to the objects derived from “sam”, “joe”, and “bob” by the formatter, and

```
sally +newtopts:ins
```

would specify the object pointing to the objects derived from “sam”, “joe”, and “bob” by the instrumenter according to the specifications in “newtopts”. In this regard, it should be noted that

```
sally
```

is the specification of the “ref” object. Simply specifying it would *not* return the concatenation of the source text of “sam”, “joe”, and “bob”, but rather the ref file itself:

```
sam
joe
bob
```

To obtain the concatenated source texts for these files, the user would have to request the object created by the “cmpd” tool, an Odin unary operator defined on “ref” objects.

6.2 Basic Commands—The Transfer Command

The transfer command copies the contents of one Odin object into a second Odin object, which must be atomic. This is specified by appending to the name of the first object a right-angle-bracket (‘}’) and the name of the second object. For example,

```
joe } tom
```

copies the contents of “joe” into “tom”.

```
joe:run:err } joe.err
```

puts into “joe.err” a copy of the list of errors generated in attempting to run “joe”.

Odin objects can be given as input to host-system editing tools by appending to the object a right-angle-bracket (‘}’) and a colon (‘:’), and then specifying the host system command. For example,

```
joe } :vi
```

invokes the host system editor “vi” on the Odin object “joe”.

```
joe:run:err } :more
```

uses the “more” processor to display the errors obtained in running “joe”. In case the colon and host-system command name is omitted, a default host-system editor is invoked.

7. EXPERIENCES WITH ODIN

Odin has been used in a number of tool integration projects. In this section we describe two that seem exemplary.

7.1 The Toolpack Project

As noted earlier, much of the impetus for creating Odin came from the need to create a flexible, extensible tool-integration system for the Toolpack project.

Odin was successfully used to integrate tools ranging from simple text manipulators to complex data flow analyzers. The tools integrated were produced both at the University of Colorado and elsewhere. Some were provided as source code, others only in executable form. Some had very simple structure and interaction with the rest of the toolset, while others interfaced with a variety of other tools. Some tools were highly interactive, while others were batch oriented.

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

Toolpack/IST was originally designed and implemented as an environment supporting the backend phases of development and maintenance of FORTRAN programs. It has emerged as an environment capable of supporting other languages and development phases as well. In fact, its earliest use was in integrating host-system C tools in support of its own development. As FORTRAN tools were integrated by Odin, it incrementally became an environment for supporting C and FORTRAN backend development and maintenance. Frontend tools have not been integrated, but no obstacle to doing so appears to exist.

Much of the perceived power of Odin-integrated toolsets seems to arise from its object orientation. Odin orients users of even modest collections of tools toward identifying key object types, some of which may only be produced as the result of long and complex tool applications. Users seem to readily embrace the central importance of such key types, once they have vehicles for efficiently creating and studying instances of the types. For example, source-text modules are instances of one such key type, but the objects containing all errors arising from compiling and loading collections of procedures into an executable are instances of another key type—an error report type. Once all of a program's source code is assembled into a single Odin structure, users are able to readily make and evaluate changes through a small iterative loop, entailing the creation of instances of error-report type objects (through Odin's integration of the compiler and loader) and the use of a text editor to make changes to source-text type objects. After source text edits are made, Odin determines which source code procedures to recompile, invokes the loader (only if the new compilations change any object code), and assembles all error messages from both the compiler and loader into a single error-report object to present to the user. The net effect is a feeling of being able to operate easily and efficiently on objects of just the right conceptual level.

Odin executes as a subsystem of the host operating system. Thus, running under UNIX, the user enters Odin and carries out work by invoking Odin commands. Odin gets the user's work done by invoking functional tools, some of which may be UNIX tools. The difference between using Odin and using an ordinary UNIX system with Make is that Odin has the effect of creating a powerful conceptual layer. By staying within Odin, the user can create new

objects simply by naming them as instances of key types. By virtue of their being typed, these objects have useful attributes and fit into definable places in potentially complex object structures. The other objects in these structures are automatically created as well. The consistency of these objects and structures is automatically maintained in a very efficient manner. The net effect is a feeling of being able to program efficiently in a very high-level typed language. Operating systems command languages such as the UNIX shell also attempt to support this sort of “programming in the large.” Perhaps the single most salient programming capability that Odin supports, but operating systems languages fail to support (and which Make does not support either), is a satisfactory notion of object typing. Our experience with Toolpack/IST suggests that this is a very useful and powerful capability.

