A QUICK OVERVIEW OF THE GANDALF SYSTEM

A.N. Habermann

Rapporteurs: Mr. L. Mancini
Mr. A.M. Koelmans

1. Introduction

The purpose of this paper is to give the potential user of the Gandalf environment generation system a quick overview of what is available and how the system is to be used. The paper serves as an introduction to the tutorial and the reference manuals. It does not address the research issues that were explored in the Gandalf project, nor does it describe the development history of the project. Recent work in the project has been reported in a special issue of the Journal for Software and Systems [JSS], which also cites most of the preceding work.

The Gandalf system is a workbench for the creation and development of interactive programming environments. The system consists of several components that an implementor[1] uses for designing and fine tuning a user environment with task-specific tools and facilities. Most of the environments produced with the Gandalf system so far are intended to support software system development. For example, the Gandalf Prototype environment [GP82] provides assistance for program development, system version control and project management.

User environments are generated with the Gandalf system following the scheme depicted below.

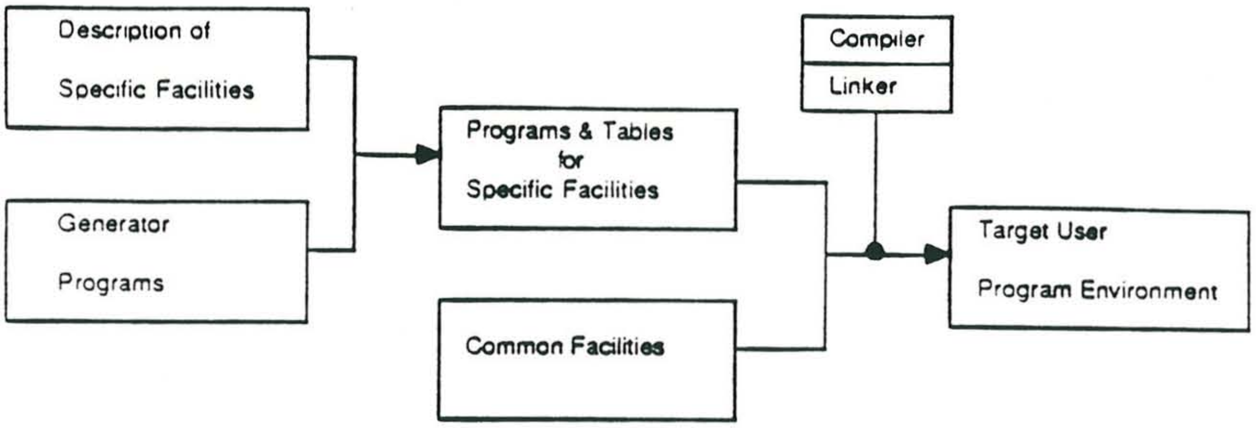
Description of Specific Facilities
Generator Programs
Programs & Tables for Specific Facilities
Common Facilities
Compiler
Linker
Target User Program Environment

Fig. 1 The Gandalf System Generation Scheme

The common facilities form the core of a wide range of target environments. These facilities are general enough that parameterization suffices for environment-specific design. The three major services provided by the common facilities are:

- operating system and file system interface

[1]We refer to the designer, builder, and maintainer of an environment as an *implementor*.

- user interface, including terminal I/O, cursor motion and help
- database management for the representation of all information in the user environment.

The common facilities constitute a large part of a target user environment. The Gandalf system alleviates the implementor's job by providing the code for these common facilities, requiring the implementor to be concerned only with providing the description of specific facilities.

In order to facilitate fine-tuning and variation, the process of generating the specific facilities for a target environment has also been automated to a large degree. An implementor does not write the specific facilities directly in C, but instead writes a concise high level specification which is transformed into C programs and tables by a generator program. These specifications are much easier to modify and expand correctly than ordinary C code. The implementor must be familiar with the formalism in which these specifications are written, but needs no insight into how the generator program works. The current Gandalf System runs on UNIX[2] UCB 4.2 and is written in C. User environments produced with the Gandalf system do depend on UNIX, but not on C. The functionality designed into user environment is determined by the support needs of the task to be performed and not by the language in which this support is represented. Environments produced with the Gandalf system can support any language the environment builder chooses.

