Experiences with an environment generation system

Steven S. Popovich,[1] William M. Schell, and Dewayne E. Perry

AT&T Bell Laboratories, Murray Hill, NJ 07974

Abstract

We present our experience using the Gandalf environment generation system as a prototyping vehicle for the Inscape Environment. Our positive experience included experimentation, incremental evolution, multiple views, the coupling of semantic and editing actions, and the use of the domain-specific facilities. Our negative experience consisted primarily of problems with presentation and object management. On the whole, our experience was positive.

1. Introduction

This paper is about our experiences using a generation facility, specifically the Gandalf system [1], as a prototyping technique for the Inscape Environment [2]. It is not about experience with a syntax editor; Zelkowitz [3] treats this area, discussing a similar (specification-based environment) application. In this paper, we will try to avoid, as far as possible, discussing problems specific to syntax editing.

We first provide a general view of the Gandalf System and the Inscape Environment. We then present our positive and negative experiences. Finally, we summarize and evaluate our experiences.

2. The Gandalf System

Gandalf is a generation system for user environments based on structure editing. An environment is specified in terms of a grammar, which describes the language (such as a programming or specification language) and associates attribute data structures and semantic actions with the constructs of the language. Subsidiary grammars may be specified for complex attributes such as symbol tables. All of these grammars are entered into a grammar editor, which is itself generated using Gandalf, and are used by the system to generate a set of tables describing the language to the editing kernel.

A Gandalf-based environment consists of four parts: A structure editor kernel, which is simply linked into each executable, a set of grammar tables describing the language to the kernel in terms of its abstract syntax, one or more concrete syntax views, and a collection of action routines written in the extension language, ARL. These action routines are further divided into *daemons*, *functions*, and *procedures*, which implement semantic checking and other actions that occur as side effects of normal editing commands, and *extended commands*, which, true to their name, add commands to the editor that perform environment and application-specific actions. The ARL extension code is written using another Gandalf editor, configured with the syntax and static semantics of the high-level tree manipulation language. The ARL editor generates C code (about 3 lines of C for each line of ARL) from the tree-oriented ARL code, and the code is then linked with the kernel and the grammar tables to form a Gandalf editor.

3. The Inscape Prototype

The Inscape Environment is centered around two orthogonal concepts:

- the constructive use of formal module interface specifications; and

1. Department of Computer Science, Columbia University, New York NY 10027. The work reported here was done at AT&T Bell Laboratories during the summers of 1989 and 1990.

- the support and enforcement of cooperation by means of environmental structures, mechanisms and policies.

The first of these concepts is used to address the problems of system construction and evolution, both in terms of programming-in-the-small and programming-in-the-large. The second concept is used to address the problems of programming-in-the-many and has been build as a separate prototype [4].

A prototype of the environment has been built,[2] using Gandalf, that consists of the following components: A subset version of *Instress*, the module interface specification language (with a decidable subset of the underlying logic); a subset version of *Inform*, the module construction component (subsetted by focusing on generic programming language constructs, successful operation results, and simple object descriptions); and *Inquire* [5], the predicate-based browser and deductive retrieval mechanism. In addition, the syntactic version of the full specification language complete with various interactive and publication views has been implemented.

Inform uses Instress specifications as the basis for interactive, incremental analysis, using instantiated interfaces for function calls and generated interfaces for sequence, selection and iteration to generate function interfaces and to perform error detection [6]. Inquire uses these interfaces together with the symbol table information as the basis for browsing and deductive retrieval facilities for both the user and the system.

4. Positive Experience

There are a number of obvious advantages to using a system like Gandalf to generate the Inscape Prototype. These, in fact, were important factors in the original decision to use Gandalf as the prototyping base.

First and most important, it is possible to build a working prototype very quickly and to add functionality to it incrementally. To generate an editor for a programming language or a specification language, it is necessary only to write its grammar. The Gandalf kernel provides a screen-oriented editor for syntax trees, and allows us to experiment with our environment's presentation of data by modifying the unparsing schemes used in displaying the trees to the user and by changing the windowing specifications. Since Gandalf provided us with an editing and presentation layer rather than requiring us to construct one ourselves, we were able to concentrate on the important issues of syntax and semantics at a very early stage.

Second, Gandalf allows incremental addition of statements to the programming and specification languages, making it easy to extend them over time. The initial prototype had only function call and sequence as language statements. Extending the prototype to include both the syntax and semantics for selection and iteration was straightforward. Another useful facility in the Gandalf system is that of *Transformgen* [7] which builds transformations from the changes to the grammar in such a way that trees built under the previous version of the grammar can be automatically transformed into trees of the current version. This has been particularly useful in refining the definition of Instress and evolving the editor for it. Using Transformgen, the examples evolve along with the editor.

