Gandalf: Software Development Environments

A. NICO HABERMANN AND DAVID NOTKIN, MEMBER, IEEE

Abstract—**Software development environments help programmers perform tasks related to the software development process. Different programming projects require different environments. However, handcrafting a separate environment for each project is not economically feasible. Gandalf solves this problem by permitting environment designers to generate families of software development environments semiautomatically without excessive cost.**

Environments generated using Gandalf address both programming environments, which help ease the programming process, and system development environments, which reduce the degree to which a software project is dependent on the good will of its members. Gandalf environments integrate programming and system development, permitting interactions not available in traditional environments.

The paper covers several topics including the basic characteristics of Gandalf environments, our method for generating these environments, the structure and function of several existing Gandalf environments, and ongoing and planned research of the project.

Index Terms—**Environment generation, incremental program construction, programming environments, project management, software development environments, structure-oriented editing, syntax-directed editing, system development environments, system version control.**

I. INTRODUCTION

EVEN simple programming is a challenging process. When compounded by the difficulties of scale, the process becomes so complex as to rely heavily on the good will of programmers. The basic goal of a *software development environment* is to provide software support that helps simplify the software development process. Current research in software development environments is taking place in at least two areas. The first area is concerned with developing *programming environments* that ease the programming process itself. The second area focuses on construction of *system development environments* that reduce the degree to which a software project is dependent on the good will of its members. The Gandalf project [44] is unusual in its interest in producing, through semiautomatic generation, software development environments that integrate the notions of both programming and system development environments. In this paper we describe both the characteristics of Gandalf software development environments and also the mechanisms used to generate such environments. We further describe three actual environments, constructed using these mechanisms, that manifest these characteristics. We conclude with a description of ongoing and planned research by the members of the Gandalf project.

A. Characteristics of Gandalf Environments

Software development environments support many tasks related to the software development process. Users of these environments are provided with program editors, debuggers, version control systems, and documentation support tools, just to name a few. The ways in which the user communicates with the environment and the ways in which the tasks within the environment interact characterize Gandalf software development environments.

- *Integration:* A common development database acts as a conduit for sharing knowledge among tasks in a Gandalf software development environment. Integration through sharing permits tasks to interact in ways not possible in nonintegrated environments, which are structured as independently created tools. For example, consider a traditional environment that uses the SCCS source code control system [55] in the context of Ada® [65]. If a programmer simply changes a comment in the visible part of an Ada package, the entire package, as well as all dependent packages, must be recompiled because SCCS has no understanding of the contents of the files that represent the packages. In an integrated Gandalf software development environment, the knowledge that only a comment has been changed is accessible to the compiler and the version control system, allowing such unnecessary recompilation to be avoided.
- *Uniformity:* The wide range of tasks supported by software development environments increases the need for uniform interaction with the users. Uniformity and consistency of the interface reduces the costly overhead common in introducing new tasks and in educating new users. Gandalf environments support uniform interaction by providing commands that are broadly applicable as well as consistent mechanisms for command application. Integration provides additional uniformity because the separate tasks are viewed as a single environment.
- *Interactive User Interface:* Gandalf environments and their users communicate through interactive user interfaces. Small-grained interaction provides benefits at various levels of an environment. For example, interaction during program editing can support creation and inference

Manuscript received August 31, 1982; revised January 31, 1986. This work was supported in part by the Software Engineering Division of CENTACS/CORADCOM, Fort Monmouth, NJ.

A. N. Habermann is with the Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213.

D. Notkin is with the Department of Computer Science, University of Washington, Seattle, WA 98195.

IEEE Log Number 8610712.

®Ada is a registered trademark of the U.S. Department of Defense (Ada Joint Program Office).

of type declarations for used but undeclared variables. As another example, if a user makes a change that would require recompilation of the entire system, a warning to this effect can be given before the change is actually implemented. If the change is unintended or can be avoided by an alternative approach, the user can avoid massive recomputation. Interactions of this style are not easily provided in noninteractive systems.

•*Well-Defined System State:* Information describing the state of the software system under development is incorporated into a Gandalf software development environment. This state information permits direct enforcement of rules that help users manage the complex system development process. Proper definition of the environment's commands allows a well-defined state to be retained; the commands and the state can be viewed as forming an abstract data type. Many management-oriented approaches to software development [3], [4] use specialized personnel and approaches to ensure that a well-defined state is retained. In Gandalf software development environments, this function is embedded in the environments themselves.

In addition to these general characteristics, Gandalf programming and system development environments each have some specific characteristics. Gandalf programming environments are language-oriented editing systems that support source-level debugging and either interpretation or incremental compilation (with associated incremental linking and loading). Gandalf system development enviroments are characterized by:

•*System version control* support that aids in describing and manipulating the interfaces, composition, and dependencies of modules, subsystems, and systems. Such support is related to the notion of programming-in-the-large [10].

•*Project management* support that helps control the development process so that programmers make changes in an orderly fashion. Since this is essential to the cooperation, communication, and coordination of programmers in a project, we have christened this *programming-in-the-many.*

These are general characteristics that may be realized in many ways. The details of several different Gandalf environments that represent such realizations are described in Section III.

B. Gandalf's Project Orientation

In constructing Gandalf software development environments with the given characteristics, an environment designer must decide whether rules and conventions apply uniformly to all users or whether they may be tuned to satisfy the individual. The Gandalf project has taken a position between these extremes: Gandalf promotes the creation of *project-oriented* software development environments in which many traits, such as protection policies, are tuned to groups of persons working on a project rather than to the entire computing community or to particular individuals. Tuning to the community requires prescience that is, at best, beyond our abilities. Tuning each trait to each individual user is sometimes undesirable. In most projects, for example, the choice of programming language is made by management rather than individual programmers. Other traits only make sense when considered in a context greater than that of an individual. For instance, a protection policy for access to shared information must be common for all members of a project.

Hand-crafting a software development environment for each project is economically infeasible. Gandalf solves this problem by generating sets of related environments. Each environment shares the characteristics and objectives described above, and may be constructed without excessive cost. Gandalf uses the *Gandalf System*—the designers' environment—to facilitate semiautomatic generation of a set of related environments [16].

C. Related Work

Related work on software development environments falls into two categories: projects that consider programming environments and projects that consider system development environments.

Many programming environments are language-oriented systems: some have been generated while others have been constructed by hand. Gandalf's previously described IPE [39], [18] is a system, generated from Medina-Mora's ALOE [41], [40], that integrates a syntax-directed editor with an incremental compiler and linker to produce a uniform environment for GC.[1] The Cornell Program Synthesizer [62], [63] is a handcrafted environment for PL/CS, a teaching subset of PL/I, that supports execution through interpretation. Interlisp [61], Smalltalk [26], [27], Mesa/XDE [58], and Cedar [59], [60] are all single-language programming environments developed at Xerox, each of which supports powerful debugging and error correction tools in the context of a powerful personal workstation with a high-resolution bit-map display and a mouse.

