Submitted to
International Conference on Information Technology

Error Analysis and Reporting in Programming Environments

W. M. Waite

CU-CS-456-90 January, 1990

ABSTRACT: A programming environment combines elementary operations into single composite operations to be applied by the user. The operands of the composite operation may be decomposed, their components processed by elementary operations, the results combined in arbitrary ways and manipulated by other elementary operations, and a final product delivered to the user. If one of the elementary operations detects an inconsistency in its data, the raw report of that inconsistency will almost certainly be incomprehensible to the user. The programming environment must therefore be capable of using information about its own structure to process the raw report and provide the user with relevant diagnostics. We have developed a general approach to the design of such error recovery mechanisms and demonstrated its usefulness in an experimental programming environment. Our technique can also be used to provide a comprehensive help facility based on a hypertext version of the system documentation.

1. Introduction

There are four general steps required to solve any problem on a computer:

1) Understand the problem.
2) Have an idea about how to solve the problem.
3) Show that the idea does, in fact, solve the problem.
4) Implement the idea.

Often our understanding of the problem consists of recognizing that problem as an instance of some other problem. If the other problem has already been solved, this recognition greatly simplifies steps 2-4 above. For example, the construction of a telephone book can be recognized as an instance of a sorting problem. Step 2 consists of choosing a known sorting technique appropriate to the data, step 3 need only show that the data is compatible with the requirements of that technique, and step 4 is carried out by extracting the sort routine from a library.

A so-called ''fourth-generation language'' provides a mechanism for a person to describe an instance of a particular problem. The processor for the fourth-generation language, which embodies steps 2-4, then generates a program that solves the problem instance described. Sort generators and report generators are common examples of fourth-generation language processors. Parser generators like yacc,[1] which accept a context-free grammar describing a particular instance of the parsing problem and generate a program to carry out the parse, are also examples of fourth-generation language processors although they are not often labelled as such.

Most problems are too complex to be solved monolithically. Instead, they are decomposed into simpler problems. These simpler problems may be decomposed in turn until some set of elementary components is reached. For example, a compiler for a new programming language/target machine pair can be recognized as an instance of a compiler problem. We know that compiler problems can be decomposed in a certain way into simpler problems like scanning, parsing, name analysis, code generation, and so forth. Moreover, we can formally describe instances of many of the elementary component problems.

The technique of using fourth generation languages to describe problem instances can be extended to complex problems by building programming environments that embody standard decompositions of those problems. The standard decomposition embodied in the environment identifies a set of component problems, such that an instance of the complex problem is equivalent to a set of instances of the component problems. A user thus describes the complex problem instance by describing the instances of its components. Construction of the program that solves the complex problem is managed by the programming environment; the user need not be aware of how the component problems are solved or how their solutions are combined.

Generally speaking, the component problems of a complex problem are not specific to that particular application. For example, one of the component problems of a compilation problem is *scanning* — the process of recognizing sequences of characters as instances of source language basic symbols. Scanning is also a subproblem of the bibliographic search problem,[2] the problem of correcting spelling errors in text,[3] and many similar problems. This means that fourth-generation languages and generators for solving these component problems are often available ''off-the-shelf''. When building a programming environment to solve a complex problem, a designer should take advantage of these existing processors to shorten the development time and improve the reliability of that environment.

When a programming environment is constructed partially from existing components that were not originally designed to work together, the construction techniques must allow sufficient flexibility to accommodate the disparate interface requirements of those components. It should also be easy to replace a component when a better processor for that component's subproblem becomes available. These requirements for flexibility suggest that the programming environment itself be implemented as an expert system: an inference engine capable of activating components, and a separate knowledge base describing how those components interact to solve the overall problem.[4] We have used this approach very successfully in a programming environment for compiler construction.[5]

2. Error Reporting in a Programming Environment

Consider the model of a programming environment proposed in the introduction. It consists of a collection of off-the-shelf components that were not originally designed to work together. The operation of these components is controlled by an expert system that understands how to construct a solution to the complex problem from solutions to the component problems. A user supplies a set of specifications, and requests a solution to the complex problem. The user's specifications are examined, possibly combined or further decomposed, possibly transformed in some way, and submitted to the environment's components. Information resulting from these operations may be processed further by other components. Ultimately a processor that solves the instance of the complex problem described by the original specifications is constructed and delivered to the user.

