DOE Form IR-427
(1/79)

U.S. DEPARTMENT OF ENERGY

UNIVERSITY-TYPE CONTRACTOR AND GRANTEE RECOMMENDATIONS FOR DISPOSITION OF SCIENTIFIC AND TECHNICAL DOCUMENT

See Instructions on Reverse Side

1. DOE Report No.

2. Contract No.
DE-FG02-84ER13283

3. Title
CRITICS: An Active Approach to Tools and Environments

4. Type of Document ("X" one)
- [X] a. Scientific and technical report
- [ ] b. Conference paper:
  - Title of conference _____
  - Date of conference _____
  - Exact location of conference _____
  - Sponsoring organization _____
- [ ] c. Other (Specify Thesis, Translations, etc.)

5. Recommended Announcement and Distribution ("X" one)
- [X] a. DOE's normal announcement and distribution procedures may be followed.
- [ ] b. Make available only within DOE and to DOE contractors and other U.S. Government agencies and their contractors.

6. Reason for Recommended Restrictions

7. Patent Information
Does this information product disclose any new equipment, process or material? [ ] Yes [ ] No

Has an invention disclosure been submitted to DOE covering any aspect of this information product? If so, identify the DOE (or other) disclosure number and to whom the disclosure was submitted. [ ] Yes [ ] No

Are there any patent related objections to the release of this information product? If so, state these objections.

8. Submitted by
Leon J. Osterweil, Professor and Deborah Baker, Assistant Professor
Name and Position (Please print or type)

University of Colorado, Department of Computer Science
Organization

Signature _____ Date 4/20/85

FOR DOE USE ONLY

9. Patent Clearance ("x" one)
- [ ] a. DOE patent clearance has been granted by responsible DOE patent group.
- [ ] b. Report has been sent to responsible DOE patent group for clearance.
- [ ] c. Patent clearance not required.

CRITICS AN ACTIVE APPROACH TO TOOLS AND ENVIRONMENTS

by

Deborah A. Baker and Leon J. Osterweil

CU-CS-285-84 December, 1984

University of Colorado, Department of Computer Science, Boulder, Colorado.

ANY OPINIONS, FINDINGS, AND CONCLUSIONS OR RECOMMENDATIONS EXPRESSED IN THIS PUBLICATION ARE THOSE OF THE AUTHOR AND DO NOT NECESSARILY REFLECT THE VIEWS OF THE NATIONAL SCIENCE FOUNDATION.

Critics: An Active Approach to Tools and Environments

Deborah A. Baker and Leon J. Osterweil

Department of Computer Science
Campus Box 430
University of Colorado at Boulder
Boulder, Colorado 80309

Abstract

This paper describes the notion of *software critic tools*. A critic analyzes an object (such as a program) for compliance with one or more design rules or guidelines, but, unlike more traditional software tools, a critic is active, rather than passive, in nature. Critics approach the problem of controlling the complexity inherent in, and the voluminous information produced during, software development by furnishing powerful analytic feedback to the user automatically, easily and efficiently. This paper defines the notion of a critic, and also discusses issues involved with the implementation and definition of critics. The implementation issues also raise certain architectural considerations which seem to be of considerable importance to tool and environment architects.

Keywords

Software engineering, tools, static analysis, software environments.

This work is partially support by National Science Foundation grant # DCR-8403341 and Department of Energy grant # DE-FG02-84ER13283.

1. Introduction

There has been continuing strong interest in creating tools and toolsets to support the software development process and in particular to aid in coping with the complexity inherent in producing large software. Initially software tool development efforts focussed on the creation of increasingly large, sophisticated and powerful tools. As such tools were made available in increasing quantity and diversity, it rapidly became apparent that some guidelines for their use and coordination were essential to the effective exploitation of the tools. Considerable previous work in the area of software development methodologies suggested that one way to effectively coordinate tool use and application would be to construct a set of tools capable of sharply and effectively supporting a single specific software methodology. The continuing absence of agreement and consensus about which development methodologies are superior continues to discourage and frustrate those who would focus their tool development efforts on narrow support of a particular methodology. Instead tool developers have concentrated on building integrated collections of tools coordinated and integrated according to a variety of innovative strategies and aimed at providing power and flexibility. These innovative integrated toolsets are generally referred to as software development environments. They represent an excellent vehicle for experimentation with tools, tool combinations, and development methodologies, while also serving as experimental objects themselves.