Other interesting language-oriented systems and generators include: the Synthesizer Generator [49], [54], which uses optimal incremental attribute grammar evaluation for checking of static semantics in a syntax-directed editor [9], [51]–[53]; Mentor [13]–[15], which uses a descriptive language called Metal [31] to construct a multilingual editing environment; PDE [1], [42], an editing and execution environment based on incremental parsing techniques; Syned [24], [30], a totally hybrid system that supports editing of text or structure at any point; Pecan [47], [48], a graphics-oriented system that provides a wide variety of user views of a program under development; Magpie [11], a Pascal programming environment based on incremental compilation; Cope [2], [8], which provides powerful recovery techniques in the context of a language-oriented editor; Poe [21], which relies on syntactic error repair to retain correctness of programs entered as

[1]GC, or Gandalf C, is a slightly modified version of C [35] that supports type checking of parameters.

text; $\mathbf{R}^n$ [29], an environment for Fortran; DICE, a distributed incremental programming environment [22], [23]; and DOSE, a graphics-oriented structure editing environment generator produced at Siemens [19], [34].

System development environments vary greatly. UNIX™ [36] and the Programmers' Work Bench, or PWB/UNIX [12] are typical examples of the tool-kit approach to environments, where development tools are added to an existing environment with little regard to the commonality among the tools; further, communication among tools is entirely based on conventions. Toolpack/IST [45] focuses on integrated environments that support mathematical software development through the use of a central database. Cades [56] is an environment that attempts to mechanize an entire project organization. The Stoneman [5] requirements for an Ada programming environment, describes, in varying degree of detail, how such an environment should address nonprogramming issues such as configuration management. Arcturus is a prototype of an advanced Ada programming environment [57]. The SAGA Project focuses on producing environments that manage all phases of the software life-cycle for small to medium-sized development projects [6], [37]. The IPSEN project is concerned with software development environments based on underlying graph technololgy as opposed to the more common tree model [17]. Some of the programming environments address some of the system development environment issues as well; for example, Xerox's Cedar supports users in the system modeling process [38].

Many of these programming and system development environments share some of Gandalf's goals and characteristics. Gandalf differs from these systems by focusing on both areas—easing of the programming process and reducing the dependence on good will—as well as its strong emphasis on the generation process.

II. The Generation Environment

The need for semiautomatically generating software development environments arises because each project requires a different, although related, environment that must be constructed in a cost-efficient manner. The Gandalf System facilitates this process through the use of the ALOE structure-oriented editor generator [40], [41].

A. User View of ALOE Editors

The commands of ALOE editors generated by this process are based on software development constructs (such as programming language statements, access control lists, and module descriptors) rather than on lines and characters. Each ALOE editor provides two kinds of commands to manipulate the database trees: language-dependent *constructive* commands and language-independent *editing* commands.

Each ALOE editor provides one constructive command for each construct that can be created. Constructive commands are best described by an example that shows how to construct **while** statements in a procedural programming language: when a user issues the constructive command "wh" the ALOE editor responds by displaying:

```
while %bool-exp do
   %stat
od
```

with the cursor on the screen highlighting the syntactic unit "%bool-exp". The phrases "%bool-exp" and "%stat" are typed placeholders that are to be expanded. For example, "%bool-exp" could be replaced with a relational expression or a conjunction of Boolean expressions involving Boolean operators, but not with a variable declaration. Similarly, the programmer can choose to replace "%stat" by an assignment statement, a compound statement, etc. The user continues expanding placeholders until all leaves of the tree are replaced by terminals such as variable names or constants. The syntactic correctness of the tree is ensured by limiting the sets of constructs that can replace particular types of placeholders.

All ALOE editors share a collection of language-independent editing commands that manipulate the database trees. Typical commands are "delete a construct" and "move the cursor." On deletion or transformation, syntactic correctness is preserved by replacing the modified nodes with the appropriate placeholders.

Use of ALOE as the basis for Gandalf environments directly supports the construction of environments with interactive and uniform user interfaces. The interactive nature of ALOE editors is apparent. The uniformity of ALOE interfaces is achieved in two ways. First, the language-independent commands are shared among all ALOE editors. Second, the way that the language-dependent commands are applied is uniform across all editors, even though the structures that they represent vary.

B. Constructing Editors

Generation of these editors, which form the basis of Gandalf software development environments, is done with the support of the ALOE Generator. This generator takes as input a description of the language that is to be manipulated and produces as output an editor for that language. In describing a language, the environment implementor must give a description of the *abstract syntax*, the *concrete syntax*, and the names of *action routines* to be called as the tree is manipulated. The high-level view of this process is shown in Fig. 1.

The abstract syntax for a language describes the *operators*, which include the terminals and nonterminals, and the *classes*, which indicate the set of operators that may replace particular placeholders. An example of one nonterminal definition and two class definitions is:

```
      WHILE = %bool-exp %stat
%bool-exp = ID | TRUE | FALSE | LSS | LEQ |
            EQL | NEQ
    %stat = ASSIGN | IF | FOR | WHILE | RE-
            TURN
```

™UNIX is a trademark of AT&T Bell Laboratories.

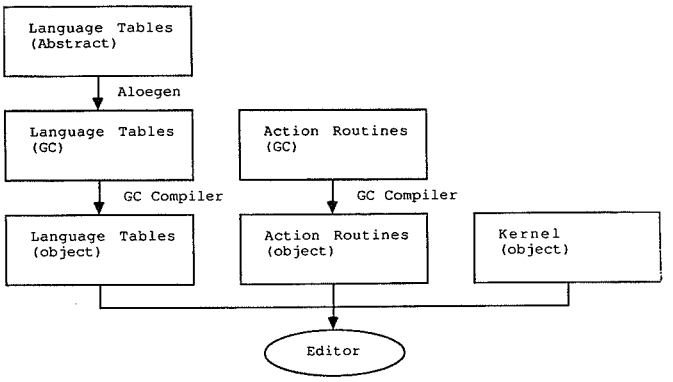

Fig. 1. Constructing an editor in ALOE. Language tables are developed in an abstract form that represents the abstract and concrete syntax directly. They are then transformed by ALOEGEN, an ALOE editor that helps the implementor construct new editors, into GC tables that are compiled. The action routines are defined in GC and compiled. The object code for the tables and the action routines are linked with the object form of the kernel to produce an editor.

The children of operator "WHILE" are the placeholders "%bool-exp" and "%stat". These placeholders represent the two classes with corresponding names that list the possible operators that can be substituted for these placeholders.

The concrete syntax describes a suitable screen representation of each language operator. The visible representation on the screen is derived from the syntax tree by the *unparser*, an ALOE task that maps the syntax tree representation of a program into a textual representation. It performs this mapping while using the concrete syntax provided by the environment implementor. An example of a concrete syntax description, or *unparsing scheme*, is:

"WHILE = **while** @1 **do** @N@2"

where @1 and @2 designate the children of the WHILE operator and @N the insertion of a new line in the output. The design of the concrete syntax is largely up to the editor implementor. For instance, simply changing the **do** to **begin** above would change the displayed representation of the **while** loop, but not the abstract structure.

