# The GANDALF Project*

**David Notkin**
*University of Washington*

The GANDALF project is concerned with the automated generation of software development environments. In particular, the project has considered project management environments, system version control environments, and incremental programming environments. The artifacts surrounding these environments are described. Later versions of these environments have been constructed as structure editors. The processes and tools involved in generating structure editors for software development environments are also discussed. Future plans of the project are briefly mentioned.

## 1. INTRODUCTION

*Software development environments* provide automated software support that helps ease the software development process. This broad definition comprises at least two major areas of research. *Programming environments* are software development environments that reduce the challenges presented by the programming process. Language-oriented editors and source-level debuggers are two examples of programming environment tools. *System development environments* are software development environments that help reduce the challenges presented by the complexities of time and scale. It is well-known that the problems that arise when multiple programmers work, over time, on a system consisting of many modules are qualitatively different from common programming problems. Traditionally, mechanisms dependent largely on the good will of project members have been used to address these system-level problems. System development tools place such support -- for instance, in the form of version control descriptions or module histories -- into software and hence reduce the degree of reliance on the good will of programmers.

Project management, version control, and incremental programming are three specific areas of the software development process that the GANDALF project is interested in integrating into a single environment. This paper discusses the systems that we have developed to investigate these areas, including stand-alone and integrated systems. However, we understand that a single environment, because of differing and ever-changing needs, is not sufficient to meet the needs of all software designers and implementors (including ourselves). In order to reduce the cost of developing a variety of related integrated software development environments, we have become interested in methods for environment generation, which are also discussed in Section 2. The GANDALF project is uncommon in its concern with producing, through semi-automatic generation, software development environments that integrate both programming and system development environments.

## 2. A HISTORY OF GANDALF

Providing a purely chronological history of GANDALF would be both difficult and confusing. Instead we focus the history on the major ideas and artifacts of the project. This history aids in understanding the contributions, status, and directions of GANDALF. Figure 2-1 lays out many of the systems developed as part of the project.

[1] Perhaps the most common question raised about the GANDALF project is "Why is it called GANDALF?" Nico Habermann named the project GANDALF after the benevolent wizard in J.R.R. Tolkien's fantasy series *The Lord of the Rings*. GANDALF watched over his charges in a very passive way, intervening only when there was no alternative. He was there when he was needed but was not around, as a rule, otherwise. This attitude, we hope, has carried over into the GANDALF project, with the environment playing GANDALF while the software developers are the ones he watched over.

* Research on GANDALF is supported in part by the Software Engineering Division of CENTACS/CORADCOM, Fort Monmouth, NJ.

*Address Correspondence to Dr. David Notkin, Department of Computer Science, FR-35, University of Washington, Seattle, WA 98195*

0164-1212/85/$3.30

## 2.1. Project Management -- SDC

*Project management* in GANDALF is based on the belief that a developing software project should be viewed as a set of abstract data types upon which only certain operations may be applied. By carefully defining the types and operations, one can ensure that the software representing a project under development is always in a well-defined state. The System (or Software) Development Control system [1], better known as SDC, was the first artifact of this research. SDC is a set of programs, written first in the UNIX[1] Shell and later in C, that provide some basic management and communication support.

SDC defines four basic objects. A *project* consists of a set of related *source files.* A source file represents a small amount of related code or data -- for example, a procedure or a type definition. A *log file* is associated with each project and contains historical descriptions of modifications of the source files in the project. An *access control list* is also associated with each project.

Most SDC commands are applied to one or more source files in a project. The most commonly executed commands are *reserve* and *deposit.* A user must reserve a set of source files before making a modification. The effect of reserve is to lock those files and permit the user to modify them. Modifications of the actual files are done using UNIX commands. When the changes are completed, the user deposits the source files back to the project. This backs-up the old version of the files, unlocks the new version of the files, and prompts the user for a message describing the modifications. If, after checking the reserved files -- using standard testing, debugging, or verification techniques -- the user decides that the changes should not be made, the *release* command can be applied to unlock the files without any modification to the project. Every reserve command must ultimately be followed by either a deposit or a release (deposit is by far the more common case).

The log file contains one entry for every deposit performed on a project. Additionally, project members may explicitly add entries that augment the documentation of the system. SDC automatically places the user name and the time of modification in an entry at the time of deposit. The contents of the entry are up to the user performing the deposit. Our belief is that, if it is made easy, users are willing to write intelligible comments.

SDC supports three levels of users -- readers, who have only read access to the data in the project files; project programmers, who have the right to create, modify, and retrieve source and log files; and project leaders, who have the right to expunge data and to modify access control lists, in addition to having all project programmer rights. The commands described above, along with a variety of commands provided to manipulate log files, access control lists, backups, are restricted to users with appropriate rights.

Design, development, and maintenance of SDC was started in 1976 by Nico Habermann and was later continued by Barbara Denny and the author. The latest version consists of nearly 3500 lines of C code representing about 150 functions, objects, and types.