2. The Implementor's Task

The Gandalf environment generation scheme is based on the model of generic data structures. The essence of that model is that one can define the format of a structure and the operations on it without specifying the actual type of the data elements that will be placed in that structure. Well-known examples are generic definitions for Btrees, queues, stacks, etc. The stack operations push and pop, for instance, can be defined for all stacks without being specific about the type of the data objects that will be placed in a stack. The common facilities of the Gandalf system (the lower middle box in Fig. 1) define a generic tree as the general database structure for every target user environment. That is, the objects created in a Gandalf user environment will be stored in a tree-structured database, whose nodes can be single objects (atomic or structured) or can be lists of such objects. The typical primitive operations defined on a tree structure are insertion and removal of individual elements or subtrees, tree traversal and comparison tests.

The essential role of the implementor in Gandalf's generic scheme is to produce a description of the actual types of the objects that will be created in the target user environment. These type

[2]UNIX is a registered trademark of Bell Laboratories, N.J.

descriptions determine the kind of objects a user will be able to put in the generic database and how the objects will be displayed to the user. The description of these data types corresponds to the specification of the specific facilities (the left upper box in Fig. 1).

A description of the specific facilities consists basically of two parts, each divided into two subparts:

- syntax
  - abstract syntax
  - concrete syntax
- semantics
  - static semantics
  - runtime support

The *abstract syntax* is a BNF-like description that determines the logical structure of the data objects for the user environment and also defines their relationships. For example, if we were to design a user environment for electronic mail, we would specify the structure of mailboxes and mailmessages and we would express the fact that a mailbox can hold a number of messages. The *concrete syntax* determines the textual representation of the data objects in the target user environment and how those objects are displayed on the user's terminal screen. The concrete syntax drives the process called *unparsing* which maps the tree representation of data objects into text. A major feature provided in the Gandalf system for describing concrete syntax is "multiple unparsing schemes" which means that the implementor can specify more than one textual representation for an object. For example, the implementor could specify two unparsing schemes for mailboxes, one that shows the entire mailbox and one that shows all the message headers while leaving out the text.

Static semantics describes the consistency rules that apply to the objects in the user environment. A typical example of static semantics for programming languages is that a variable declaration must precede the use of that variable in an expression or statement. An example of static semantics that might be specified for the mailbox example is that the name of a sender must be known to the system or that the date of a message cannot refer to some day in the future.

Runtime support is sometimes called dynamic semantics. It concerns the particular collection of object values in the user environment that depends on the history of the user's interactions with the environment. A typical example for a compiler based system is the propagation of changes that determine whether or not particular modules must be recompiled. Another example is the instantaneous notification of the arrival of a mail message for logged-on users. In all cases, dynamic semantics involves the particular development history of the task performed in the user's environment.

3. Gandalf System Components

The Gandalf system provides a collection of facilities that the implementor uses at various times for the description and generation of user environments. These facilities are grouped in five system components:

- the modular version control environment, SMILE
- the syntax development environment, ALOEGEN
- the semantic description formalism, ARL, with its standard library
- the interface description generator, DBGEN
- the ALOE kernel, providing the common database and I/O facilities.

SMILE supports programming-in-the-large. It is a multi-implementor, multi-module environment that provides facilities for access control, version control and automatic recompilation. ALOEGEN is an environment for programming-in-the-small produced by applying the Gandalf ideas to its own design. It assists the implementor with generating the description of abstract and concrete syntax. ARL is an imperative tree-oriented programming language that is used for semantic analysis and primitive operations on data objects in a database. ARL is used in conjunction with an extensive library of routines to access the user interface, I/O, and file system. DBGEN fine tunes the user interface and other editor functions unrelated to the database. The ALOE kernel contains the implementation of the database primitives and of the terminal I/O routines. It also contains the standard command interpreter for the user environment and the mechanism for activating the runtime support routines. Each of the five Gandalf system components is described in more detail in subsequent sections.

4. SMILE

SMILE is the environment in which an implementor designs, builds, and maintains target user programming environments (for details see the SMILE user's manual in the GANDALF System Reference Manuals [GSRM]). SMILE maintains all the information about a target environment and all the descriptions an implementor has written in a *SMILE database*. A SMILE database is partitioned into a collection of modules which can be used to encapsulate information. SMILE provides a mutual exclusion mechanism for situations when more than one implementor is working on an environment.