Another feature permits an editor implementor to define more than one unparsing scheme per operator. The implementor may then choose to provide one representation without semicolons, another with semicolons, and yet another with explicit closing delimiters (e.g., "**end loop**"). A more important application of the multiple unparsing feature concerns the abstraction of detailed information. For instance, a procedure consists of a name, a parameter list, and a procedure body that contains declarations and statements. At times in the editing process, a user may wish to work on a particular procedure and look at all its detail. At other times, this same procedure is condsidered as part of a collection of objects and the procedure body in particular is rather irrelevant. Multiple unparsing schemes enable the implementor to define two concrete representations for procedures: one that displays the declarations and statements of the body and one that does not. In this manner a programmer can get a bird's eye view of a collection of procedures, abstracting from their implementation details. Unparsing schemes are switched implicitly in response to user commands such as cursor motion.

Action routines implement the specific run-time behavior that the designer might want to realize in the environment being created. Action routines can be used for various purposes including checking of semantics, window and memory management, and keeping the system being developed in a well-defined state. Action routines can check consistency and can enforce system design rules.

The generation process supports construction of Gandalf environments that are integrated. By constructing a grammar that represents the entire set of constructs associated with the software development process, an environmental implementor designs a shared database that supports sharing of knowledge among related tasks of the environment. Such integration makes it easier for a user to view an environment in a uniform way.

III. Existing Gandalf Environments

Several major Gandalf environments have been designed and implemented. One of these is a full-fledged software development environment. The others are smaller scale programming and system development environments. In this section, we focus on three of these environments: the Gandalf Prototype, a complete Gandalf software development environment; GNOME, a set of programming environments used in an educational situation [7], [25]; and SMILE, an internal system development environment, which, although it is not a generated environment, still displays many of the characteristics of Gandalf environments. All of the environments described in this section have been implemented fully.

Although we do not have room to discuss them all, a variety of other smaller scale environments have been constructed. Most noticeable is ALOEGEN, a Gandalf environment that supports creation of descriptions from which other Gandalf environments can be generated. ALOEGEN's existence is an indication of the ability of Gandalf environments to be bootstrapped. Environments for Alfa (a functional programming language), Ada [65], and Modula-2 [66] have also been developed. Environment for tasks other than software development—such as mail systems and document formatting systems—have been constructed as well [43].

To understand the descriptions of the GP and GNOME environments, it is essential to remember that they are implemented as ALOE editors. For example, construction of nodes is always performed by replacing typed placeholders; this holds true whether the user is creating an IF statement or an import clause for a module. In several cases, additional language-specific commands have been added to particular environments. These commands are displayed in **boldface** throughout the descriptions.

A. The Gandalf Prototype

The Gandalf Prototype (GP) is a full-fledged software development environment created by members of the

Gandalf group. GP, while a prototype rather than a production system, displays all the characteristics of Gandalf environments described in the introduction. The primary focus of this description is to present GP's system version control, incremental compilation, and project management support for GC programs.

The construction of GP is based on development of an abstract ALOE grammar that represents a set of software development structures. Since GP supports more than just programming, constructs in the system include modules and their versions, access control lists, documentation, etc. Throughout the descriptions of the rest of GP, it is important to remember that the described structures are actually pieces of structure, represented by ALOE operators, which appear in the grammar description for ALOE. Action routines, along with several editing commands added specifically for GP, implement the required semantic checking and execution support.

System Version Control: The system version control support in GP supports two objectives—description of systems and automatic generation of system versions. Further details on system version control in GP are available elsewhere [28], [33].

GP's description of a software system has a static and a dynamic component. The static component is the description of modular interfaces, specifying how the *facilities*—such as data objects, type definitions, and procedures—can be used in other modules. The dynamic component is the description of the composition of a subsystem out of particular versions of modules.

A system description is based on several notions, each of which is defined as a piece of the abstract syntax of GP. *Boxes*, similar to directories in traditional file systems, contain a collection of other boxes and *modules*. Modules provide both an interface, consisting of a set of facility specifications, and also a set of *versions* that realize this interface in different ways. Each version consists of a set of *revisions* that in turn represent an actual implementation of a module. The tree structure of boxes and modules, along with the two-dimensional structure of versions and revisions within modules is shown in Fig. 2.

GP's system version control eases changes that are frequently made by programmers. Commonly performed actions by system programmers include modification of existing programs, creation of alternative realizations, and modifications of existing interfaces. Minor modifications of existing programs are achieved by developing new revisions for a version. Whenever a modification is made, by application of the **revise** command, an element is added to the list of revisions, ordered by revision date. Creation of alternative implementations is supported by creation of a new version for a module. Each version of a module must realize exactly the same interface. Given this restriction, versions of a module may be arbitrarily substituted and should only affect performance of the system. Another implication of this restriction is that to modify the interface of a module, a new module must be created. This module can be based on an existing module through use of the **diverge** command, which creates a new module

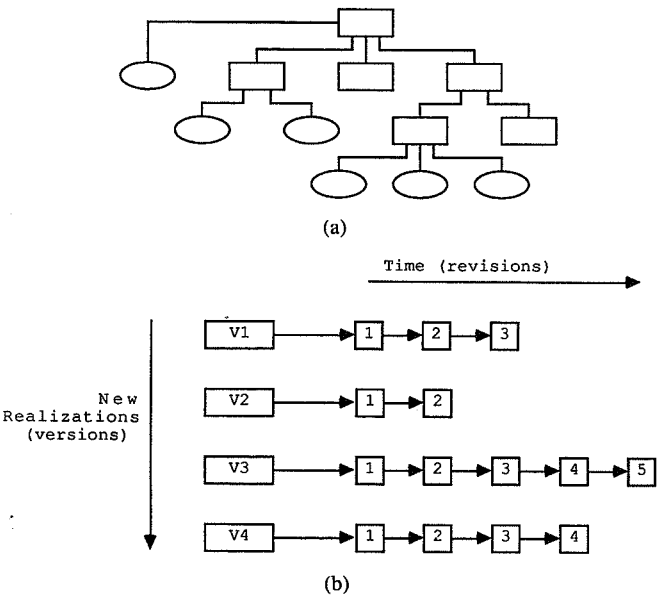

Fig. 2. Version control structures for GP. (a) Boxes, which are represented by rectangles, impose a tree structure on GP's system descriptions. Modules, which are represented by ovals, represent definitions of program modules. (b) Each of the modules [the circles in part (a)] can be realized by multiple versions. Each version can by successively updated, over time, to produce a set of revisions.

node within a selected box, renames the module, and copies a selected version and revision into the new module.

GP further eases extension and modification of interfaces by permitting realizations of modules to use facilities defined in other modules. This is accomplished by distinguishing between *implementations*, which represent actual code, and *compositions*, which represent repackaging of existing modules.