7.2 The Integration of the GAG Toolset

Odin was also used to integrate the GAG attribute grammar system [7], which had previously been integrated with command-language scripts. GAG takes an attribute grammar specification and an associated lexical analyzer, and produces the specified semantic analyzer. GAG was implemented as a large set of tools coordinated through operating system procedure files for management, control, intertool communication, and error reporting. As Odin contains facilities for supporting just these sorts of needs, the conversion to integrate GAG under Odin was not too difficult. The result was a version of GAG that ran under Odin, but that offered few execution-speed or space efficiencies. The Derivation Graph did not offer a clear model of the GAG toolset, although the user interface did seem cleaner as a consequence.

In a second phase of this experiment we scrutinized the GAG system structure to detect ways in which Odin’s philosophy and approaches might be applied to integrating GAG more efficiently without changing the existing GAG tool fragments.

The three basic approaches to improving efficiency via Odin object management are abstraction, partitioning, and parameterization. Abstraction (producing intermediate abstractions of source objects) and partitioning (splitting source objects into disjoint pieces) require extensive modification to the internal structure of a tool system, and thus were not appropriate to the GAG experiment. The third approach, parameterization, requires identifying which system parameters affect which tool fragments. Frequently, there are intermediate objects that are not affected by the specified parameters, and therefore can be reused in several different parameterized queries.

In the GAG system, flags and options are used to modify tool behavior. Odin allows each flag or option to be a distinct parameter type, and each tool to specify which parameters are of interest. In the original GAG system, all options to all tools were stored in a single file. This file was preprocessed to produce a “control file,” which was then passed to each succeeding tool fragment, which extracted option values of interest. Splitting the single control file into a set of individual parameters allows Odin to determine which derivatives are affected by a parameter change. Since many GAG options only affect later tool fragments, the specification of each option as a separate parameter type in the derivation graph

provided significant increases in reuse of objects. For example, if a user makes several requests that differ only in the options passed to the cross-reference tool, the Odin system reuses all analyses and simply reruns the cross-reference tool with the differing parameters. This parameterization provided the most significant benefits that were observed.

This experience with GAG helped to strengthen our opinion that it is useful and natural for software objects to be labeled by the activities that created them. Clearly, this labeling should include a specification of the tools that created them, but it should also reflect any adaptations made to these tool functions.

The effects of the optimization described above varied considerably—in some cases little speed improvement was noticed, but in other cases speedups of a factor of ten were measured. Savings of storage space are far harder to evaluate. Odin automatically rederives requested objects and automatically purges objects when it has no more cache storage. Thus the size of Odin's storage area can be specified by the user. Less storage obliges more recomputation and longer running times. Thus there can be no absolute claim 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 automatically support effective tool execution in varying amounts of space is probably its greatest contribution to storage efficiency. This enables users to make time versus space trade-off decisions and have them effectively supported by Odin.

7.3 Evaluations

These experiments have helped us evaluate various principles for integrating software tools into effective environments. Most directly, the Odin project was a vehicle for studying the use of an object-management system to integrate tools. Within that context, we chose to integrate smaller tool fragments, but to manage relatively large-grained objects. We chose to manage objects by organizing them using two relations—hierarchy and derivation. We chose to optimize for tool flexibility and extensibility by materializing the underlying structure of object types and tool fragments, making this structure essentially an object itself. In the following subsections we evaluate each of these architectural choices.