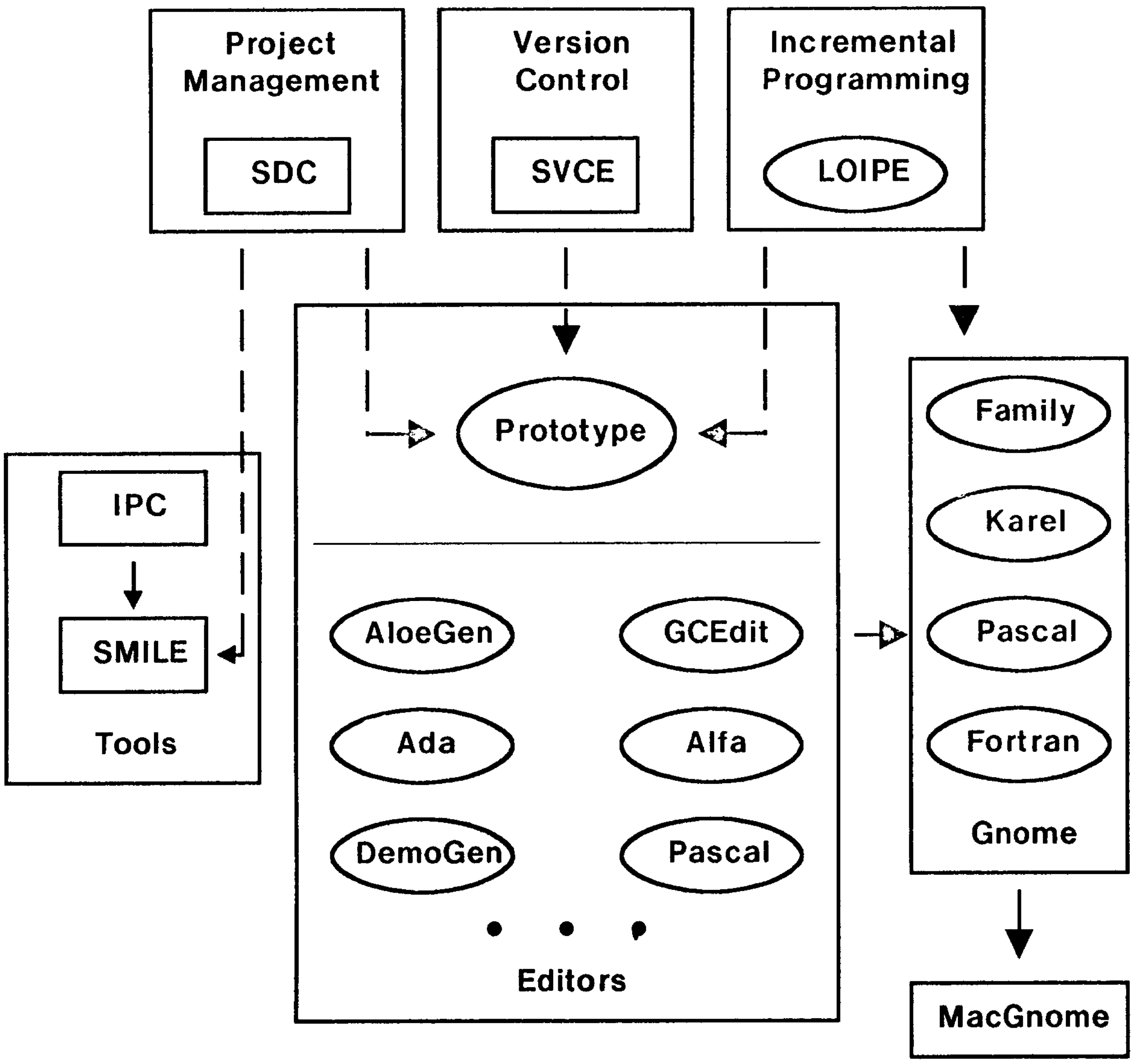
## 2.2. System Version Control -- SVCE

Concurrent with the initial project management work was the start of GANDALF's research in system version control environments. Cooprider's thesis [2] was an outgrowth of the FAMOS project [3], which investigated the potential for creating families of operating systems that shared subsystems. Cooprider developed a notation for describing interconnections among subsystems, for describing different versions of subsystems (for instance, the source code, the object code, and the documentation), and for describing the actual mechanisms (such as compiling and linking) that are needed to construct systems. The explicit nature of the descriptions is perhaps the most important characteristic of Cooprider's work.

Tichy's research [4] followed Cooprider's. Tichy defined the Intercol language as a descriptive mechanism to describe the interconnections and multiple versions of modules in a system. As a departure from Cooprider's system, Tichy specified three kinds of versions -- *parallel* versions, which define different implementations of the same specifications; *successive* versions, which define the evolutionary process of a module as bugs are fixed and features are added; and *derived* versions, which are different instantiations of a system based on the same source code (optimized code is one example of a derived version). Further, Tichy built into the system the knowledge of how to construct the system from its parts, rather than requiring individuals users to tell the system how the construction process must proceed.

Following Tichy's work on Intercol, Habermann, Dewayne Perry, and later Gail Kaiser continued research in the area that was by 1980 being called SVCE -- for System Version Control Environment. One of the primary contributions of this phase of the research was a distinction between implementations

[1]UNIX is a trademark of ATT Bell Laboratories.



Key: All small rectangles represent traditional systems (written in C) while all ovals represent syntax-directed editing environments.

The three basic pieces of the software development process -- project management, version control, and incremental programming -- and their primary artifacts are displayed at the top. The center box lists just a handful on the many syntax-directed editors developed as part of GANDALF. The most important editor, in terms of the full-fledged goals of the project, is the *Prototype* system, described in Section 2.5, that combines the notions of software development environments into a generated syntax-directed editor. Another key editor is ALOEGEN, which is used to generate language descriptions for other editors, thus demonstrating the bootstrapping capabilities of GANDALF. The GNOME project consists of four editors that are used by over 700 students a semester at Carnegie-Mellon University. The GNOME staff is now producing a similar PASCAL environment, called MACGNOME, for the MacIntosh. The box on the left represents some internal development tools used extensively by GANDALF project members.

**Figure 2-1:** A History of GANDALF

-- that is, versions that actually provide code to implement a set of specifications -- and compositions -- that is, versions that define new subsystems in terms of groups of existing subsystems. Within this framework, formal definitions of well-formed systems were defined; for instance, a system must provide -- either directly or through importation of another system -- an implementation for each specification that it exports. The benefit of providing compositions as well as implementations is that compositions ease the definition of new modules that are constructed in part or in entirety from existing modules. The version control work is dependent on the seminal work of DeRemer and Kron [5]. Explicit modular interdependency descriptions facilitate the construction of system development tools. Ada[2] [6] depends on similar descriptions -- the public parts of packages -- but enforces a restriction of zero or one implementations of a specification. The Cedar system's [7] system modeling work [8, 9] focuses on the problems of version control in a distributed environment.

The SCCS [10] and RCS [11] systems provide a combination of limited project management and version control support. Source files are checked-out and checked-in in ways similar to reserve and deposit in SDC. The Unix **make** program [12] is often used in conjunction with SCCS and RCS. **make** controls the construction of systems through use of a file describing both the dependencies in the system (for instance, that a given source file includes a specific set of macros) and also the actual construction process (for example, that a given object file is produced by applying the C compiler to a specific source file).

## 2.3. Incremental Programming -- LOIPE

Concurrent with the middle stages of the work on project management and system version control was the initial investigation of incremental programming environments. The goal of this piece of research was to develop techniques and approaches that permit development of LISP-like environments for compilation-based languages. Peter Feiler developed the LOIPE system (Language-Oriented Incremental Programming Environment) to explore these issues [13, 14].

LOIPE supports incremental compilation, linking, and loading for the language GC, or GANDALF C -- a minor variation of C that supports inter-procedure type checking of parameters. Incremental compilation takes place at the grain size of procedures (in contrast to the work of Fritzson [15], for instance, that supports compilation at the statement level). Whether recompilation can be limited to the procedure that has been modified depends on the type of modification. When minor changes to local variables and local control flow are made, only the modified procedure need be recompiled. If changes to the procedures specifications are made, then the procedures depending on the modified procedure must also be recompiled. The **make** program differs from this approach as it cannot distinguish among different types of modifications.

Incremental linking and loading relies on the use of indirect procedure calls. When a procedure call statement is encountered in a procedure, the incremental compiler for LOIPE places an index into an entry vector in the object code. The value of the indexed entry in turn provides the actual location of the called procedure. So, when a procedure is later modified, the linker and loader need only update the entry vector to contain the location of the new code. All further procedure calls will execute the new procedure rather than the old. Currently, all newly compiled procedures are loaded at the end of the address space and no compaction takes place.