An implementation is essentially a source program that defines bodies for all the facility specifications in the module interface. An implementation description consists of a name for the implementation along with a *with list* that lists imported module names. In addition to supporting abstraction, permitting importation of interfaces, eases augmentation of existing modules. For example, suppose we have a module M that exports the list of facilities (f, g) and suppose we need the extended list of facilities (f, g, h). This is accomplished by defining a new module P with an implementation P1 that consists of a program for facility h, while its with list contains module M (note that the with list is associated with the implementation instead of the module since there can be multiple implementations of a single module):

```
module P provides f, g, h;
   impl P1 with M;
end P
```

Implementation P1 is a valid representation of module P because the combination of P1's program and module M provides all the facilities exported by module P.

A composition is described by the list of modules or module versions from which it is composed. For example, suppose a module M provides the set of facilities (f, g) and a module N provides the set (h, k). The compo-

sition

comp C = (M, N)

would be a legitimate version of a module that exports the set of facilities (f, g, h, k). GP ensures that the composition provides at least the resources listed in its module's **provides** list.

As the user applies ALOE commands to modify modules, implementations, compositions, and revisions, GP incrementally checks that the well-defined state of the environment is retained. Note that an environment may be in a well-defined but still semantically incorrect state. For example, if a module has no implementation to match a facility specification in an interface, it is correct. However, if the environment knows this is incorrect and does not permit other modules that are dependent on the incorrect module to be built until the facility is implemented, the state is well-defined.

The second objective of system version control is the automatic generation of executable system versions. In traditional programming environments programmers must remember, either on paper or in system command files, the various loading sequences needed to accomplish the correct grouping of module versions. Fine tuning such procedures while including, for instance, the option of skipping a recompilation when a satisfactory object code version exists, is difficult. Instead of placing this burden on the programmers, the GP version control facilities automatically derive, from the system description, the minimal steps needed for system generation. The **make** facility [20] of UNIX supports a similar process but requires the implementor to construct a separate description of the dependencies among the files, not the logical modules, that represent the system.

Subsystems are generated by applying the **instantiate** command to a module, a version, or a revision. For **instantiate** to work in the case that only a module is given instead of a particular version of that module, each module designates one of its versions as its *standard* version. A user may change this designation, but there will always be exactly one. Every time an unqualified module name is encountered during the system generation process, the standard version of that module will be chosen. Similarly, each implementation must have a standard revision.

The policy of selecting the standard versions and revisions gives systems generation a dynamic character. The result of **instantiate** is time-dependent because the standard versions and revisions may change between successive calls. Because of this, descriptions of instantiated systems must store the complete descriptions of all revisions ultimately involved in building the system. Since the standard pieces can change, it is essential to store the actual paths to the pieces assuming that someone may later wish to rebuild the identical system. If versions or revisions other than the standard ones are desired by a project member, they can be designated in with lists and compositions.

Incremental Program Construction: Incremental program construction in GP is based on three improvements in the construction cycle of {text edit, compile, load, execute}. First, the use of an ALOE editor reduces the time and effort it takes to get a program to compile. Second, incremental loading and linking is provided. Third, source-level debugging support is supplied.

Many of the advantages of using an ALOE editor for a programming environment address drawbacks of the standard text editing, parsing piece of the cycle. The basic problem with text editors is that they are so general that they do not understand the objects they manipulate. If one enters the letters "b", "e", "g", "i", "n" on a keyboard, the text editor does not understand that this represents a keyword and understands even less that this word should be matched by an occurrence of a closing keyword. It is peculiar that we enter programs as text while we think of them as having a definite syntactic, to say nothing of semantic, structure. It is even more odd that the very next thing we do is to submit the character string to a syntax analyzer to see if we really entered a legitimate program, which was what we intended to do in the first place. An ALOE editor, on the other hand, helps a user enter programs in terms of the syntactic structures of the language. Hence, it is impossible for a user to produce programs that are syntactically incorrect. This saves substantially in the time it takes to get a program to compile correctly; one no longer needs those few compiles "just to get the semicolons right." (Some modern text-based editors have been extended to help users maintain syntactic correctness of edited programs. However, the other benefits of syntax-directed editors, such as uniformity and cost-effective generation, are not generally available through text editors.)

Modern language systems often provide facilities for separate compilation and some allow incremental compilation. But in most systems it is necessary to repeat the linking process and generate a new executable version of a subsystem if one of the modules is modified. Larger systems take a long time at the link and load phases, which decreases the productivity of programmers. A substantial shortcut is achieved by restricting the linking and loading (as well as the compilation) process to the procedures that are modified. This shortcut can be taken if modifications involve the code of a procedure, but not its specifications of parameters or results. In the latter case, procedures that call the modified procedure must also be recompiled and relinked because the calling sequence is no longer valid. Whether or not a modified executable version can be restarted depends on the effect a procedure or function call has on global data. Restarting at an earlier point, for instance at the beginning of a procedure execution, is possible if there are ways to save the global state at procedure or function entry. In other cases, particularly if procedures have side effects such as reading from input, the execution is irreversible and cannot be resumed at an earlier point.

GP avoids relinking of the entire system by using in-

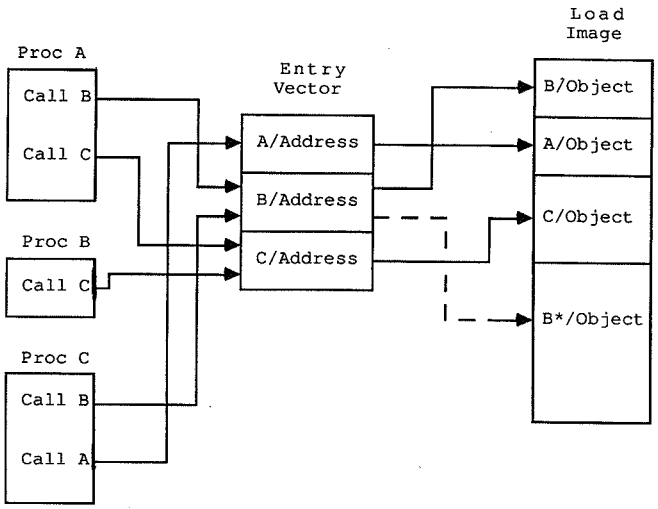

Fig. 3. Incremental linking and loading in GP. The left column represents procedure calls in source programs. At run-time, when a call is made, the address for the called procedure is determined indirectly through the entry vector. For instance, when proc A calls proc B, the object code at the top of the load image is executed. If a modification is later made to the source for proc B, its object code—represented by B* Object in the figure—is loaded at the end of the address space and the entry vector is updated to refer to the new code (the darkened dashed line represents the updated indirect reference for calls on proc B). All new calls to proc B will execute the B* code.

variant addresses for procedures within modules. Each module has an entry vector that contains one entry for each procedure that this module provides to other modules. Outside of a module these procedures are known by a fixed position in the entry vector. This permits relocation of procedures without affecting the references to them from other modules. The content of an entry-vector component is the absolute address of a procedure. Hence, the price for invariant linking is one indirect memory reference at every procedure call. Informal measurements indicate that this cost is acceptable. This approach is displayed in Fig. 3.