In the best of all possible worlds, the user of the programming environment need understand only how an instance of the complex problem is described by describing instances of the subproblems. There is no need to understand how the specifications are processed, what the subproblem solutions look like, or how they are combined into the solution of the complex problem. The actual construction of the solution becomes a single atomic operation. The details of solving the complex problem are hidden in the programming environment just as the details of computing a square root are hidden in the Pascal operation *sqrt*.

Consider what happens, however, when the user makes an error in describing one of the subproblems. If the error changes the subproblem description into a valid description of a different instance of this subproblem than the user intended, the programming environment will successfully create a processor for an instance of the complex problem different from the one the user intended. No error will be detected in the process, because there is no way that any program can divine the intent of its user. It is only if the error makes the subproblem description either invalid or inconsistent with the description of some other subproblem that the programming environment is able to issue a report to the user.

Most errors will be detected by elementary components of the programming environment. Remember that most of these components were developed independently of the programming environment, and thus their error reports will be stated in terms of their own inputs and world view. The reporting component's inputs are generally *not* the original specifications provided by the user, but intermediate objects that were derived from those specifications; its world view is restricted to the particular subproblem it was designed to solve, which may be one not even known to the user. (Descriptions of many small subproblem instances can be derived descriptions of larger ones.) Therefore most of the raw error reports output by components of programming environments would be incomprehensible to the user.

In order to be useful, error reports produced by individual components of a programming environment must be referred to input that the user provided and concepts with which the user is familiar. The individual components cannot be responsible for producing reports that are useful in this sense, because they were designed in a different context. It is not generally possible for the designer of the programming environment to modify the individual tools because they may be available only as object modules. Even when source code is available, the cost of retrofitting useful error reports is prohibitive. Not only must all of the components be understood and modified, but this task must be redone whenever the organization of the programming environment is changed in certain ways. Maintenance therefore becomes a nightmare.

The key insight into the solution of the error reporting problem for programming environments is the principle of ''unhurried diagnostics'', stated by W. S. Brown.[6] A program having information about a failure may take any one of four actions:

1) Output the information and terminate.
2) Output the information and proceed by an alternate route.
3) Suppress the information and proceed by an alternate route.
4) If not the main program, pass the information to its caller and indicate failure.

Each of the first three actions assumes that the information is relevant on the level at which the program is operating; the fourth allows the caller to alter the information in response to environmental considerations unknown to the program having the information.

In the context of a programming environment that fits the model presented in the introduction, the ''caller'' of a program is the expert system that manages user requests. The expert system's knowledge base contains all of the information available about the relationship between the program reporting the error and the user's input specifications. Our only problem is thus to use that information to change the conceptual level of the error report. The change might require arbitrary processing, and possibly additional information in the knowledge base. It has the same characteristics as the derivation of a program to solve the complex problem, however, and can therefore be managed by the expert system in the same way.

3. A Specific Solution of the Error Reporting Problem

My group at the University of Colorado has developed a programming environment to solve the problem of constructing compilers. Called Eli,[7] it consists of a collection of compiler construction tools managed by an expert system. The expert system shell that we used is Odin.[4] Its knowledge base is a directed graph, called a *derivation graph*, whose nodes represent data objects. Each node is associated with an atomic operation that creates the node's data object from the data objects represented by the node's predecessors. The expert system maintains a cache of these data objects. When a user requests production of an object represented by a node, the inference engine determines whether the requested object is in the cache. If so, it is simply delivered; otherwise the inference engine uses the derivation graph to select a sequence of actions that will construct all precursors of that object not already in the cache and then the object itself.

An atomic action is the invocation, by Odin's inference engine, of a Unix shell script. The script itself selects the program that will interpret it and decides which directory to run in. Before running the script, the inference engine modifies it by replacing certain distinguished strings with the file names of the node's predecessors in the derivation graph, the name of a newly-created directory that will be deleted after the script completes, and the names of several files to be used for results. Because the script can literally do anything, it can satisfy the interface specification of any program it invokes. This flexibility is what allows us to use arbitrary off-the-shelf software to implement elementary actions.

Suppose that the program implementing a particular atomic action creates an error report. The designer of the programming environment must now make one of the four choices listed at the end of the last section. This choice can be made on the basis of any arbitrary analysis of the error report, carried out by a program or programs invoked by the script controlling the atomic action. For example, the Berkeley Pascal compiler issues reports whose first character is ''E'' to indicate unrecovered errors. If an error was recovered reliably then the report begins with ''e'', whereas reports that are simply warnings begin with ''w''. A designer could treat these messages differently by using one of a number of Unix facilities to build a filter that separated them into different files.