LOIPE also supports source-level debugging. GC has been modified to add *break* and *trace* statements that LOIPE understands. Single statements or entire procedures may be traced or may cause breakpoints to be executed. Breakpoints and tracepoints are set by simply modifying the source code. Debugging at the source level relieves the programmer from learning a completely separate language and environment for debugging, as the basic language support is available. Monitoring of variables is also supported. The names of variables to be monitored are listed and their current values are continuously displayed. Given the combination of incremental compilation and the local nature of adding or deleting debugging information, LOIPE implements source level debugging in a cost-effective way.

The user interface to LOIPE is based on the ALOE syntax-directed editing system described next.

## 2.4. Syntax-Directed Editing -- ALOE

Raul Medina-Mora's syntax-directed editing and editor generation system is called ALOE [16, 17], which stands for A Language Oriented Editor. The initial and primary goal of ALOE was to investigate the feasibility of producing and using syntax-directed editors as the basis for programming environments.

[2]Ada is a trademark of the US Department of Defense.

Our interest in syntax-directed editors comes from several major beliefs.

Initially programmers conceive of a program as structure; then they transform their mental picture of structure into text; and finally a parser transforms the text back into structure. We believe that the user benefits greatly when they are relieved from the first transformation of structure into text. This has the added benefit of allowing us to eliminate, either totally or partially, the need for parsing as the user develops ALOE trees directly.

Our editors further provide a good mechanism for replacing the traditional {edit, compile, link, debug} cycle with a more natural {edit, execute} cycle; indeed, the LOIPE system just described is based on this simple tools cycle. We believe that this is just one example of situations where traditional mechanisms can be replaced by mechanisms that are more suitable to users.

ALOE's focus on the generation process has led to some confusion about terminology. For the record, *ALOE* refers to the common kernel that is used by all of syntax-directed editors generated through this process. The kernel includes support for the language-independent commands as well as library support, called *ALOELIB*, that can be used to add semantic actions and additional commands to an editor. On the other hand, an *ALOE editor* is a specific instantiation of a syntax-directed editor. Often, these ALOE editors are given specific nicknames that more clearly define the intentions of the system. For instance, ALOEGEN is an ALOE editor that is used to create the syntactic descriptions of a language for which an implementor wants to generate an editor.

All ALOE editors share a tree-oriented full-screen user interface. A set of language-independent commands -- such as subtree deletion and cursor motion -- are available in all editors. Language-specific operators that represent language structures -- for example, PASCAL WHILE statements or ADA PACKAGES -- are defined separately for each particular editor, as are static semantic checks that are language-dependent (type checking is a classic example). A set of extended commands, supporting explicit invocation of actions such as "execute-program" may also be added to specific editors.

To construct an editor for a particular language, an implementor must define the syntax, the static semantics, and the set of extended commands.

The description[3] for a language consists of two major parts -- syntax and semantics. The generation process is pictured in Figure 2-3. The syntactic definition of an ALOE editor divides into two parts:

The *abstract syntax* description defines the underlying structures -- in ALOE these structures are trees -- of the language. The abstract syntax defines both a set of *operators*, which represent node types that can appear in the tree, and a set of *classes* that indicate the ways in which these operators can be composed. It may be helpful to think of operators as commands and classes as menus that list the operators, or commands, that may be legally applied. This two-level description mechanism permits generation of a user interface that restricts manipulations to those that retain the syntactic consistency of language trees. The abstract syntax is a grammar that formally defines a set of abstract syntax trees that represent the legal syntactic programs in the base language.

The *concrete syntax* describes the mapping from the abstract syntax to a representation suitable for display to the user. The concrete syntax defines, for display purposes only, the syntactic sugar that has traditionally been needed in order to support parsing of text to structure.

The semantic support, which includes checking of static semantics and also definition of extended commands, comes in two flavors:

Editor implementors optionally define *action routines*, which are procedures that are invoked as a user makes changes to a tree. Each operator, or node type, in the language may have an associated action routine. When a node of that type is created, deleted, or modified, the action routine for the operator is invoked with the actual node and the type of the action (for instance, create or delete) as parameters. The invocation of action routines is done implicitly as users perform general manipulations.

An implementor may wish to provide semantic actions that are to be invoked explicitly. An *extended command* is defined by a procedure that is associated with a command name. Common extended commands include ones that invoke execution of a program or reserve a module.

The ALOE kernel, including ALOELIB, consists of about 1700 procedures, types, and objects, represented by 33,000 lines of code. Sizes of typical ALOE editors are given later.

[3]These descriptions are now developed using ALOEGEN but were initially developed using the *ALOE preprocessor* that took text descriptions of an ALOE grammar and generated the desired tables.

**Language Constructs (operators):**

```
        PROGRAM     = <stmt>          | "@0@n"
        PRINT       = string          | "PRINT '@1'"
        IF          = exp prog prog   | "IF @1 THEN@+@n@2@-@nELSE@+@n@3@-@n"
        AND         = exp exp         | "@1 AND @2"
        STRING      = {string}        | "@c"
        TRUE        = {static}        | "true"
        FALSE       = {static}        | "false"
```

The language construct description consist of three parts. The first part is simply the name of the language construct. The user enters the name in order to construct a node of that type. The second part defines the abstract syntax of the construct. Three cases are shown in the example. Lists are designated by the surrounding brackets ("<" and ">"). For instance, the PROGRAM construct consists of a (possibly empty) list of stmts. The structures surrounded by braces ("{" and "}") represent terminal nodes of the tree. In the case of STRING, a string constant is indicated. The other case represents nodes with a fixed number of offspring. The IF construct, for example, has three children, the first of class exp and the other two of class prog. The final part of the construct definition defines the unparsing scheme (the mapping to the display representation). Unparsing schemes vary according to their associated abstract syntax. Special unparsing directions are given by commands preceded by the "@" character. A new line is displayed when "@n" is encountered. Indenting levels are altered by the "@+" and "@-" commands. Lists are displayed by "@0" followed by the list separator. The N$^{th}$ child of a node with a fixed number of children is displayed when an "@N" is unparsed. Constants are displayed by "@c" commands. All text not prefixed by "@" is displayed as is.

**Classes (menus):**

```
        stmt        = PRINT IF
        prog        = PROGRAM
        string      = STRING
        exp         = AND TRUE FALSE
```

The description of classes has two parts. The first part defines the name of the class. The class names are referenced in language constructs for operators that are lists or that have a fixed number of offspring. The second part of the class is a list of construct names. This list indicates which constructs may replace appearances of the class name during program construction. The exp class definition indicates that an expression can be replaced by the AND, TRUE, or FALSE constructs.