Source-level debugging is partially supported by extending the list of statements in GC with TRACE and PAUSE. When the run-time system encounters instances of these statements, state is saved and control is passed back to the user. Commands for starting from an initial state and restarting from a saved state are available to the users. Support for monitoring the value of objects in the user program is also supported through a special monitor window on the screen that contains a list of name and value pairs. To monitor a new object, the user simply enters the monitor window, extends the list of pairs, and places the name of the object in the name field. From then on, the value of the object will appear next to its name. In addition, the value of the object may be changed during a break in the execution simply by entering the monitor window and modifying the value directly. In all cases, any recompilation needed to support the monitoring is done automatically.

Project Management: The function of project management in GP is to guarantee the integrity of development status information and to provide access to this information. To this end, the system descriptions are augmented with some explicit status information, while manipulation of this data is controlled by the action routines of the ALOE interface. Protection against chaotic modifications of the state of a project is obtained by treating system descriptions as collections of typed objects. The abstract grammar for GP restricts modifications that are structurally unsound, but project management supports additional restrictions.

Project management primarily involves the top-level description of a system expressed in terms of boxes, modules, and versions. The system description is extended with some objects that specifically serve the purpose of project management. The most important objects of this kind are *access lists*, of which there is one attached to each box, and *revision histories*, of which there is one linked to each box, one to each module, and one to each module version. The access list of a box describes the rights of the programmers that may manipulate the box. Users are classified by the specific rights they have. The most privileged user is the superuser of a box who has the right to modify boxes and modules, and also has the right to delete information and to change the access list. The next class is that of the project programmers who have the right to modify boxes and modules, but who cannot change the access list and who can only mark information as obsolete. Every user who is included in the access list has the right to read information and to execute programs. A revision history is a collection of messages, ordered by date, briefly describing the revisions that were made. Each message is dated and marked with the name of the programmer who caused the message to be created. The programmer is responsible for writing the text of the message.

When a group of programmers works on a joint project, there is a chance that several programmers simultaneously try to modify some system module. GP project management ensures that such potentially disastrous situations do not arise. Commands are provided to supply the handshaking necessary to avoid such conflicts. The most important such operations are **reserve, release,** and **deposit,** which are similar to commands found in source code control systems like SCCS [55] and RCS [64]. If a programmer intends to modify a module, he calls the reserve command with cursor located at the module. If it is already reserved, an error message will be displayed by ALOE to the programmer. Otherwise, the module is reserved in his name and then copied, in modifiable mode, into the programmer's private database. On completion of his modifications, the programmer may make the changes permanent, by calling **deposit** to place the modified module back into the public database. If, during the modification process, the programmer realizes that he does not wish to make the changes permanent, he may **release** the module, which simply deletes the programmer's reservation and leaves the status as it was just before the reservation was made.

The deposit action is strictly controlled by GP. The deposit command begins by checking that its caller has the rights of a programmer or superuser. Then it checks whether its caller holds the reservation. If these conditions are satisfied, the deposit action prompts the user for a brief description of the modification. The text typed by the user is extended with a heading and stored as a message in the appropriate history list. Next, the list of revision descriptors is updated, and finally the revised program is included as a new source object.

The status of a developing system is accessible to GP users by direct access to the database through ALOE. Creation of information–such as boxes or access control list entries–is done either explicitly through invocation of commands such as **revise** and **instantiate,** or implicitly by construction of nodes through the standard ALOE mechanism.

3. *The GNOME Project*

The GNOME project [7], [25] consists of a family of ALOE editors used as the sole programming environments in the introductory programming courses at Carnegie-Mellon University. The environments have been used for several years by over 700 students a semester. While not supporting version control and project management, as GP does, GNOME is interesting in its implementation of the other characteristics.

The first substantial editor introduced to the students is for Karel [46], a turtle-graphics language designed for instructional purposes. To the student, the Karel editor is a completely self-contained system. Syntactically correct programs are developed, semantically checked, and executed entirely within the context of the editor. Internally, the syntax and semantic checking are performed under the guidance of ALOE, but the execution is performed by the Karel interpreter: after the program is checked by the ALOE version, a text version is created and sent to the Karel parser and interpreter. This separation is hidden from the students by ensuring that static semantics are checked, and any errors are reported, before execution of the program begins.

The definition of the abstract syntax of Karel was initially based on the concrete syntax required by the Karel parser. Over time the grammar has been improved to ease the user interface since in GNOME, with its army of novice users, this is of prime importance. One representative modification is the addition of comment nodes to the grammar.

Many static semantic errors are checked by the implementors of Karel. These can be broken into two groups—incremental checks to be detected as they occur and terminal checks to be detected on program completion. The incremental checks are use of an undefined procedure, multiple definitions of a procedure, self-recursion, mutual recursion, and deletion of a procedure still in use. The terminal checks are procedures out of order, unfilled-in nodes still remain, no accessible **end** statement, procedure defined but not used, and deletion of a procedure still in use. One check—deletion of a procedure still in use—appears in both lists since a user could ignore the warning returned on the initial deletion.

Many of these checks arise simply because the existing Karel parser and interpreter are used for execution. The situation would be different if execution were performed directly from the abstract syntax tree. The most obvious example is the check for unfilled-in nodes, which is necessary in the current Karel since the Karel parser does not understand about such nodes (or, more accurately, the unparsed representation of such nodes). During direct interpretation of the tree, these nodes can simply be ignored. The checks for use of an undefined procedure and deletion of an instruction still in use can also be dealt with differently. As direct interpretation permits execution of incomplete programs, these checks can be delayed until runtime, when the missing procedure can then be defined. Of course, this may not be desirable for pedagogical reasons, but it is technically straightforward and might be a reasonable approach for some classes of users. In addition to these checks, some semantic errors are interactively corrected. For instance, procedures are reordered, if the user desires, to meet the criteria of the Karel system.

GNOME also provides a Pascal editor for creating and modifying stand-alone programs. The initial abstract syntax description used was taken from a standard Pascal syntax description. A variety of problems with this description arose, most of which concerned the user interface. During the development of the initial editor, the abstract syntax description changed frequently. Over time, these changes became less frequent. Two typical changes were:

- Modifications to support abstraction through the use of windows. Procedure headers, for example, are displayed at the main program level by the name of the procedure and the parameter list. Cursor motion into the header causes the procedure to be opened and implementation to be shown (in a window separate from the main program level).
- Support for common procedure calls, such as writeln, were added to the grammar, to ease the students' task in common situations.

Although incremental semantic checking is currently being implemented, the existing GNOME-Pascal editor leaves checking of static semantics to the compiler, which is hidden from the students. Any errors that occur during the compiler's processing of the text version of the program are mapped back into the tree for reporting to the students.

Type checking in Pascal makes the implementation of incremental static semantic checking more difficult than that required in Karel. However, since Pascal provides only compile-time types, all checking can be done before execution. Support for a language with run-time types is possible only if the editor also controls execution of the program.

C. *SMILE*

SMILE, a production-quality internal development tool of the Gandalf project, is an environment that aids in

source code control and project management. Although SMILE presents a more traditional user interface than GP or GNOME, it nonetheless represents a contribution in the area of software development and retention of well-defined state in a developing software system.