In order to output the information and terminate, the designer would make the script send that information to a file whose name is represented by the distinguished string `(|>ERROR|)`. The inference engine will replace this string by the name of a file in the cache. If this file is not empty on completion of the script, the inference engine does not carry out actions corresponding to successor nodes. Other actions required for the ultimate product will be executed. Those other actions might also result in files containing information about failures. When the inference engine completes all of the requested actions that can be completed, it concatenates all of the files containing information about failures and *derives* printed output from the result.

The important point is that the derivation graph can specify arbitrary manufacturing steps to be applied to failure information. Those manufacturing steps can access arbitrary information about both the derivation graph and the current contents of the cache. Thus it is possible to apply a complex interpretation process to reports provided by any component program, without altering that program in any way. A user of the system is unaware of this interpretation process, and need know nothing about the program issuing the report.

Consider a script that executes a program capable of producing failure information. That failure information relates to some input of the atomic action controlled by the script. Any analysis to determine this relationship must be done locally, either by the script itself or by other programs that it invokes. Further analysis depends upon relationships described by the knowledge base and/or data stored in the cache, and is therefore outside the competence of the script. The script must therefore pass the (possibly modified) failure information to the expert system for further analysis. It must specify two things in addition to the failure information: the input object to which the failure information relates, and the kind of failure information being provided.

When the expert system receives failure information from an atomic action, it applies a derivation described by the knowledge base to that information. The derivation to be applied is determined by the two pieces of additional information supplied by the atomic action. Each kind of failure information corresponds to a distinct object to be derived. If a derivation is possible from the input object specified by the atomic action to the object corresponding to the kind of failure, then that is the derivation applied to the failure information. Otherwise, the expert system attempts to find a derivation from some predecessor of the input object specified by the atomic action. The search is breadth-first, so that a derivation from the ''closest'' predecessor is used.

A script may, of course, need to specify different analyses for different parts of the failure information. The script indicates that the expert system should apply a derivation to a particular part of the failure information by bracketing that part:

(| *InputObject* : *FailureKind* : *Information* |)

Here *InputObject* is the name of the input object of the atomic action to which the failure information relates, and *FailureKind* is the name of the object to which the expert system should derive the failure information. There can be an arbitrary number of bracketed groups in the file created by a script and written to `(|>ERROR|)`, each of which might specify distinct input objects and failure kinds. Also, *Information* can be any sequence of characters, possibly including newlines, without restrictions on length. Bracketed constructs can be nested to cascade error analyses; any failure information that is not contained within bracketed constructs is presented to the user unchanged.

4. Provision of Additional Documentation

We have increased the amount of information available for error reporting by placing the system documentation on line in hypertext form. Not only does this allow the user to browse the documentation in order to answer questions arising while creating specifications, but it permits the error analysis to access the documentation needed to explain a failure. There is no need for the user to have the printed form of the documentation at hand because a simple request will place them in hypertext browsing mode at the most likely explanation of their problem. If the error analysis is wrong about the explanation, the user is in a position to browse the entire set of documentation if necessary.

This facility is implemented by a simple combination of the error analysis discussed in the previous section and a general hypertext system called *Texinfo*, developed by the Free Software Foundation. Texinfo generates printed documentation and hypertext nodes from a single body of text designed for the page formatter Tex.[8] The hypertext nodes are grouped into a number of text files that contain special character sequences to provide linkage. A browsing facility for the hypertext is included in the text editor Emacs, and is also available separately. (All of the Emacs documentation is provided in Texinfo form.) We used Texinfo to implement the documentation of the Eli system, independent of any consideration of incorporating it into a specific help system for error reporting. It was only after we had the error reporting facility running that we realized the two could be used in combination.

A normal user request to Eli might look something like the following:

```
name.specs : exe > name.exe
```

This asks that an executable version of a compiler be derived from the problem description `name.specs` and stored in the user's file `name.exe`. If inconsistencies in the description are detected then the system will simply respond ``System abort status set for name.specs : exe''. In order to obtain the analysis of the errors, the user must explicitly request it:

```
name.specs : exe : err
```

(Since all of the relevant objects were already placed in the cache by the previous request, this request need only carry out the error analysis.) The expert system is being asked to *derive* the error analysis here. This derivation, like any derivation, can call upon all of the information encoded in the knowledge base and stored in the cache. All of the power of the technique is derived from the use of a derivation for error analysis.