Our research has recently focussed on creating and experimenting with a tool integration strategy, called Odin [Clemm 84], which emphasizes flexible, modular, extensible tools, and the importance of a central repository of objects which are the products of these tools and their constituent parts. This integration strategy has been used to create a specific software development environment, called Toolpack/IST [Osterweil 83]. This early work has lead us to believe that it is useful to center environments around a collection of data objects and to construct toolsets out of small modular pieces. We have also come to realize that these sorts of environment architectures make it all too easy for the user to create large collections of large data objects. This large mass of data then unfortunately becomes an obstacle to clear insight and understanding of the software rather than an effective vehicle to this insight and understanding.

Thus we have become increasingly interested in the creation of aids to the process of inferring and distilling more concise knowledge from larger bodies of software information. These aids are properly thought of as modular tools themselves, but they differ from the other sorts of tools in our environment in two ways. First, their primary job is not the creation of new information, but rather the summarization and distillation of existing information. Thus they resemble a class of tools which we have earlier referred to as *viewers*. More importantly, however, these tools are significantly different in that they seem to us to be more effective when invoked automatically rather than explicitly by the user. For this latter reason, we refer to these automatically invoked tools as *active* tools, to contrast them to our earlier tools which operate passively, only when explicitly invoked. Because these tools have an analytic flavor as well, we believe it is appropriate and suggestive to refer to them as critics.

We believe that, as tools and tool configurations become increasingly complex and powerful, they will increasingly have to become more active and self-critical of their own actions and products. Thus we are increasingly persuaded that it is time to study the potential for converting existing passive tools into active critics and for synthesizing combinations of analytic capabilities into critic environments. This paper describes

the notions of active tools and critics. It also discusses some issues raised by the need to effectively and efficiently implement critics. This in turn leads to consideration of architectural issues raised by the need to create environments increasingly centered upon effective support for critics as a focus of the environment's user interface.

2. Critics

A *software critic* is an *active* analytic software development tool. A critic applies one or more design rules or rules of thumb to programs, designs or specifications as they are being developed or after they have been completed. We say that a critic *animates,* or *embodies,* its rules. In the following paragraphs the active nature of critics and the rules critics embody will be discussed in more detail.

Traditional software tools, whether they exist individually (such as Lint [Johnson]), as part of a tool suite (such as Toolpack [Osterweil 82, Osterweil 83]), or as part of a development methodology (such as the tools associated with SREM [Bell 77] or PSL/PSA [Teichroew 77]), are intended to be explicitly invoked; furthermore, traditional software tools are intended to operate on the particular object or set of objects named, implicitly or explicitly, upon invocation. An active software tool is distinguished from traditional passive tools in two respects. First, active tools are itinerant, automatically identifying and processing all objects in an information repository which are in their purview. They do not have to be given instructions specifying which objects they are to process. Second, active tools operate without being explicitly invoked. These characteristics define what is meant by the term *active*: an active tool does not need to be explicitly invoked nor does it have to be told which objects are in its purview.

Critics are active tools whose processing is aimed at analyzing, abstracting, summarizing, or otherwise digesting and interpreting data which has presumably been generated by the actions of other tools.

The goal of providing critics is to help users effectively absorb and cope with the masses of information which software tools are becoming increasingly able to make available. Critics seem able to do this by making analysis of programs, specifications, designs, and other objects practical and efficient in routine use. Critics address this goal in several ways. First, because it is not necessary to explicitly invoke a critic, the use of such a capability should be quite convenient. In addition, because critics are not invoked explicitly, their processing need not be completed and results returned to the user immediately. Instead it is expected that critic tools will, once started, execute in the background while the user addresses his or her attention to the processing of passive tools. Thus the critics would execute the fastest while the user was using passive tools the least -- presumably during periods of contemplation. As a result, critics would deliver their analytic and summary results to the user some time after significant changes had been made, but presumably not during periods when the user is making rapid changes using passive tools.

Critics can embody rules ranging from simple to complex. A trivial example of a critic tool is one that would warn of a procedure whose source text became too long to list on one or two pages. Such a design critic embodies the folk rule of thumb concerning length of procedures. In this case the critic would be a trivial piece of code whose execution could readily be triggered any time a procedure was edited. Critical feedback to the user would be almost immediate.