Third, Gandalf allows incremental addition of unparsing views. Multiple views enable us to display the syntax tree to the user in various ways: as code alone, code annotated with interfaces, specifications alone, or any of several other options. The most recently added unparsing view automatically generates a `troff` document of the specification using multiple fonts and indentation to emphasize and differentiate various aspects of the interface.

Fourth, it is very easy to tie semantic actions (such as checking that preconditions are satisfied) to editing actions (such as adding a statement to a function) in a Gandalf editor. It is simply a matter of writing and declaring a daemon (along with its supporting attributes, functions and procedures) for the appropriate node type.

Finally, Gandalf provides a high-level language (ARL) for tree editing, which further simplifies the implementation of semantic actions and extended commands. There are several advantages to using such a language. First, the domain-specific control and data structures make it easy to experiment with semantic processing and internal representations of the program tree. Second, the encapsulation facilities make it easy to

2. The primary work on the Instress specification editor and the Inscape Prototype has been done by Dewayne Perry; Steve Popovich revised the symbol table and implemented Inquire; Bill Schell implemented the SunWindows package as part of a prototype applying Inscape to the design of finite state machines.

experiment with the internal representations of such data structures as the symbol table. For example, we were able to completely reimplement the symbol table in a very short time and with minimal effects on the rest of the prototype. The daemon invocations remained the same, and only a small amount of additional tree walking was needed to collect the additional information needed by the added functionality in the newer symbol table package. Third, the abstraction facilities make it easy to reuse existing code. For example, we were able to reuse much of the predicate manipulation code used for predicate propagation in the Inform subsystem to manipulate the query predicates for deductive retrieval in the Inquire query subsystem.

5. Negative Experience

Although Gandalf's advantages as a generation system were substantial, we experienced problems with other aspects of Gandalf during the construction of the prototype. These divide into three general classes: dealing with the on-screen presentation of information to the user; dealing with the Gandalf's management of its data objects containing the program, the specifications, and other information; and dealing with the the lack of appropriate support tools in the underlying environment (*Smile* [8]). The presentation problems are mainly due to the age of Gandalf, and the object management problems are attributable to the fact that Gandalf was designed for language-specific editors, which do only a relatively small amount of static semantic checking. Inscape's semantic requirements greatly exceed those of a typical programming language, so we have run afoul of assumptions that were made in the implementation.

5.1 Presentation Issues

The root of our problems with presentation lies in the fact that the Gandalf user interface is modeled after the display version of Emacs, which was the state of the art in the late 1970s. The Gandalf user interface has been modified only slightly since then. Although the basic design of the windowing subsystem and its interface is sound, its implementation is simply outdated. It has a limited number of windows arranged vertically on the screen, and window management is primitive. This was unsatisfactory for our prototype, which derives a large amount of auxiliary information about the program under construction that is inappropriate to display in the main window with the code. This auxiliary information includes such information as the specification-level interface for the current statement and the current block, including the "propagated preconditions" which must be true before the statement or block begins execution, and the "propagated postconditions" which will be true after it finishes.

We also want to be able to display semantic error messages in a separate window, with links maintained back to the program, so that the user can treat the error window as a menu: by ‘‘selecting’’ an error, the user ‘‘selects’’ the point in the program where the error occurred. None of this could be done using the standard Gandalf user interface without running into the basic limitations on the amount of display space available; we would have ended up with a large number of two- and three-line (albeit, scrollable) windows on the screen, none of which could show enough information to be of reasonable use.

Properly adding a windowing capability to Gandalf — that is, integrating it with the existing interface rather than simply grafting it on separately — is too difficult a task to expect an environment implementor to undertake (ironically, because of one of Gandalf's advantages we mentioned previously). The default user interface is in the Gandalf kernel, and (because the Gandalf system is so well integrated) there is no way to replace it without wholesale kernel alterations.

As a partial solution, we added a separate module containing code for creating and modifying Sun windows, with its own interface, separate from the Gandalf windowing interface, and with no direct connection to the standard Gandalf interface. The barrier is the mapping of graphics operations, mouse I/O and keyboard I/O into the Gandalf kernel paradigm. Given these limitations, we could not, for example, read input from a Sun window. We did, however, find a way to automatically update the display in the SunWindow when the underlying Gandalf tree changed by judicious addition of deletion and replacement code in the appropriate daemons to mirror the changes in the tree. We have been able to work around some of these technical problems, but find new difficulties whenever we try to change the user interface, because the basic, underlying problem — an inflexible, non-extensible user interface in the Gandalf kernel — remains unresolved.