SMILE, through a command-oriented interface, supports three major activities. The first activity is the definition and manipulation of module interconnections. Modules, as a whole, are made visible or are hidden from other modules in the system. Modules may only import facilities that are exported from visible modules. A smaller grain size of interconnection is available through commands that import or export particular facilities—C functions, data types, and objects. The description of the interconnections cannot be manipulated in one piece but instead evolves as individual SMILE commands are applied. Commands for viewing the structure are provided by SMILE. The second activity is control of public and experimental versions of the project database. At any time, each programmer on a project may have a set of modules reserved, in the GP sense. In essence, each programmer has an experimental database that consists of modifiable versions of the reserved modules and read-only versions of the other modules of the system. SMILE's third activity is support for implicit compilation. The commands used to modify objects designate whether they are modifying specifications or implementations of facilities. As in GP, the system automatically derives, from the module interconnection descriptions, what compilations need be applied after a given modification. The user is given commands to control automatic compilation, if desired.

SMILE also supports three subenvironments for programming-in-the-small—ALOEGEN, ARL, and DBGEN. The function of ALOEGEN (assisting the implementor in defining the abstract and concrete syntax of a target environment) was described in Section II-B. The ARL environment helps the implementor to describe the run-time support that will be available in the target environment, while DBGEN assists in defining the static semantics in terms of attributes.

ARL is the Action Routine Language in which the implementor can write the necessary run-time support for the target environment. An Action Routine, or Daemon, is essentially a procedural record field attached to the operator types described in Section II-B. When the user operates on the database in the target environment, an action routine is automatically activated when an object to which it is attached is visited. The invoked action depends on the particular operations applied, such as CREATE, DELETE, or ENTER. The ARL language in which action routines are described is a tree manipulation language specifically designed for the database model of Gandalf environments.

DBGEN defines the way in which the implementor can declare attributes and attribute equations that describe the static semantics of a target environment. It also allows the implementor to define operations on objects in the tree structured database that can be added to the standard set of user commands. One can think of providing the user with operations such as swap two objects of a list, put procedure declarations in alphabetic order, count the number of occurrences of a name in a subtree, etc. The usefulness of the target environment in supporting a user community can be increased considerably by adding a well-chosen set of extended commands.

SMILE is currently used as the development environment for ALOE, GP, GNOME, SMILE itself, and several other projects at CMU and elsewhere.

IV. Current Research

In building task-oriented programming environments, the implementor has to deal with syntax, semantics, and expertise. Abstract syntax determines the structure of objects that can exist in a target user environment and specifies the possible connectivity of these objects. Concrete syntax describes the various representations of these objects as they will appear as output. Static semantics determines the consistency of a collection of objects according to naming and scoping rules. Dynamic semantics is the run-time support that is needed to record the effect of a sequence of user actions. Rules of semantics basically determine when an environment and the collection of objects in it is in a correct state. A particular purpose of implementing semantic checks is to report errors that occur when the environment gets into an ill-defined state. In contrast to semantics, expertise goes beyond applying correctness criteria and is able to distinguish a good design from a bad one. In a program development environment, for instance, a check for undeclared variables is a matter of semantics but observing, for example, that a program is unnecessarily inefficient is a matter of expertise.

How to deal with syntax is basically a solved problem. The description tools for abstract and concrete syntax are effective, particularly in the ALOEGEN environment that helps designers construct these descriptions interactively. Some problems remain in making the user interface pleasing and fast. We want, in particular, to extend the feature of selective representation of environment objects to a facility that allows the designers to define different views of an object type. These views are not so much determined by the various ways in which the user wants to look at collections of data objects, but more by the way that each individual tool perceives the database. The environment is then assembled by a synthesis of the various tool views. This mechanism will replace the current primitive representation of unparsing schemes.

It is fair to say that the technology for dealing with syntax and semantics has been developed in sufficient detail to be useful. The issue of syntax has been investigated in several projects during the early 1980's [13], [19], [24], [40], [44], [62], while matters of semantics have been mastered in more recent years [7], [31], [50], [52], [32].

How to deal with expertise in programming environments is still an open question. A solution may be derived from the successful work that is going on in AI on this topic. Instead of paying attention to this problem now, we

decided first to make more headway with the description of semantics and tackle another problem—that of distributed systems. When designing a nontrivial software system, programmers will work on their personal workstations and share system modules on a network. The Gandalf system should provide features for a designer to define safe ways of sharing information and updating parts of the database. The approach currently being pursued makes use of the extensive research on atomic transactions in database management.

In all the research conducted in the Gandalf project, the emphasis is very strongly on the descriptive generic approach. Modifying descriptions is much easier and more reliable than modifying programs. At the same time, the generic approach makes it possible to experiment and generate revised versions of an environment in a matter of hours and days instead of the weeks and months it takes to modify handwritten software systems.

Acknowledgment

The Gandalf project has been possible due to the cooperation of many people including B. Denny, B. Ellison, P. Feiler, D. Garlan, G. Kaiser, C. Krueger, R. Medina-Mora, D. Perry, S. Popovich, and B. Staudt. Members of the GNOME project, including P. Miller, L. Miller, and R. Chandhok also deserves thanks. In addition to the helpful comments of our Gandalf associates and the referees, the following people graciously made detailed comments on the paper: E. Borison, M. Donner, H. Gayle, P. Hayes, S. Minton, L. Rudolph, and M. Shaw.

References

[1] C. N. Alberga, A. L. Brown, G. B. Leeman, M. Mikelsons, and M. N. Wegman, "A program development tool," *IBM J. Res. Develop.*, vol. 28, pp. 60–73, Jan. 1984.

[2] J. E. Archer, R. Conway, and F. B. Schneider, "User recovery and reversal in interactive systems," *ACM Trans. Program. Lang. Syst.*, vol. 6, pp. 1–19, Jan. 1984.

[3] F. T. Baker, "Structured programming in a production programming environment," *IEEE Trans. Software Eng.*, vol. SE-1, June 1975.

[4] F. P. Books, Jr., *The Mythical Man-Month: Essays in Software Engineering*. Reading, MA: Addison-Wesley, 1975.

[5] J. N. Buxton and L. E. Druffel, "Rationale for Stoneman," in *Proc. 4th Int. Comput. Software and Applications Conf.*, Oct. 1980, pp. 66–72. Reprinted in *Interactive Programming Environments*, D. R. Barstow, H. E. Shrobe, and E. Sandewall, Eds. New York: McGraw-Hill, 1984, pp. 535–545.

[6] R. H. Campbell and P. A. Kirslis, "The SAGA project: A system for software development," in *Proc. ACM SIGSOFT/SIGPLAN Software Eng. Symp. Prac. Software Develop. Env.*, Apr. 1984, pp. 73–80.

[7] R. Chandhok, D. Garlan, D. Goldenson, M. Tucker, and P. Miller, "Structure editing-based programming environments: The GNOME approach," in *Proc. NCC 85*, July 1985.

[8] R. Conway, D. DeJohn, and S. Worona, "A user's guide to the COPE programming environment," Dep. Comput. Sci., Cornell Univ., Ithaca, NY, Tech. Rep. 84-599, Apr. 1984.