7.3.1 Object Orientation. Our experiences have shown that centering an environment around an object store is a very effective way to integrate an environment. In Toolpack/IST it made the functional capabilities of a wide variety of tools more easily and economically accessible to users. We found a number of situations where it was natural and easy to name a desired object that could only be constructed by a complex chain of tool applications. In GAG, we found that identifying the underlying objects to be managed led to more efficiency in applying tools. At the very least, this work has indicated that other systems that make tools the center of attention err in not giving at least equal attention to the objects managed by those tools.

7.3.2 Tool Granularity. In studying the object structure of environments, we realized that most tools manipulate a surprising variety of data objects. Considering the lifecycle of these objects led to a better understanding of the internal structure of many common tools and the idea that tools should be viewed as aggregates of tool fragments. In retrospect, this simply asserts that tools should be synthesized from commonly used modules. There should be no need to further justify this architectural decision.

7.3.3 Object Granularity. Odin manages objects that may be either large or small. This paper describes our experiments using Odin to manage large objects. We experimented briefly with Odin as a manager of integers using tool fragments that were simply arithmetic operators. The result was an amusing, though inefficient, computational system which did, however, underscore the high overhead costs that Odin incurs in order to manage objects. Thus, it seems most effective to use Odin to manage larger data objects that may be aggregates of smaller objects. Thus, for example, Toolpack/IST manages entire lexical strings rather than tokens, and entire symbol tables rather than individual symbols. This enables users to access the large-grained objects easily, but requires special-purpose tools to access the smaller objects. These tools present a nonuniform appearance to users and hide the relations among the smaller-grained objects. We believe that an object-management system to efficiently and effectively manage a broad spectrum of sizes of objects is needed. Odin is capable of doing this, but in its current implementation is unacceptably inefficient.

7.3.4 Organization of the Object Store. We developed an organizational structure for objects that was adequate to represent the relations we encountered 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, because the relational model is ill-suited to represent needed relations such as hierarchy. Further, we feared that this would invite modeling more relations than necessary, and would require excessive effort in propagating the effects of changes in objects.

Instead, we chose to use only hierarchy and derivation to organize the objects in our store. These two relations effectively supported the integration of the toolsets described in this paper. We believe that both are essential organizing agents in any environment object store. Hierarchy enables us to effectively manipulate and reason about complex software objects. 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. We believe that these two relations are a minimal subset of those needed in an environment object store. Others may be useful, especially in managing a store of objects of diverse sizes.

7.3.5 Flexibility and Extensibility. The Odin architecture effectively supports the need to alter and extend the functionality of an environment. The key to doing this is the isolation in a single structure—the Derivation Graph—of all information about how objects of various types are created from each other by tools. Tool fragments have no knowledge of how they relate to each other,

thereby avoiding the need to modify existing tools, to add new tools, or alter other existing tools.

The Odin Derivation Graph is best considered to be an object—namely, the object used to schedule the coordination of tool interaction and to modify an environment's tool and type structure. Odin facilitates these processes by providing a language with which to describe the Derivation Graph and tools to support its compilation, viewing, and alteration.

Our experiences with Odin strongly indicate the importance of materializing the type and tool interaction structure of an environment as an object.

8. FUTURE RESEARCH DIRECTIONS

We believe that the basic ideas underlying Odin are a sound basis for the integration of software environments. Having completed this work, however, we see ways in which these basic ideas can and should be extended.

8.1 Varying Grain Sizes

We believe that an environment must effectively manage objects whose sizes range from very large to very small. The problem of efficient management of the very large collection of very diverse objects in a software project is one which has not yet been effectively addressed. Further, the problem of developing a uniform interface that effectively furnishes access to these objects needs to be better addressed.

8.2 User Interface

This project has sharpened our appreciation of the problems of providing users suitable access to the resources managed by an environment. Our primary focus was on managing large-grained objects. We provided a clean and uniform command language to access these objects. We found, however, that most users spent most of their time dealing with the smaller objects contained in the larger objects. Access to these smaller objects was provided through viewer tools created for each of the various object types. These viewers were custom built and not standardized. As a result, users were confronted with a nonuniform interface to the smaller objects, with which they spent most of their time.