Figure 2-2: Description for Sample Grammar

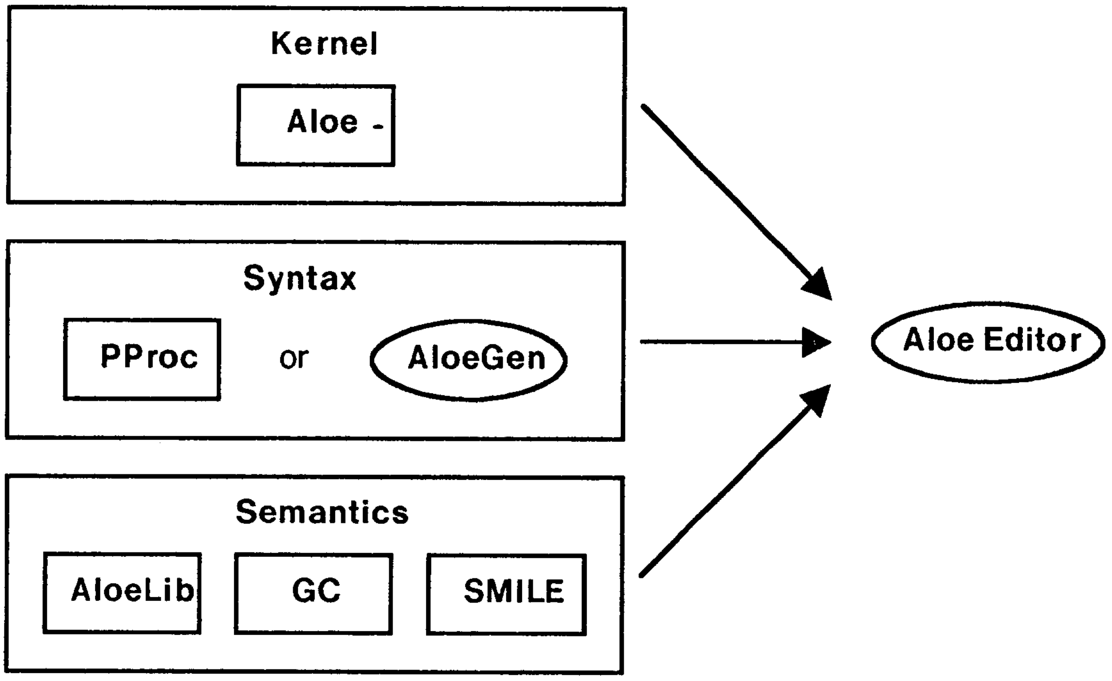
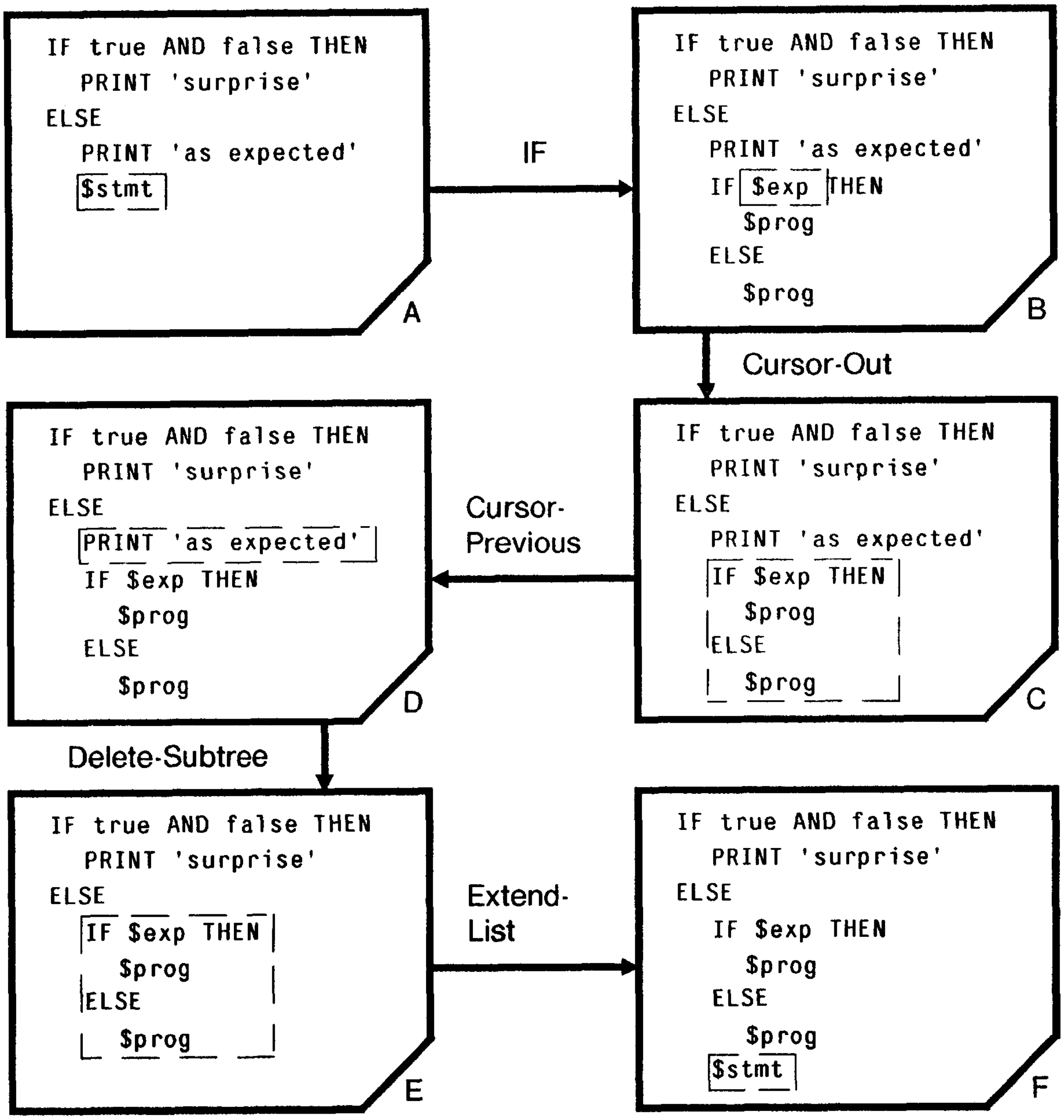


Figure 2-3: Constructing an ALOE Editor



```
IF true AND false THEN
   PRINT 'surprise'
ELSE
   PRINT 'as expected'
   $stmt
```

```
IF true AND false THEN
   PRINT 'surprise'
ELSE
   PRINT 'as expected'
   IF $exp THEN
      $prog
   ELSE
      $prog
```

```
IF true AND false THEN
   PRINT 'surprise'
ELSE
   PRINT 'as expected'
   IF $exp THEN
      $prog
   ELSE
      $prog
```

```
IF true AND false THEN
   PRINT 'surprise'
ELSE
   PRINT 'as expected'
   IF $exp THEN
      $prog
   ELSE
      $prog
```

```
IF true AND false THEN
   PRINT 'surprise'
ELSE
   IF $exp THEN
      $prog
   ELSE
      $prog
```

```
IF true AND false THEN
   PRINT 'surprise'
ELSE
   IF $exp THEN
      $prog
   ELSE
      $prog
   $stmt
```

The cursor, representing the current node of the tree, is displayed in dashed boxes. Unfilled-in nodes, called *meta-nodes*, are displayed as $CLASS where CLASS defines the language constructs that may replace the node.