5.2 Object Management Issues

The problems with Gandalf's object management divide into two subproblems: "syntax tree bias" and reference semantics. When we mention syntax tree bias, we're not breaking our promise to steer clear of problems specific to syntax editors. This is merely the manifestation, in a structure editor, of a far more common problem in CASE tools: Namely, the editor knows how to deal with only one kind of data — in this case, syntax trees. No other data objects can be edited. This is fine, but Gandalf also makes the assumption that the objects that an environment wants to save to the disk are the same kind of data — syntax trees again. Unfortunately, the Inscape Prototype runs afoul of these assumptions in two ways.

First, we want to save structures other than syntax trees to disk. The prototype's internal symbol tables, for example, are simple linear lists implemented as a syntax tree. They could be much more efficiently implemented as hash tables, if only it were possible to add hash tables as a data type that could be saved to disk. This adversely impacts efficiency and makes it less realistic to expect to be able to build "large" systems in the environment; the symbol table implementation slows down both the query commands and the semantic verification checks to the point where they are prohibitively expensive for anything much larger than a prototype system.

The second assumption that we run up against is the one that all syntax trees are permanent. Sometimes we need a user-editable syntax tree, but we do not need to save it. This is the case, for example, with our query and query result trees. Query trees are constructed by the user to specify which preconditions, postconditions, and obligations he wants to search for in the database. They have the same structure as the subtrees for specifications of functions in the program, and the user edits them similarly, though in their own window. After the query is processed, the query tree is retained, since often a user may want to issue a slight modification of the query based on the initial result. There is never any reason to save these trees to disk, although the current implementation does. It would not be hard to set up a daemon, called on exit from the editor, that deleted the temporary trees so that they would not be saved. However, daemon code is an awkward way to specify something that should be a simple attribute of a tree.

The other, more complex, problem area deals with reference semantics. References were originally added to Gandalf as a means of creating cross-tree links, mainly to allow symbol tables (implemented as attribute trees) and the main program tree to cross-link with one another for the benefit of semantic action (*e.g.*, typechecking) routines. The semantics of Gandalf references, unfortunately, still reflect that origin, and proved to be inadequate for some of the uses we had planned for them. For example, the original plan was to keep only one copy of the syntax tree for each pre- or postcondition specified for a function, and make heavy use of references back to these definitions in the consistency checking code, keeping lists of references as intermediate results describing the specification-level interface for each statement and block in the program. But Gandalf references were, at that time, not displayable (this is no longer the case); there was no unparsing specification that meant "go through this reference and show me what it points to". Since, as we have already mentioned, we needed to display these intermediate interface results to the user, we had no choice but to copy the definition into each of these intermediate lists. This copying overhead did even more to slow consistency checking down than the symbol table problem we mentioned previously.

Copying of subtrees containing references within themselves (as opposed to having references outside themselves) led to other problems. Each predicate and function definition has a local symbol table for arguments, *etc.*, so when copying definitions into intermediate lists, it was necessary to copy the symbol table, as well. Gandalf took care of this to a large degree; we merely had to specify that the insertion (copying) semantics were the same as the creation semantics for the affected statement types. This was, again, inefficient; we would have preferred to have been able to copy a subtree as a template, with all attributes and references in the copy being set up analogously to those in the template. The Gandalf model, however, does not include such an operation; instead, all references must be constructed individually by action routines. We could have written a user-level ARL routine to perform the copying operation, but only at the expense of no longer being able to use the "clip" and "insert" (cut and paste) commands to make copies of subtrees. The template copying routine would have been in direct conflict with the semantic action routines, since creation semantics are called whenever a syntax tree node is created by ARL code.

Our final problem with reference semantics also seems to be tied to the implicit assumption made by Gandalf, not stated in any of its documentation, that references are used only for symbol table and typechecking semantics implementation. When the "tail" of a reference (the node being referred to) is deleted from its tree, Gandalf "suspends" the reference, effectively removing it from the tree unless the deletion is later undone. This is exactly the right action for a symbol table implementation, but another of our attempted uses of references, as described below, wound up tripping over these semantics.

During the processing of a query by Inquire, the user's query is transformed into a conjunction of primitive terms, queries are done on each of these primitive terms, and the results of these primitive term queries are then combined in a later step. This is obviously not the best way to process a query on a large database, but is quite adequate for a small prototype system and was relatively simple to implement. These primitive terms are not visible to or editable by the user; they are kept in an internal part of the tree, and deleted once their processing is finished. Originally, each result returned from a query was tagged with a reference to the primitive term that the result satisfied. This caused bugs when results were found for various terms during query processing, and the terms were deleted before the final combination step; the references to the terms were suspended, and the combination code could not tell which terms had produced which results. We had been thinking more in terms of a reference-counted (or garbage-collected) programming language, where having a reference to an object protects it from actually being deleted, even if the original pointer to the object has been lost. This was a case of a conflict between our intuition of reference semantics and Gandalf's actual semantics, which assumed that references were used only to facilitate symbol table and typechecking semantics.