This guarantees that changes made by one implementor will not interfere with the work of others. For each environment being built with SMILE there will be a common pool of modules and a number of local user workspaces as depicted in Fig. 2.

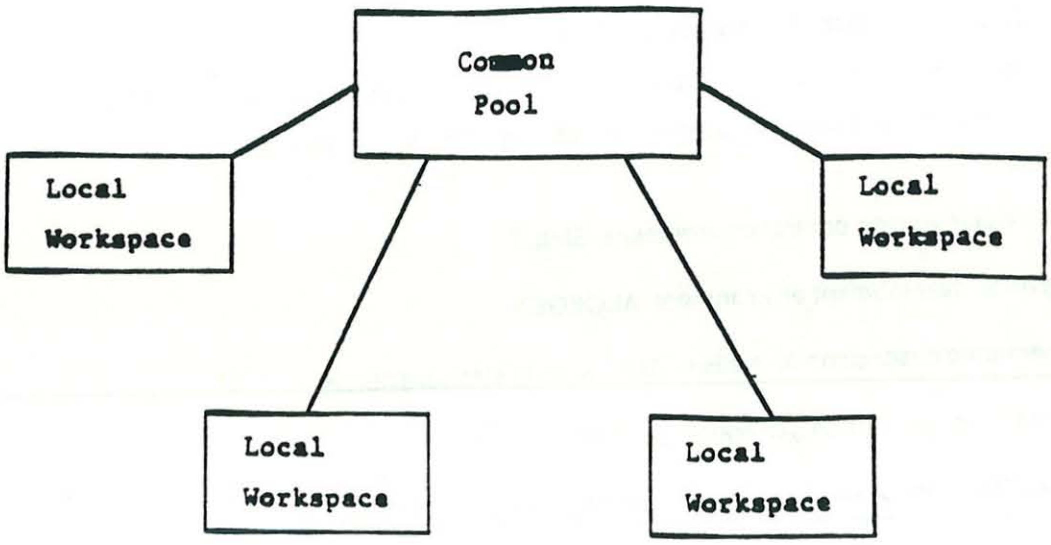

Fig. 2 The SMILE Development Environment

The common pool contains all the shared module versions of an environment design project. All implementors have access to the shared modules, but at most one implementor at a time has modification rights to a particular module at a time. When an implementor gets modification rights to a shared module, that module is frozen in the common pool and a copy is made available in the implementor's local workspace. The frozen state gives other implementors read or compile access, but forbids replacement until the implementor who gained write access either replaces the module or explicitly relinquishes the right to modify.

The components an implementor creates in a SMILE module are typically of three kinds:

- ALOE grammars for the syntax descriptions
- ARL routines for the semantics descriptions
- DB grammars for fine tuning environment behavior

When the implementor decides to create or modify a component, SMILE invokes the appropriate programming-in-the-small environment for that type of component.

SMILE provides automatic source and object code control and system configuration. Whenever a component in a SMILE database is modified, that component and all those which depend on it will be recompiled in a background process. At the request of an implementor, SMILE will automatically configure the link of an executable target environment using the appropriate modules from the common pool and possibly local versions of frozen modules being modified.

SMILE was originally conceived as a programming environment for C with automatic recompilation and support for programming-in-the-large. It was later extended to a complete environment for the Gandalf system that the implementor can use for designing the syntax and semantics of programming environments. The option of writing C program modules is no longer of great importance to the implementors of Gandalf environments.

5. ALOEGEN

The syntax of a user environment is described in an ALOE grammar that defines the data types of the projected user environment. The task of producing such a description is supported by the syntax development environment, ALOEGEN. The interface of ALOEGEN is a structure editor that provides three kinds of structures at the top level: *terminals*, also called *atoms*, *non-terminals*, also called *operators*, and *classes*. The implementor can instruct ALOEGEN to create one of these structures or can enter an existing structure for further editing.

The terminal productions typically describe the primitive objects for the user environment such as identifiers, numbers and symbols. The implementor can attach lexical routines to these terminals, although ALOEGEN provides default lexical routines for commonly used terminals. Examples are found in the ALOEGEN user's manual in the GANDALF System Reference Manuals [GSRM].