Screen A shows an existing program. In screen B, an IF node replaces $stmt as the cursor moves to the first new meta-node. Applying cursor-out increases the focus of attention as shown in screen C. In screen D the cursor moves to the previous statement in the list. Deleting the PRINT statement yields screen E. Screen F shows the state after the list of statements is extended.

**Figure 2-4:** Sample ALOE Session

To clarify these ideas, the abstract syntax, along with associated unparsing information, for a tiny programming language with two statement types in shown in Figure 2-2. A sample editing session for the language is shown in Figure 2-4.

Other early syntax-directed editing systems, such as the Cornell Program Synthesizer [18, 19] and Mentor [20, 21], have investigated ideas similar to these. The Program Synthesizer supports editing, execution, and debugging of a subset of PL/I. Expressions and assignment statements are entered as text and are parsed. The original Mentor editor is a single-language editor for PASCAL. Programs entered as text are parsed with later actions occurring through structure-oriented manipulations. Both these projects have moved on to consider the generation of editors, based on the Cornell Synthesizer Generator [22] and the system surrounding the Metal meta-language description formalism [23]. More recent projects in this area include Pecan [24], Poe [25], Syned [26], and Magpie [27].

While we have, in most cases, constructed editors in which expressions are manipulated as structure, it is not true that "in GANDALF you have to edit expressions as structure." Given that GANDALF is, at its heart, a generator of editors, such a statement is patently false. Indeed, it is straightforward to construct an ALOE editor in which expressions (or indeed, even statements or entire programs) are text, which is parsed and analyzed in traditional ways. We have chosen to investigate the advantages and disadvantages of structure for expressions, but we have by no means fixed that choice in concrete for all our editors. In our experience with the students using GNOME PASCAL, which does consider expressions as structure, the students have not had significant problems.

## 2.5. The GANDALF Prototype

Our goal in constructing the GANDALF Prototype, or GP, was to construct a software development environment based on ALOE generated editors. The motivation for merging these ideas grew from the the need to integrate the system development tools -- such as SDC and SVCE -- into a single system. As an integrated system should be based both on a single implementation model and also on a uniform user interface, the desirability of constructing GP using ALOE was clear.

In creating GP, we constructed what may be the largest existing syntax-directed editing environment. GP merges the notions of SDC, SVCE, and LOIPE into an integrated environment that provides a syntax-directed interface to its users. The functions that GP provides include:

project management support, including reserve, deposit, and log creation;

system version control support, including parallel and successive versions, along with automatic system generation; and

programming support, including incremental compilation and source-level debugging.

In terms of function, GP is similar to other well-known development environments such as Arcturus [28] (an Ada-specific, full-life cycle prototype environment), Toolpack [29] (an environment characterized by its focus on composing large sets of small tools), and Interlisp [30]. The uniform syntax-directed interface and the generic nature of GP are among the primary characteristics that differentiate it from these related systems.

GP consists of almost 800 items (procedures, objects, and types) and roughly 17,000 lines of GC code. The grammar for the prototype has 191 operators (fifty-four terminals operators and 137 non-terminal operators) and eighty-two classes. It was initially introduced in July of 1981 and was enhanced and modified during the following year. Extensive modifications made to the ALOE kernel have caused the existing GP code to be out-of-date, and we have decided not to make the necessary updates to it.

In an earlier paper [31] we made the distinction between the *GANDALF System* and *GANDALF Environments*. The GANDALF System represents the underlying support system for generating environments similar to GP. The GANDALF System is composed of ALOE, ALOELIB, ALOEGEN, and a set of tools, procedures, and techniques that support construction of ALOE editors. GANDALF Environments, on the other hand, are the actual ALOE editors generated by the GANDALF System (including GP, of course). So, GANDALF Environments are ALOE editors that specifically address the issues of software development. Another way of looking at it is that the GANDALF System is a superset of ALOE (including ALOELIB and ALOEGEN) and that the set of GANDALF Environments are a subset of the set of ALOE editors.

## 2.6. GNOME -- Educational Environments

The GNOME project [32] is a spin-off of the GANDALF project that is concerned with the use of syntax-directed, incremental programming environments in educational situations. The project has developed four ALOE editors. The first editor used in the course, which is taken by over 1500 undergraduates a year at Carnegie-Mellon University, is a simple system that permits students to manipulate family trees. After becoming comfortable with the basic interface, the students are moved to the ALOE editor for KAREL [33], which is a turtle-graphics language that supports definition of simple programs. ALOE editors for PASCAL and Fortran are used in the remainder of the course. The PASCAL editor is the GANDALF equivalent of the Cornell Program Synthesizer and original Mentor editors, but as it is a generated environment it has been easier to enhance over time.

## 2.7. Tools

In developing the systems just described, we frequently found that our existing programming environment -- largely the environment provided for UNIX -- was insufficient to meet our needs. So, as do all good software designers and implementors, we ended up constructing some tools to help us along. These tools have been intended to be extremely practical, but they are also used to investigate and demonstrate some of the underlying concepts of the GANDALF project itself.

As Feiler began to write code for the LOIPE system described above, he decided that the current software development tools available on UNIX were not sufficient for his needs. He designed and built the IPC system that supports several useful mechanisms:

- Definition of a one-level module structure for software projects;
- Definition of a set of procedures, types, and objects within each module;
- Importation and exportation of individual items (procedures, types, or objects) from and to specific modules;
- Distinction between modification of item specification and item implementation in order to permit automatic recompilation of dependent program parts;
- An interactive user interface that helps manage modifications and queries about the database.

| Editor | Terminals | Non-Terminals | Classes |
|---|---|---|---|
| GANDALF-Prototype | 54 | 137 | 82 |
| ALOEGEN | 29 | 17 | 37 |
| GNOME-Family | 1 | 5 | 5 |
| GNOME-Karel | 28 | 9 | 9 |
| GNOME-Fortran | 37 | 67 | 64 |
| GNOME- PASCAL | 20 | 103 | 73 |

For each grammar, the number of terminal operators, the number of non-terminal operators, and the number of classes are given. The sum of the terminal and non-terminal operators gives the number of language-dependent actions that are provided to the user (with the exception of any extended commands, which are usually limited in number).