5.3 Supporting Tool problems

There are two specific problems and one general problem. First, there was no debugger for ARL. The ARL code is translated into C and the debugging must be done with one of the standard debuggers with the programmer mentally decompiling the C code back into ARL during debugging. This is not a major problem, but one that is annoying nonetheless.

Second, there is only a very limited notion of version in Smile: the base version and the experimental version. This is too limiting for the development of any reasonable sized project, especially if it is of an experimental nature. There have been a number of times when we would have liked to back out a base version to a previous base version. Since only two versions are kept, this is not possible in Smile.

Finally, the more general problem is that, because of the tightly integrated and endemic structure of Smile, it is not possible to integrate available tools into the support environment. Thus, we can not make use of existing tools that are not part of Smile, nor add new tools to the environment.

6. Conclusions

On the whole, our experience using Gandalf as a prototyping mechanism has been positive. Using the Gandalf System enabled us to concentrate on the important issues of syntax and semantics without having to build the enclosing editing and presentation layer, and the Gandalf philosophy of incremental construction served us well in several ways. Not surprisingly, most of these positive experiences were expected. These expected benefits were the reasons for choosing to use the Gandalf System as the prototyping vehicle. We also expected some of the negative experiences. For example, we knew that our application would stretch the limits of the original intent of the system. Even so, most of these negative experiences were unexpected, and result from the less obvious, underlying implications of that original intent. Because these negative experiences were less obvious, we required more space to explain them.

The most hindering problems we had involved presentation. We were effectively limited in the facilities we could provide by an outmoded implementation of the user interface. There was no easy way to alter the presentation facilities because they were enmeshed in the Gandalf kernel. We made extensive use of multiple unparsing views, but were hindered by the inability to have multiple, simultaneous, interactive views. By grafting a SunWindows extension onto Gandalf, we were able to provide multiple simultaneous views, but only one of these could be interactive.

Although we have discussed object management problems at length, these were less serious than the presentation problems, since they were for the most part surmountable (usually at the expense of efficiency).

It bears mentioning that without the rapid prototyping capability of Gandalf, we would have taken much longer to reach a point where we could have had similar experiences.

Acknowledgements

The Gandalf Group at Carnegie-Mellon University provided both us with timely support and solutions to many of our problems. Tom Wetmore gave us a careful reading and insightful suggestions.

References

[1] A.N. Habermann and D. Notkin. ‘‘Gandalf: Software Development Environments’’. *IEEE Transactions on Software Engineering*, SE-12:12 (December 1986). pp 1117-1127.

[2] Dewayne E. Perry. ‘‘The Inscape Environment’’. *Proceedings of the 11th International Conference on Software Engineering*, Pittsburgh PA, May 1989. pp 2-12.

[3] M. V. Zelkowitz. ‘‘Evolution Towards Specifications Environment: Experiences with Syntax Editors’’. *Information & Software Technology*, 32:3 (April 1990). pp 191-198.

[4] Gail E. Kaiser, Dewayne E. Perry and William M. Schell. ‘‘Infuse: Fusing Integration Test Management with Change Management’’. *Proceedings of COMPSAC ’89 — The 13th Annual International Computer Software and Applications Conference*, Orlando, FL, September 1989. pp 552-558.

[5] Dewayne E. Perry and Steven S. Popovich. ‘‘Inquire: Predicate-Based Use and Reuse’’. Inscape Technical Report, AT&T Bell Laboratories, September 1990.

[6] Dewayne E. Perry. ‘‘The Logic of Propagation in The Inscape Environment’’. *Proceedings of SIGSOFT ’89: Testing, Analysis and Verification Symposium*, Key West, FL, December 1989. *SIGSOFT Software Engineering Notes* 14:8 (December 1989). pp 114-121.

[7] Barbara J. Staudt, Charles W. Krueger and David Garlan. ‘‘A Structural Approach to the Maintenance of Structure-Oriented Environments’’. *Proceedings of the ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments,* Palo Alto, CA, December 1986. *SIGPLAN Notices,* 22:1 (January 1987). pp 160-170.

[8] Gail E. Kaiser and Peter H. Feiler. ‘‘Intelligent Assistance without Artificial Intelligence’’. *Thirty-Second IEEE Computer Society International Conference*, San Francisco, CA, February 1987. pp 236-241.