[9] A. Demers, T. Reps, and T. Teitelbaum, "Incremental evaluation for attribute grammars with applications to syntax-directed editors," in *Conf. Rec. 8th Annu. ACM Symp. Principles of Program. Lang.*, Jan. 1981.

[10] F. DeRemer and H. Kron, "Programming-in-the-large versus programming-in-the-small," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 80–86, June 1976.

[11] N. M. Deslisle, D. E. Menicosy, and M. D. Schwartz, "Viewing a programming environment as a single tool," in *Proc. ACM SIGSOFT/SIGPLAN Software Eng. Symp. Prac. Software Develop. Eng.*, Apr. 1984, pp. 49–56.

[12] T. A. Dolotta, R. C. Haight, and J. R. Mashey, "UNIX time-sharing system: The programmer's workbench," *Bell Syst. Tech. J.*, vol. 57, part 2, July–Aug. 1978. Reprinted in *Interactive Programming Environments*, D. R. Barstow, H. E. Shrobe, and E. Sandewall, Eds. New York: McGraw-Hill, 1984, pp. 353–369.

[13] V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang, and J. J. Levy, "A structure oriented program editor: A first step towards computer assisted programming," INRIA Tech. Rep. 114, Apr. 1975.

[14] V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang, "Programming environments based on structure editors: The mentor experience," INRIA Tech. Rep. 26, May 1980. Reprinted in *Interactive Programming Environments*, D. R. Barstow, H. E. Shrobe, and E. Sandewall, Eds. New York: McGraw-Hill, 1984, pp. 128–140.

[15] V. Donzeau-Gouge, G. Kahn, B. Lang, and B. Melese, "Documents structure and modularity in Mentor," in *Proc. ACM SIGSOFT/SIGPLAN Software Eng. Symp. Prac. Software Develop. Env.*, Apr. 1984, pp. 141–148.

[16] R. J. Ellison and B. J. Staudt. "The evolution of the Gandalf system," *J. Syst. Software*, vol. 5, pp. 107–119, May 1985.

[17] G. Engels, C. Lewerentz, M. Nagl, and W. Schafer, "On the structure of an incremental and integrated software development environment," in *Proc. 19th Annu. Hawaii Int. Conf. Syst. Sci.*, 1986.

[18] P. H. Feiler, "A language-oriented interactive programming environment based on compilation technology," Ph.D. dissertation, Dep. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, 1982.

[19] P. H. Feiler and G. E. Kaiser, "Display-oriented structure manipulation in a multi-purpose system," in *Proc. IEEE Comput. Soc. 7th Int. Comput. Software and Applicat. Conf. (COMPSAC 83)*, Nov. 1983.

[20] S. I. Feldman, "Make—A program for maintaining computer programs," *Software Practice and Experience*, vol. 9, Apr. 1979.

[21] C. N. Fisher, G. F. Johnson, J. Mauney, A. Pal, and D. L. Stock, "The POE language-based editor project," in *Proc. ACM SIGSOFT/SIGPLAN Software Eng. Symp. Prac. Software Develop. Env.*, Apr. 1984, pp. 21–29.

[22] P. Fritzson, "Preliminary experience from the DICE system, a distributed incremental compiling environment," in *Proc. ACM SIGSOFT/SIGPLAN Software Eng. Symp. Prac. Software Develop. Env.*, Apr. 1984, pp. 113–123.

[23] ——, "Towards a distributed programming environment based on incremental compilation," Ph.D. dissertation, Dep. Comput. and Inform. Sci., Linkoping Univ., 1984.

[24] E. Gansner, J. R. Horgan, D. J. Moore, P. T. Surko, and D. E. Swartwout, "SYNED—A language-based editor for an interactive programming environment," in *Dig. Papers Spring CompCon '83*, IEEE Comput. Soc., Nov. 1982.

[25] D. B. Garlan and P. L. Miller, "GNOME: An introductory programming environment based on a family of structure editors," in *Proc. ACM SIGSOFT/SIGPLAN Software Eng. Symp. Prac. Software Devel. Env.*, Apr. 1984, pp. 65–72.

[26] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*. Reading, MA: Addison-Wesley, 1983.

[27] A. Goldberg, *Smalltalk-80: The Interactive Programming Environment*. Reading, MA: Addison-Wesley, 1984.

[28] A. N. Habermann and D. Perry, "Well-formed system compositions," Dep. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, Tech. Rep. CMU-CS-80-117, Mar. 1980.

[29] R. Hood and K. Kennedy, "A programming environment for Fortran," in *Proc. 18th Annu. Hawaii Int. Conf. Syst. Sci.*, 1985.

[30] J. R. Horgan and D. J. Moore, "Techniques for improving language-based editors," in *Proc. ACM SIGSOFT/SIGPLAN Software Eng. Symp. Prac. Software Develop. Env.*, Apr. 1984, pp. 7–14.

[31] G. Kahn, B. Lang, B. Melese, and E. Morcos, "Metal: A formalism to specify formalisms," INRIA Tech. Rep., 1982.

[32] G. E. Kaiser, "Semantics for structure editing environments," Ph.D. dissertation, Dep. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, 1985.

[33] G. E. Kaiser and A. N. Habermann, "An environment for system version control," in *Dig. Papers Spring Compcon '83*, IEEE Comput. Soc., Nov. 1982.

[34] G. E. Kaiser and P. H. Feiler, "Generation of language-oriented editors," in *Programmierumgebungen und Compiler*, German Chapter of the ACM, Apr. 1984.

[35] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*. Englewood Cliffs, NJ: Prentice-Hall Software Series, 1978.

[36] B. W. Kernighan and J. R. Mashey, "The UNIX programming environment," *Computer*, vol. 14, pp. 25–34, Apr. 1981. Reprinted in *Interactive Programming Environments*, D. R. Barstow, H. E.