The non-terminal productions describe the structured object types in terms of object classes. For example, a mail message is described by the non-terminal

```
MESSAGE        =        sender  date  subject  text
```

where sender, date, subject and text are classes of terminals and non-terminals. The class "date", for instance, may include dates in numeric form and dates in literal form. The class "text" might consist of the options STRING and FILE. A class acts as a placeholder for a set of types that may be substituted as a component of a non-terminal, allowing the composition of different types into a single structured type in a more elegant manner than variant records do. (The mechanism resembles that of ALGOL68 where non-terminals correspond to modes and classes to unions.)

The example description of a non-terminal production above shows the abstract syntax, but omits all other parts, the concrete syntax among them. In the current Gandalf system, concrete syntax is represented by strings, called unparsing schemes, written in a small representation language that allows the implementor to indicate punctuation and format. For instance, two alternative concrete syntax descriptions for mail messages are

```
[0] from: @1@Ndate: @2@Nsubject: @3@N@4@N

[1] subject(@1 on @2) = @3
```

Each two character sequence beginning with an '@' is a formatting command. The numbers in the unparsing scheme refer to the components in the abstract syntax definition. The symbol @N stands for "newline", while the identifiers and other symbols are taken literally. The first unparsing scheme prints the entire message in full and separates the successive parts by a newline symbol. The second unparsing scheme places an abstracted form of the message header on a single line. It uses a procedure call format with sender and date as arguments. For instance, if Jane Smith sent a message about terminal connections on April 10, 1986, the second unparsing scheme would display the message as

```
subject (Jane Smith on 10 Apr 86) = Terminal Connections
```

Further details about syntax and additional features such as synonyms and separate windows are found in the ALOEGEN user's manual in the GANDALF System Reference Manuals [GSRM].

6. ARL: Language and Library

Semantics, both static and dynamic, are described in the Action Routine Language, ARL. ARL allows the implementor to declare operations that can be attached to terminal and non-terminal productions. The attached operations are called Action Routines or Daemons. Attaching a daemon to a production in ALOEGEN is accomplished by including its name in a special field in that production. Daemons can be considered as record fields that are not directly accessible to the user.

The concept of a daemon is something not found in traditional languages such as Pascal or Ada. Record fields in those languages are restricted to data fields, and exclude the possibility of declaring procedures or functions as record fields. Daemons transform objects from passive to active data. The ALOE kernel triggers a daemon attached to an object type whenever an object of that type is created, modified, or visited in the user environment. The effect of executing a daemon depends on the kind of database operation performed by the user. A daemon must, for instance, do different things when a new object is created than when one is deleted or simply visited.

A daemon is essentially a case statement that selects on the particular database operation performed on the object to which the routine is attached. Its general format is

```
<declarations>
    ...
case CREATE:
    ...
    ...
case DELETE:
    ...
    ...
case ENTER:
    ...
<other cases>
    ...
    ...
```

The database operations are sent to a daemon as an event signal. The major database operations serve as predefined events that are recognized by the ALOE kernel. In addition to these predefined events, the implementor can also define his/her own signals and program daemons to be activated when implementor-defined events occur. Details are found in the ALOE Action Routine Language Manual in the GANDALF System Reference Manuals [GSRM].

Daemons are written separately with the support of the ARL environment for programming-in-the-small. This environment is entered automatically when the implementor indicates to SMILE the wish to write a daemon (see Fig. 3).