**Figure 2-5:** ALOE Editor Grammar Sizes

| System | Source Lines | Items |
|---|---|---|
| ALOE (kernel) | 33,000 | 1700 |
| SDC | 3500 | 150 |
| SMILE | 14,500 | 600 |
| GANDALF Prototype | 17,000 | 800 |
| GNOME-Family | 500 | 25 |
| GNOME-Karel | 2300 | 150 |
| GNOME-Fortran | 800 | 50 |
| GNOME- PASCAL | 17,000 | 650 |

Two statistics are given for each listed tool. The first number is an approximate count of the number of lines of source code. The second number is an approximate count of the number of procedures, objects, and types in the system. For the ALOE editors that are listed (all the systems from the GANDALF Prototype to the end are ALOE editors), the statistics are for the semantic portions of the code.

**Figure 2-6:** GANDALF Code Sizes

While IPC was successful for fairly small, stand-alone systems, as more of the GANDALF work became cooperative in nature, it was insufficient for our needs. So, by merging the concepts and code of IPC and SDC, we developed the SMILE (Software Management and Interactive Language-Oriented programming Environment). SMILE adds the notions of reserve, deposit, and history files to IPC and can be considered to be a small, hand-coded, version of the GANDALF Prototype. It supports concurrent users well and has been used for almost all GANDALF development over the past three years.

SMILE was developed and maintained by Feiler, Barbara Denny, and the author. In more recent years, Bob Ellison, Charlie Krueger, and others too numerous to mention have taken a hand to the code. It currently consists of about 14,500 lines and 600 items.

## 3. OUTSTANDING ISSUES AND PROBLEMS

There are many outstanding issues and problems that the GANDALF project is likely to pursue. The following brief outline of these research and development areas should help give a feel for the future directions of GANDALF. The ideas are not necessarily well-formed, as yet, and should not be construed as a plan of action.

### 3.1. Facilities

There are a variety of facilities that we must improve upon for all of the environments in which we are interested.

Improving the screen display, both in utilization and performance, is an area in which we have worked and intend to continue working. Some of the specific areas of study include:

Our recent experiments with workstations -- in particular, PERQs and SUNs -- has given us a chance to study the obviously useful mouse technology. As our previous systems have always been constructed on twenty-four by eighty "smart" terminals, this is an advance that is new to us, although we can clearly take advantage of the well-known work in the field. Straightforward modifications of the cursor-movement routines have already been made in order to permit selection of specific subtrees in an editor. Selection of operators through actual picking from displayed menus is planned.

We have yet to investigate the best ways to use the added real estate available on these more powerful displays.

Display based on syntax-directed editing requires unparsing from the internal tree structure to the text version to be displayed. Our long-time implementation of unparsing has, throughout its many incarnations, been primarily a two-level scheme that maps trees to a form of text that a display package understands. David Garlan is considering a new approach to unparsing in which the internal tree structure is mapped to an intermediate form that is later mapped to text. More function, improved efficiency, and greater clarity of the process are expected using this approach.

A common database is central to all ALOE editors, including GANDALF Environments. In general, the use of a common database has successfully facilitated wide-scale integration of a variety of software development tools into a single environment. Several problems, however, have arisen in relation to use of the database:

The physical size of the database has increased with the more complex editors and environments that we have developed. Segmenting the physical database is necessary.

On the other hand, developing large environments is an intellectually complex task. Logical segmentation of the database into smaller pieces is necessary in addition to the physical separation. Whether logical and physical segmentation of the database should be related or separate is a current issue being discussed.

Even when a large database is logically separated, there are often abstraction problems that remain. In particular, when several tools -- for instance, a user interface and a symbol table manager -- are using the same logical set of data, it is frequent that each tool needs only a subset of the logical portion of the database. Garlan is studying how to permit different tools to have different *views* of the database [34].

Issues surrounding the command interpreter include:

Further study of the problems of entering and manipulating expressions. Kaiser and Kant's paper, which appears later in this issue, discusses an alternative to both the standard parsing and GANDALF approaches.

An understanding of the degree to which the internal structure should be apparent to the user.

Investigation as to whether command interpreters for novices and experts should be similar or how they should be different.

Development of macro mechanisms for combining frequently executed sequences of commands into a new command.

As the entire computer industry moves towards distributed systems consisting of individual workstations, so does the GANDALF project. As with other research projects, our problems with distribution of the system focus on the database. How should we distribute it? Should it be centralized or decentralized?

## 3.2. Syntax

To a large degree, we are happy with our approach to dealing with the syntactic portion of syntax-directed editors. There are some particular areas that still need some work, however. Some of these, including a few mentioned in the previous section, are:

An improvement in the unparsing facilities. In addition to improving the performance of unparsing, the creation of unparsing schemes for complex editors must also be eased.

In many ways, the multiple views mentioned above are largely syntactic. Some of these views are *static views*, that is, views that do not depend on the contents of the trees -- for instance a view of a program tree that displays only the comments. Other views are *dynamic views*, that is, views that vary according to specific values in the trees -- for example a view of all occurrences of uses of a given variable. Ways for the implementor to create static and dynamic views and for the user to access the database by these views need to be understood in greater depth.

A hierarchical help facility is needed. When many commands are consistently available, as is true in ALOE editors, a flat help system is overwhelming in most cases. Access to a multi-level help system is a must to ensure that the system is both useful and used.

One potential disadvantage of syntax-directed editors is that sharing among disparate tools may be difficult. In text-based systems, the text itself is the common medium that is used for sharing: Each tool need only start with a parser and end with a text formatter for the data in order to communicate with other tools. In syntax-based systems, where the primary representation is the internal structure representation, sharing is more difficult. In particular, we need tools to support tree-to-tree syntax-based transformations that can then support sharing of structure-based databases among tools.

## 3.3. Semantics

The advances that we have made in the area of syntax have not yet been followed by equally clear-cut advances in the area of semantics. A powerful alternative to our use of action routines for implementing semantics for an environment is attribute grammars, as proposed by Reps and Teitelbaum as part of the Synthesizer Generator project at Cornell. Attribute grammars have the advantage of being formal but are largely limited to checking of static semantics. As we are interested in dynamic, run-time semantics as well, action routines have served us better. However, the procedural and *ad hoc* nature of action routines make it difficult to develop and maintain our systems. Much of our current work focuses largely on developing formal semantic models that permit us to more easily develop and maintain environments that support dynamic as well as static semantics.

Development of a more consistent model for action routines. Work by Kaiser, Vincenzo Ambriola, and Ellison on definition of ARL, an Action Routine Language, is underway. Kaiser, in her thesis work, is also defining TML, a tree manipulation language, to support construction of action routines for syntax-directed programming environments. TML can be considered to be the assembly language into which ARL programs are compiled.