A more interesting example of a critic would be a tool for enforcing the design rule for a distributed Ada[1] program mandating that a task should not call one of its own entries. (While not prohibited by the rules of the language, any task that calls one of its own entries will immediately deadlock [ALRM].) A critic embodying this rule would be harder to construct, requiring lexical, syntactic and some semantic analysis of Ada tasks. Thus the user of such a critic should not expect that the critic would give instantaneous diagnostic feedback when the error was committed. Instead the critic would, at some point after the code for the task had been completed, initiate its analysis, reporting the observed violation of the design rule, presumably after the user had begun to do other work with passive tools. It is important to note that, because a critic is an active tool, the user does not control when the critic initiates or completes its analysis. As a result the user is not required to wait until the critic has completed its analysis before proceeding. Conversely, the user is not assured that once he or she proceeds all critical analyses have concluded and that it is therefore safe to assume that he or she has not committed any violations of critic-enforced rules. This suggests the desirability of giving the user some amount of power to specify at least the policy or strategy under which critics are to be applied. Presumably critics should execute as rapidly and automatically as possible, but without being intrusive, obstructionist or meddlesome. For example, a critic should not annoy the user with diagnostic feedback while the user is in the middle of carrying out a complex, multistage editing procedure. Consideration of ways in which the user might be given this power seems to imply important toolset architectural considerations which will be discussed later in this paper.

The power and value of critics can perhaps first start to be seen in considering the conversion of classical data flow analyzers into critics. Data flow analyzers are tools which statically examine the structure of a program or design specification to determine whether or not it is possible for a given sequence of events to occur when execution proceeds along any path through the code or design (eg. see [Fosdick 76]). One example of such a data flow analyzer is the Dave system [Osterweil 76] which is capable of examining a Fortran program to determine whether or not it is possible for the program to reference an uninitialized variable (a violation of language semantics) or to ignore the value of a previously defined variable (a "dead variable" -- not an error, but a cause of concern). Experience has shown that the existence of these data flow anomalies in a program is often an indirect indicator of serious structural problems in a program. Thus this type of analysis is considered to be quite useful and desirable and very much the sort of feedback which we believe a critic should deliver. Unfortunately, this sort of analysis entails a great deal of computation because in order to be correct it must be done thoroughly, often entailing the analysis of long chains of invoked procedures. Here too it seems that this sort of capability is best thought of as a critic, as the tool which carries out this analysis must be itinerant, deciding for itself which bodies of code and data objects must be found and analyzed. Further, past experience has shown that it is often impractical to use this capability in routine code development and testing contexts because the analysis can take unexpectedly long amounts of time to complete.

A critic whose job was to enforce a data flow rule would carry out its analysis in the background while the user might be working with passive tools. The user would not be required to wait until the critic had completed its work before proceeding. This is important because the critic could be programmed to optimize its analysis in a variety of ways, thereby making it difficult to predict how long its analysis might take.

[1] Ada is a registered trademark of the US Government, ADA Joint Program Office.

Rather than having to risk waiting an unpredictable length of time for the critic to finish, the user would simply go on whenever he or she were so inclined. The critic would have its own agenda in the sense that it would know, for instance if it embodied data flow rules, which invoked procedures needed to be analyzed in order to carry out needed interprocedural analyses. Thus it would not need to be directed at specific bodies of code. It could be made sufficiently clever to recognize when previous analyses had yielded reusable data and when such data needed recomputation. In particular it could be made sufficiently clever to recognize when the user had gone on and made changes which had invalidated partial analytic results. As a result, the critic might be forced to thrash a bit if the user were actively making large and pervasive changes. In the absence of such a situation, however, the critic would be free to commandeer processing time and resources as made available by user inaction, and attempt to remain reasonably close to the user's work with its analyses. The most appealing aspect of this scenario is that the user would not have to decide when he or she was ready to commit to the potentially lengthy data flow analysis process and when to run the risk of not carrying out analysis in order to save time. Instead the user would simply assume that the analysis was constantly being carried out. Current analytic results could be obtained simply by suspending all changes until the analyzer caught up to the user.