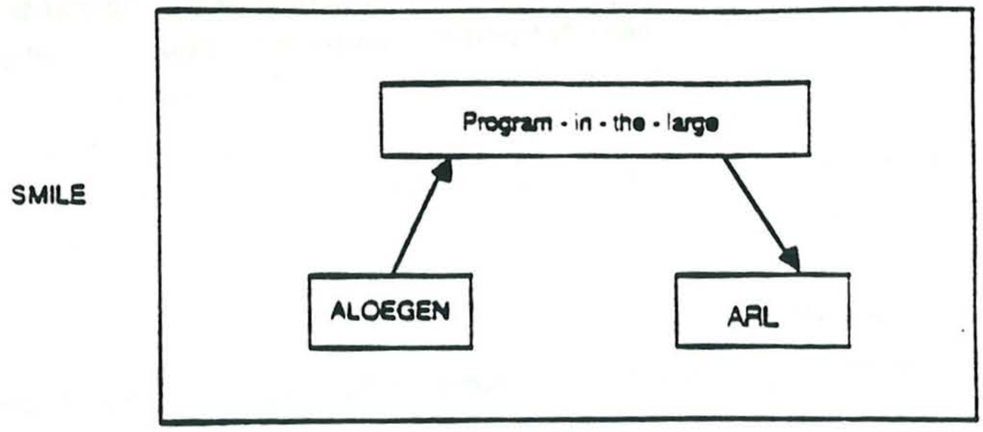

Fig. 3 Transition from Syntax to Semantics

ARL is a tree-oriented language which operates on the data trees created in the user environment. The syntax of ARL is similar to languages such as C and Pascal, but instead of operating on data objects such as records and arrays, ARL operates on abstract syntax trees. Typical statements in

ARL modify, examine, and create nodes and subtrees. Daemons written in ARL can monitor and examine programs being written by the user, and they also have equal status to the user in terms of editing operations. Daemons are, therefore, a very powerful mechanism for implementing intelligent behavior into a programming environment.

ARL is used in conjunction with an extensive library. While the ARL language is concerned with operations and state of abstract syntax trees, the ARL library controls the state of the editor. For example, the library contains primitives for displaying multiple windows, pop-up menus and error messages to the user.

In the current version of the Gandalf system, the daemons written in ARL are procedural. The plan is to replace the imperative style of ARL by a declarative language, ACL, in which the implementor can describe Assertions and Constraints on attributes. ACL is described in G. Kaiser's thesis [GK].

7. DBGEN

While ALOEGEN and ARL describe the syntax and semantics of an environment, DBGEN is used to parameterize the portions of the ALOE kernel that deal with user interface and other environment issues unrelated to the abstract syntax tree. For example, DBGEN can be used to specify extended commands. It is not uncommon that the implementor wants to extend the basic set of predefined database operations with user commands specific for the user environment. The mail system, for instance, could be extended with a command that gives the number of messages in a mailbox or that selects all messages in a mailbox that originated with a particular sender. Extended commands are written in ARL and typically make use of the ARL library. The procedure for defining extended commands and other DBGEN extensions is explained in the tutorial on using the New Gandalf system [NGS] and the DBGEN user's manual in the Gandalf System Reference Manuals [GSRM].

8. The ALOE Kernel

Neither user nor implementor has to know much about the ALOE kernel or about the generator program that generates C programs and tables from the grammar descriptions. These parts play an important role in the Gandalf system, but are invoked automatically when needed. The main parts of the ALOE kernel are

- the command interpreter
- the database operations
- the signal propagation mechanism for daemons

- the unparsing scheme interpreter
- I/O and file system access

The command interpreter is not an incremental parser, but a menu-driven template editor. Each menu corresponds to a particular class defined as part of the implementor's grammar description. The command interpreter accepts each piece of user input as a command and performs the necessary operations on the database.

Concluding Remarks

The purpose of the Gandalf system is to make it easy for implementors to design, modify and enhance programming environments. It is just a matter of a few days to generate a preliminary version of an environment that shows the objects the user can create and provides a simple user interface. This prototype can be demonstrated to its potential users, which gives the implementor a chance to take the user's feedback into account. Semantics and enhancement can then be developed incrementally over a longer period of time. The result is a flexible environment that can be tailored to the particular wishes of its users or to the particular requirements of a project.

References

[JSS]	"Special Issue on the Gandalf Project" The Journal of Systems and Software, 5, 2 (May 85)
[GP82]	Notkin, D. S. and G. E. Kaiser "The Implementation of the Gandalf Software Development Environment" Second Compendium of Gandalf Documentation Department of Computer Science, Carnegie-Mellon University (May 82)
[GSRM]	Krueger, C. W. "The Gandalf System Reference Manuals" Department of Computer Science, Carnegie-Mellon University (Jan 86)
[GK]	Kaiser, G. E. and C. W. Krueger "Using the *New* Gandalf System (a tutorial)" Department of Computer Science, Carnegie-Mellon University (Feb 86)

DISCUSSION

Dr. Harrison asked, have you talked about 'tools' in general, but does this not consist of many different sub-tools and should the differences between them not be taken into account in the construction of environments? This is not done in current systems and I feel you should be more specific about this. Professor Habermann answered, I tend to agree with you. I want to de-emphasize the notion of a tool as such; I want to emphasize the task to be performed and regard the tool as an implementation of the task.