An extensive action routine library is needed. Many action routines fall into one of several predictable patterns, and it may well be possible to construct routines that can be easily used or included from libraries rather than built from scratch. Some of these routines may be generally useful by all ALOE editors, for example window managers. Others may be specific to GANDALF Environments, for instance routines to apply actions at all use sites of a particular variable.

The success of attribute grammars for static semantic checking in syntax-directed editors, as done at Cornell, makes it attractive to study embedding some of these concepts into the action routine world. An ATribute-Action Routine Environment (ATAR) for combining the benefits of attribute grammars, and their incremental evaluation, into action routines in a consistent fashion, is being considered. ATAR is intended to be a support environment for the development of ARL programs.

LOIPE -- both in stand-alone form and as embedded in GP -- is the only GANDALF Environment that supports program execution. As LOIPE is hand-coded, it does not easily fit into the generation process that we desire. We are interested in finding ways in which we could more easily construct execution environments for a variety of GANDALF Environments. This issue's paper by Ambriola and Montagero describes one approach, based on denotational semantics, for automatically generating an interpreter for a given language.

### 3.4. Expertise

The concepts of syntax and semantics, while necessary and powerful components of software development environments, are insufficient to help with many problems that arise. Specifically, methodologies and rules for supporting the software development process are difficult to embed in syntactic and semantic descriptions of an environment. We call this added knowledge *expertise*. In other words, a software development environment needs to be, in effect, an expert system for supporting software development. The issues we are beginning to study surrounding expertise include:

What kinds of rules can we embed into our environments? How flexible are these or should they be?

What kinds of mechanisms are useful for declaring or describing these rules? Should we take advantage of the experience that AI has had with production systems for developing knowledge-based systems? Indeed, the power of the underlying mechanism will define the kinds of rules and expertise that we can hope to add into our environments. The power of the mechanism, however, must be tempered with the importance of making it relatively easy to develop and generate a variety of environments with varying rule systems.

Our initial belief is that rules should be attached to operators in our grammars. In the sense, the rules would be typed according to the syntactic structure of the underlying language, which should give added efficiency. If our investigations bear this out, it would lend AI people a hand by adding the notion of typing to production systems.

What would the environment for declaring these rule systems look like?

Will there be an expertise library, similar to an action routine library, that will ease generation of similar environments?

## 4. THE OTHER PAPERS

The remainder of this special issue consists of four papers relating to some issues and problems that have arisen over the life of the GANDALF Project. A fifth paper, which is an annotated bibliography of the significant GANDALF publications is the final paper in the group.

The first paper, by Ellison and Staudt, looks at some of the changes that the ALOE kernel has gone through since its initial development by Medina-Mora. The model of the user interface is discussed in particular.

A general concern with syntax-directed editors is whether expressions should be dealt with as structure. The Cornell Program Synthesizer chose to have the user edit expressions textually rather than as structure; on the other hand, most of the ALOE editors that have been constructed have supported strict views of expressions as structure. The second paper, by Kaiser and Kant, describes an algorithm for incrementally parsing expressions in the context of syntax-directed editors. Their approach presents a third option, based on the manipulation of lexical tokens, for having users easily manipulate expressions in the context of a syntax-directed editor.

One criticism of the GANDALF Prototype, and the version control in particular, has been that it requires permanent retention of all the information that has been created by the system. The third paper, by Habermann, describes some strategies and mechanisms for deleting obsolete information in programming environments. While created with environments such as SVCE in mind, the concepts should be applicable to a wider variety of environments.

The development of an interpreter for a given programming language is an *ad hoc* process in GANDALF. The formal techniques that have led to easier construction of compilers, for instance, have not been embedded in GANDALF. The fourth paper, by Ambriola and Montagero, proposes a method for deriving an ALOE editor that can generate an interpreter for a given language. The derivation is based on applying semantics-preserving transformations to a denotational semantic description of a language in order to obtain the required extended commands for the ALOE editor.

## 5. CONCLUSION

Neither this paper nor the rest of the papers in the issue are intended to give a complete overview of the GANDALF project. Rather, in this paper we have tried to give a general feel for how the project has reached its current state and where the project is going. The other papers in the issue show how we have addressed some specific problems. An annotated bibliography of GANDALF literature is also provided.

ACKNOWLEDGEMENTS
Thanks go to the many members (both past and present) of the GANDALF project. We also thank Bill Riddle and the referees for their comments. Special thanks go to Charlie Krueger for his yeoman's effort in preparing camera-ready versions of all the papers in this issue.

## 6. GLOSSARY OF GANDALF TERMS

The following glossary of terms related to GANDALF should help in understanding the rest of the GANDALF literature.

*Abstract syntax.* A syntax that defines the underlying structures, ignoring the syntactic sugar, of a language.

*Action routines.* Procedures associated with node types that are implicitly invoked as modifications are made to nodes of that type. Action routines are used to apply non-syntactic actions to the tree.

*ALOE.* The kernel that supports generation of tree-oriented syntax-directed editors in the GANDALF project.

*ALOE editor.* A language-specific syntax-directed editor generated using the ALOE kernel.

*ALOE preprocessor.* A predecessor of ALOEGEN that generated language tables from a text description of the syntax of a language.

*ALOEGEN.* An ALOE editor that supports definition of the syntactic descriptions that can be transformed to generate other ALOE editors.

*ALOELIB.* A library, used by action routines and extended commands, that provides basic routines for tree manipulation.

*Classes.* The rules that determine how operators for a specific ALOE editor can be composed into syntactically correct trees.

*Concrete syntax.* A description of the syntactic sugar and formatting information associated with the abstract syntax of a language.

*Extended commands.* Procedures associated with a command name that are explicitly invoked to perform some user-level action.

GANDALF *Environment.* An ALOE editor that supports a wide range of software development activities, generally in the manner of the GANDALF Prototype (which is the classic example of a GANDALF Environment).

GANDALF *Prototype.* The ALOE editor that supports project management, system version control, and incremental programming in the context of the GC, or GANDALF C, language.

GANDALF *System.* The combination of the ALOE kernel, ALOEGEN, ALOELIB, and a library of action routines that help ease the generation of software development environments similar to the GANDALF Prototype.

*Incremental programming environment.* A programming environment that supports a small-grained {modification, execution} cycle.

*Non-terminals.* Operators that represent nodes of the tree that have offspring.

*Operators.* The set of language-specific node types used by a specific ALOE editor, for instance an IF node in a PASCAL grammar.

*Programming environments.* Software environments intended to help ease the programming process. These are typically single user systems.

*Project management subenvironment.* The piece of system development environments that helps control the cooperation, coordination, and communication among project members.

*Software development environments.* Software environments intended to help ease the software development process. Software development environments include both programming and system development environments.

*System development environments.* Software environments intended to help reduce the complexities that arise as multiple programmers work on large, complex systems over time.

*System version control subenvironment.* The piece of system development environments that helps control the proliferation, description, and construction of multiple versions of a system and its subsystems.