The standard error analysis yields error messages that are linked to character positions in input files. All that is needed to support a general help facility is a means of linking these error messages to the documentation. It happens that the error messages output by the underlying tools used by Eli were numbered by the tool designers. Therefore it is easy to establish a correspondence between them and hypertext nodes simply by defining an array, indexed by error number, for each tool. The elements of the array are the hypertext file and node names, as character strings. A simple filter program extracts the number form the error report, looks it up in the proper array, and creates a hypertext menu item from the resulting string. These menu items are collected into a single hypertext node that can then be linked dynamically to the hypertext network containing the complete system documentation. All of this processing is described by a derivation, triggered by a user request of the form:

```
name.specs : exe : help
```

The key point is again that the expert system is able to perform arbitrary derivations from the text of the reports provided by individual elementary actions.

We have modified the standard Texinfo browser to allow specification of a list of ``editable files'' in any hypertext node. When browsing a node that specifies editable files, the user can begin an editing session on any one of those files without leaving the hypertext browser. On a workstation with a window-based display, the editing session and the browsing session can proceed independently in two windows. Once the editing session has begun, the user may switch the browser's attention to another window without affecting the editing session. Our help derivation actually creates a hypertext network with one node for each specification file that contains errors. The node contains the lines of the file in error, with embedded error messages, and a menu with one entry for each distinct message. That menu entry accesses the documentation node relevant to the error message. Thus the user can start an editing session on the file, then browse the documentation to explain the error. Both the documentation and the relevant part of the file are visible simultaneously, as are the corrections when they are made.

5. Conclusion

We have combined expert system technology and the hypertext concept to create a powerful approach to problem solving. This approach allows us to create programming environments that generate processors for solving complex problems. The programming environments embody specific decompositions of the complex problem, and a user describes the complex problem instance by describing instances of the component subproblems. Processors for these subproblems are generated by off-the-shelf generators, and those processors are combined by the programming environment to solve the complex problem.

Error reporting is difficult in such environments because programs that carry out elementary actions are not designed specifically for the environment in which they are embedded. We handle the analysis of failure reports by treating it as any other derivation. This allows the designer to introduce any arbitrary processing, and to use all of the resources of the expert system to manage that processing. By using hypertext for the programming environment's documentation, the designer can also provide extensive help facilities that derive a starting point and simplify user access to the entire documentation corpus for error diagnosis.

We have built a compiler construction environment according to these principles. It has been in use at the University of Colorado for three years, and was released to the general public in November, 1989. Our experience with this environment has shown that the approach to error reporting described in this paper can solve the unique problems of such systems.

6. References

1. S. C. Johnson, 'Yacc — Yet Another Compiler-Compiler', Computer Science Technical Report 32, Bell Telephone Laboratories, Murray Hill, NJ, July 1975.
2. A. V. Aho and M. J. Corasick, 'Efficient String Matching: An Aid to Bibliographic Search', *Communications of the ACM*, **18**, 333-340 (June 1975).
3. J. Bentley, 'A Spelling Checker', *Communications of the ACM*, **28**, 456-462 (May 1985).
4. G. M. Clemm, 'The Odin System - An Object Manager for Software Environments', Ph.D. Thesis, Department of Computer Science, University of Colorado, Boulder, CO, 1986.

5. W. M. Waite, V. P. Heuring and U. Kastens, 'Configuration Control in Compiler Construction', in *Proceedings of the International Workshop on Software Version and Configuration Control*, Teubner, Stuttgart, FRG, 1988.

6. W. S. Brown, 'An Operating Environment for Dynamic-Recursive Computer Programming Systems', *Communications of the ACM*, **8**, 371-377 (June 1965).

7. R. W. Gray, V. P. Heuring, S. P. Krane, A. M. Sloane and W. M. Waite, 'Eli: A Complete, Flexible Compiler Construction System', SEG 89-1-1, Department of Electrical and Computer Engineering, University of Colorado, Boulder, CO, June 1989.

8. D. E. Knuth, *The Texbook*, Addison Wesley, Reading, MA, 1984.