Shrobe, and E. Sandewall, Eds. New York: McGraw-Hill, 1984, pp. 175–197.
[37] P. A. Kirslis, R. B. Terwilliger, and R. H. Campbell, "The SAGA approach to large program development in an integrated modular environment," in *Proc. GTE Workshop Software Eng. Env. for Programming-in-the-Large*, June 1985.
[38] B. W. Lampson and E. E. Schmidt, "Organizing software in a distributed environment," in *Proc. SIGPLAN '83 Symp. Program. Lang. Issues in Software Syst.*, June 1983, pp. 1–13.
[39] R. Medina-Mora and P. Feiler, "An incremental programming environment," *IEEE Trans. Software Eng.*, vol. SE-7, pp. 472–482, Sept. 1981.
[40] R. Medina-Mora, "Syntax-directed editing: Towards integrated programming environments," Ph.D. dissertation, Dep. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, 1982.
[41] R. Medina-Mora, D. Notkin, and R. Ellison, "Aloe users' and implementors' guide," in *Second Compendium of Gandalf Documentation*, Dep. Comput. Sci., Carnegie-Mellon Univ., May 1982.
[42] M. Mikelsons and M. N. Wegman, "PDE1L: The PL1L programs development environment (principles of operation)," Dep. Comput. Sci., IBM T. J. Watson Research Center, Yorktown Heights, NY, Tech. Rep. RC 8513, Sept. 1980.
[43] D. Notkin, "Interactive structure-oriented computing," Ph.D. dissertation, Dep. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, 1984.
[44] ——, "The Gandalf project," *J. Syst. Software*, vol. 5, pp. 91–106, May 1985.
[45] L. J. Osterweil, "Toolpack—An experimental software development environment research project," *IEEE Trans. Software Eng.*, vol. SE-9, pp. 673–685, Nov. 1983.
[46] R. E. Pattis, *Karel the Robot: A Gentle Introduction to the Art of Programming*. New York: Wiley, 1981.
[47] S. P. Reiss, "PECAN: Program development Systems that support multiple views," in *Proc. 7th Int. Conf. Software Eng.*, IEEE Comput. Soc., Mar. 1984.
[48] ——, "Graphical program development with PECAN program development systems," in *Proc. ACM SIGSOFT/SIGPLAN Software Eng. Symp. Prac. Software Develop. Env.*, Apr. 1984, pp. 30–41.
[49] T. Reps, "The synthesizer editor generator: Reference manual," Dep. Comput. Sci., Cornell Univ., Ithaca, NY, 1981.
[50] ——, "Optimal-time incremental semantic analysis for syntax-directed editors," in *Conf. Rec. 9th Annu. ACM Symp. Principles of Program. Lang.*, Jan. 1982.
[51] ——, "Static-semantic analysis in language-based editors," in *Dig. Papers Spring CompCon '83*, IEEE Comput. Soc., Nov. 1982.
[52] ——, *Generating Language-Based Environments* (ACM Doctoral Dissertation Award Series). Cambridge, MA: M.I.T. Press, 1983.
[53] T. Reps, T. Teitelbaum, and A. Demers, "Incremental context-dependent analysis for language-based editors," *ACM Trans. Program Lang. Syst.*, vol. 5, pp. 449–477, July 1983.
[54] T. Reps and T. Teitelbaum, "The synthesizer generator," in *Proc. ACM SIGSOFT/SIGPLAN Software Eng. Symp. Prac. Software Develop. Env.*, Apr. 1984, pp. 42–48.
[55] M. Rochkind, "The source code control system," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 364–370, Dec. 1975.
[56] R. Snowden, "An experience-based assessment of development systems," in *Software Development Tools*, W. E. Riddle and R. E. Fairley, Eds. New York: Springer-Verlag, 1980.
[57] T. A. Standish and R. N. Taylor, "Arcturus: A prototype advanced Ada programming environment," in *Proc. ACM SIGSOFT/SIGPLAN Software Eng. Symp. Prac. Software Develop. Env.*, Apr. 1984, pp. 57–64.
[58] R. E. Sweet, "The Mesa programming environment," in *Proc. ACM SIGPLAN 85 Symp. Lang. Issues in Program. Env.*, July 1985, pp. 216–229.
[59] D. C. Swinehart, P. T. Zellweger, and R. B. Hagmann, "The structure of Cedar," in *Proc. ACM SIGPLAN 85 Symp. Lang. Issues in Program. Env.*, July 1985, pp. 230–244.
[60] W. Teitelman, "A tour through Cedar," *IEEE Software*, vol. 1, Apr. 1984.
[61] W. Teitelman and L. Masinter, "The Interlisp programming environment," *Computer*, vol. 14, pp. 25–34, Apr. 1981. Reprinted in *Interactive Programming Environments*, D. R. Barstow, H. E. Shrobe, and E. Sandewall, Eds. New York: McGraw-Hill, 1984, pp. 83–96.
[62] T. Teitelbaum and T. Reps, "The Cornell Program Synthesizer: A syntax-directed programming environment," *Commun. ACM*, vol. 24, pp. 563–573, Sept. 1981. Reprinted in *Interactive Programming Environments*, D. R. Barstow, H. E. Shrobe, and E. Sandewall, Eds. New York: McGraw-Hill, 1984, pp. 97–116.
[63] T. Teitelbaum, T. Reps, and S. Horwitz, "The why and wherefore of the Cornell Program Synthesizer," in *Proc. ACM SIGPLAN/SIGOA Symp. Text Manipulation*, June 1981, pp. 8–16.
[64] W. Tichy, "Design, implementation, and evaluation of a revision control system," in *Proc. 6th Int. Conf. Software Eng.*, Sept. 1982.
[65] U.S. Dep. Defense, "Reference manual for the Ada programming language," Rep. ANSI/MIL-STD-1815A, Jan. 1983.
[66] N. Wirth, *Programming in Modula-2*. New York: Springer-Verlag, 1983.
[67] *Unix Programmer's Manual*, 7th ed., Division Comput. Sci., Dep. Elec. Eng. and Comput. Sci., Univ. California at Berkeley.

A. Nico Habermann received the M.S. degree in mathematics from Free University, Amsterdam, The Netherlands, and the Ph.D. degree in applied mathematics from the Technological University, Eindhoven, The Netherlands, in 1967.

He worked with Dr. E. W. Dijkstra on the THE system for which he wrote the Algol60 interface. After visiting Carnegie-Mellon University, Massachusetts Institute of Technology, DEC, and IBM, he came to CMU as an Associate Professor in 1969. He was promoted to Full Professor in 1974 and became Department Head of Computer Science in 1979. He was instrumental in establishing the Software Engineering Institute at CMU and was the Acting Director of the Institute in 1985. His main interests are in programming languages, operating systems, software engineering, and programming environments. He has worked on language design and implementation for Algol60, Bliss, Pascal, Ada, and various special purpose languages. He has worked on several practical and experimental operating systems such as the THE system, the Family of Operating Systems (FAMOS), the Dynamically Adaptable System (DAS), and UNIX. He has written two books: one on operating system design and one of the Ada language (with Dr. Perry). Some of his best known contributions to the field are a critique on the Pascal language, work on deadlock prevention, path expressions (with Dr. Campbell), an efficient implementation of Ada tasking (with Dr. Nassi) and the integrated approach to software development, which is demonstrated by the Gandalf project. He has served on numerous program committees and consults for several computer firms. He spent a year at the University of Newcastle upon Tyne, England (1973), at the Technological University of Berlin, Germany (1976), and at Siemens Corporation in Munich, Germany (1983).

Dr. Habermann is a member of IBM's Scientific Advisory Committee, of the Advisory Committee for Computer Science of the National Science Foundation, and is an editor for *Acta Informatica* and the *ACM Transactions on Programming Languages and Systems* (TOPLAS).

David Notkin (S'77–M'82) received the Sc.B. degree in computer science from Brown University, Providence, RI, in 1977, and the Ph.D. degree in computer science from Carnegie-Mellon University, Pittsburgh, PA, in 1984.

He has been an Assistant Professor in the Department of Computer Science at the University of Washington, Seattle, since September 1984. His current research interests include: software development environments; adaptable, flexible extendable, and customizable systems; programming-in-the-large; version control; structure-oriented editors; interactive programs; and program generation.