*Terminals.* Operators that represent leaf nodes of the tree.

*Unparsing.* The process, driven by the concrete syntax of a language, that maps an abstract syntax into a textual representation to be displayed to a user.

## REFERENCES

1. David Notkin and Barbara Denny, Carnegie-Mellon University, Computer Science Department, *SDC User Manual (Version 2)*, 1981.
2. Lee Cooprider, *The Representation of Families of Software Systems*, PhD dissertation, Carnegie-Mellon University, Computer Science Department, April 1979.
3. N. Habermann, L. Flon, and L. Cooprider, Modularization and Hierarchy in a Family of Operating Systems, *Comm. ACM*, Vol. 19, No. 5, May 1976, pp. 266-272.
4. Walter F. Tichy, *Software Development Control*

*Based on System State Descriptions*, PhD dissertation, Carnegie-Mellon University, Computer Science Department, January 1980.
5. Frank DeRemer and H. Kron, Programming-in-the-Large Versus Programming-in-the-Small, *IEEE Trans. Software Eng.*, Vol. SE-2, No. 2, June 1976, pp. 80-86.
6. United States Department of Defense, *Reference Manual for the Ada Programming Language*, 1982, AdaTEC Special Publication.
7. Warren Teitelman, A Tour through Cedar, *Proceedings of the 7th International Conference on Software Engineering*, March 1984, pp. 181-195.
8. Eric Schmidt, *Controlling Large Software Development in a Distributed Environment*, PhD dissertation, University of California, Berkeley, December 1982, Appears a Xerox PARC Technical Report CSL-82-7.
9. Butler W. Lampson and Eric. E. Schmidt, Organizing Software in a Distributed Environment, *Proceedings of the Sigplan '83 Symposium on Programming Language Issues in Software Systems*, June 1983, Issued as Sigplan Notices (18,6).
10. Marc J. Rochkind, The Source Code Control System, *IEEE Trans. Software Eng.*, Vol. SE-1, No. 4, December 1975, pp. 364-370.
11. W. F. Tichy, Design, Implementation, and Evaluation of a Revision Control System, *Proceedings of the 6th International Conference on Software Engineering*, September 1982.
12. Stuart I. Feldman, Make -- A Program for Maintaining Computer Programs, *Software Practice and Experience*, Vol. 9, No. 4, April 1979.
13. Peter H. Feiler, *A Language-Oriented Interactive Programming Environment Based on Compilation Technology*, PhD dissertation, Carnegie-Mellon University, Computer Science Department, May 1982.
14. Raul Medina-Mora and Peter H. Feiler, An Incremental Programming Environment, *IEEE Trans. Software Eng.*, Vol. SE-7, No. 5, September 1981, pp. 472-482, Original version in 5th ICSE/Schlumberger workshop(??).
15. Peter Fritzson, *Towards a Distributed Programming Environment Based on Incremental Compilation*, Department of Computer and Infromation Science. Linkoping University., S-581 83 Linkoping, Sweden, Linkoping Studies in Cience and Technology. Dissertations., Vol. No. 109, 1984.
16. Raul Medina-Mora, *Syntax-Directed Editing: Towards Integrated Programming Environments*, PhD dissertation, Carnegie-Mellon University, Computer Science Department, March 1982.
17. Raul Medina-Mora, *et al.*, ALOE Users' and Implementors' Guide, December 1983.
18. T. Teitelbaum and T. Reps, The Cornell Program Synthesizer: A Syntax-directed Programming Environment, *Comm. ACM*, Vol. 24, No. 9, September 1981, pp. 563-573.
19. T. Teitelbaum, T. Reps, and S. Horwitz, The Why and Wherefore of the Cornell Program Synthesizer, *SIGPLAN Notices*, Vol. 16, No. 6, June 1981, pp. 8-16, The Proceedings of the ACM SIGPLAN/SIGOA Symposium on Text Manipulation.
20. V. Donzeau-Gouge, *et al.*, A Structure Oriented Program Editor: A First Step Towards Computer Assisted Programming, Tech. report 114, INRIA, April 1975.
21. V. Donzeau-Gouge, *et al.*, Programming Environments Based on Structure Editors: the Mentor Experience, Tech. report 26, INRIA, May 1980.
22. Thomas Reps and Tim Teitelbaum, The Synthesizer Generator, *Proceedings of the ACM/SIGSOFT Software Engineering Symposium on Software Development Environments*, April 1984, pp. 42-48.
23. G. Kahn, *et al.*, Metal: A Formalism to Specify Formalisms, Tech. report, INRIA, 1982.
24. Steven P. Reiss, Graphical Program Development with PECAN Program Development Systems, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, April 1984, Issued as Sigplan Notices (19,5) and Software Engineering Notes (9,3), May 1984.
25. C. N. Fischer *et al.*, The POE Language-Based Editor Project, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, April 1984, Issued as Sigplan Notices (19,5) and Software Engineering Notes (9,3), May 1984.
26. J. R. Horgan and D. J. Moore., Techniques for Improving Language-Based Editors, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, April 1984, Issued as Sigplan Notices (19,5) and Software Engineering Notes (9,3), May 1984.
27. Norman M. Delisle, David E. Menicosy, and Mayer D. Schwartz, Viewing a Programming Environment as a Single Tool, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, April 1984, Issued as Sigplan Notices (19,5) and Software Engineering Notes (9,3), May 1984.
28. Thomas A. Standish and Richard N. Taylor, Arcturus: A Prototype Advanced Ada Programming Environment, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, April 1984, Issued as Sigplan Notices (19,5) and Software Engineering Notes (9,3), May 1984.
29. L. J. Osterweil, Toolpack -- An Experimental

Software Development Environment Research Project, *IEEE Trans. Software Eng.*, Vol. SE-9, No. 6, Novemeber 1983, pp. 673-685.

30. W. Teitelman and L. Masinter, The Interlisp Programming Environment, *Computer*, Vol. 14, No. 4, April 1981, pp. 25-33.
31. A. Nico Habermann and David Notkin, The GANDALF Software Development Environment, *Proceedings of the Second International Symposium on Computation and Information*, Monterrey, Mexico, September 1983, Also in the *Second Compendium of* GANDALF *Documentation*, Carnegie-Mellon University, Department of Computer Science May 1982. A revised version has been submitted for further publication.
32. David B. Garlan and Philip L. Miller, GNOME: An Introductory Programming Environment Based on a Family of Structure Editors, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, April 1984, pp. 65-72, The proceedings appeared as Software Engineering Notes (9,3) and Sigplan Notices (19,5) for May 1984.
33. Richard E. Pattis, *Karel the Robot: A Gentle Introduction to the Art of Programming*, Wiley, 1981.
34. David B. Garlan, Views for Tools in Software Development Environments, May 1983, Thesis Proposal. Carnegie-Mellon University, Department of Computer Science, May 1983.