Clearly, viewer tools must be treated in the same way as other tools in a well-designed environment. They must be composed of smaller, modular tool fragments. We must identify a set of modular viewing tool fragments and demonstrate that they can be effectively composed into special-purpose viewers for the requisite wide range of environment objects.

In addition, it is clear that viewers must assist users in understanding the environment command-execution process itself. The help facilities provided by Odin are a primitive start in helping users understand what Odin does and how it works. Structures such as the Derivative Forest and the Derivation 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 users how it is being changed by command executions.

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 to 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.

REFERENCES

1. BOEHM, B. W., AND STANDISH, T. Software technology in the 1990s: Using an evolutionary paradigm. *Computer* 16, 11 (Nov. 1983).
2. BUXTON, J. N., AND STENNING, V. Requirements for Ada programming support environments. Stoneman, DOD, Feb. 1980.
3. CLEMM, G. M. Odin—An extensible software environment. Univ. of Colorado, Dept. of Computer Science Tech. Rep. CU-CS-262-84, 1984.
4. CLEMM, G. M. The Odin system—An object manager for extensible software environments. Ph.D. dissertation, Dept. of Computer Science, Univ. of Colorado at Boulder, CU-CS-314-86, 1986.
5. DEREMER, F., AND KRON, H. Programming-in-the-large versus programming-in-the-small. *IEEE Trans. Softw. Eng. SE-2*, 2 (June 1976), 80–86.
6. FELDMAN, S. I. Make—A program for maintaining computer programs. *Softw. Pract. Exper.* 9 (1979), 255–265.
7. KASTENS, U., HUTT, B., AND ZIMMERMAN, E. *GAG: A Practical Compiler Generator*. Springer Verlag, New York, 1982.
8. KERNIGHAN, B. W., AND RITCHIE, D. M. *The C Programming Language*. Prentice Hall, Englewood Cliffs, N.J., 1978.
9. LAMPSON, B., AND SCHMIDT, E. Practical use of a polymorphic applicative language. In *Proceedings of the 10th POPL Conference*, 1983.
10. LAMPSON, B., AND SCHMIDT, E. Organizing software in a distributed environment. *SIGPLAN Not.* 18 (June 1983).
11. LEBLANG, D. B., AND CHASE, R. P. Computer aided software engineering in a distributed workstation environment. In *SIGPLAN/SIGSOFT Symposium on Practical Software Development Environments* (Pittsburgh, Pa., April 1984). ACM, New York, 1984.
12. LEBLANG, D. B., AND MCLEAN, G. D. Configuration management for large scale software development efforts. In *Proceedings of the Workshop on Software Engineering for Programming-in-the-Large* (Harwichport, Mass., June 1985).
13. LEBLANG, D. B., CHASE, R. P., AND MCLEAN, G. D. The DOMAIN software engineering environment for large scale software development efforts. In *Proceedings of the IEEE Conference on Workstations* (San Jose, Calif., Nov. 1985). IEEE, New York, 1985.
14. OLENDER, K., AND OSTERWEIL, L. J. Specification and static evaluation of sequencing constraints in software. In *Workshop on Software Testing* (Banff, Canada, July 1986), pp. 14–22, and Univ. of Colorado Dept. of Computer Science Tech. Rep. CU-CS-334-86.
15. OSTERWEIL, L. J. Software environment research directions for the next five years. *Computer* 14 (April 1981), 35–43.
16. OSTERWEIL, L. J. Toolpack—An experimental software development environment research project. *IEEE Trans. Softw. Eng. SE-9* (Nov. 1983), 673–685.
17. RIDDLE, W. E. The evolutionary approach to building the Joseph software development environment. In *Proceedings IEEE Softfair—Software Development Tools, Techniques, and Alternatives* (Crystal City, Va., July 1983). IEEE, New York, 1983, pp. 317–325.
18. SCHMIDT, E. E. Controlling large software development in a distributed environment. Ph.D. dissertation, Computer Science Div., EECS Dept., Univ. of California, Berkeley, Dec. 1982.

Received August 1986; revised September 1987, July 1989; accepted July 1989