This last example suggests that critics could be made even more powerful. Rather than considering that a critic must be restricted to evaluating an object or objects produced by only a single tool it should be clear that a critic could just as easily carry out analysis of sets of objects produced by an ensemble of tools. In fact it seems reasonable to suggest that the job of a critic might best be described as the construction and analysis of a large knowledge structure requiring the execution of diverse tool capabilities. From this perspective we see the possibility of subsuming many currently passive tools as components of active tools and slowly changing the character of some current passive environments into increasingly knowledge-driven active environments.

The examples given to this point were critics of code. The notion of critic can be extended to any objects for which there are design rules or guidelines. For instance, design critics could exist for the specification of an abstract data type (say as Anna [Luckham 84] annotations to an Ada package being designed). Such design critics could embody certain completeness criteria for abstract data types [Cline 83]. *Discriminatory completeness*, for instance, ensures that a type includes operations for differentiating between objects of the type. For distributed programs, further completeness criteria could, for example, incorporate the guideline that updates to a shared variable require exclusive access to that variable [Taylor 83, Taylor 80].

The notion of a critic could also be exploited by tools other than those for the development of software. In an office environment, critics could embody various guidelines for ordinary text. Each time a paragraph were finished, the words could be checked for correct spelling and the sentences for readability.

The results of a critic (like the results of a static analysis tool such as Dave) serve as a warning that a possible design flaw is present in the code, design, specification or other object examined. The software engineer (designer, specifier, or programmer) could choose to heed or to ignore this advice based on his or her insight into the problem at hand.

As stated earlier, we believe that the notion of a critic might well be an important organizational hub of the user's interface to a software development environment.

The architectural ramifications of deciding to give it such prominence are worth considering. Before proceeding to this consideration it seems important to relate our notions to other tool and environment research efforts.

2.1. Related Work

There are existing environments that incorporate some active components. Our work differs from these previous uses of active components primarily in the extent to which active components are used and the roles they are intended to fill. The *programmer's assistant* of Interlisp [Teitelman 81] and the *programmer's apprentice* [Waters 82] are active agents. The programmer's assistant, together with the *DWIM* (Do What I Mean) facility of Interlisp, correct spelling errors a programmer might make and allow a programmer to undo or redo operations. The programmer's apprentice keeps track of details and thereby assists the programmer in building, documenting and modifying a program.

The DOMAIN Software Engineering Environment (DSEE) is a distributed workstation environment [Leblang 84]. Whenever a new version of some piece of information is created, the *history manager* on the network node where the update occurred sends a message to other nodes. This is an active method of providing source code control.

In the PECAN system, multiple views of the same object are possible [Reiss 84]. For instance, a programmer might have accessible views of the syntax, the symbol table, and a Nassi-Schneiderman chart. If, for instance, the syntax view is changed with the syntax directed editor, the other views are automatically updated to conform to the change. Thus, each of the current views are kept up-to-date without explicit intervention from the user.

Syntax directed editing is, of course, a form of active analysis [Teitelbaum 81] where the analysis that is performed is syntactic. Language based environments, such as the Cornell Program Synthesizer and Magpie, often proved incremental compilation, which is also a form of active analysis [Reps 83, Delisle 84].

In the Odin system, *sentinels* can be set into action whenever an object of a certain type is changed [Clemm 84]. A sentinel may correspond to an arbitrary semantic constraint; a program is provided for each sentinel and embodies its constraint. Sentinels are similar to *policies* [Cooprider 78].

Certain interactive systems have been built with active components. One such system with an active help subsystem is the BISY text editor [Fischer 84]. This system monitors a user and gives advice when the user accomplishes tasks in a suboptimal manner. The help system consists of plan specialists, each of which corresponds to a complex activity that can be done more or less optimally. Care is given to give help unobtrusively (i.e. not too frequently and only for tasks that the user has repeatedly done in a suboptimal way).

2.2. Example

We give an example in this section of how a collection of critic tools can assist in the development of a program. Issues dealing with the specification and implementation of critics will be addressed in following sections. We will assume here that critics can

actually be specified and implemented. Critics can be developed for any guideline, and in particular, for programs written in any programming language.

We will assume for the following scenario that a large software system is being developed in a critics environment. An exhaustive list of the critics that are active for this scenario is not important. It suffices that the system is developed under the criticism of the data flow and other sorts of critics that have been mentioned as examples elsewhere in this paper.

A programmer wishes to make a change to an interface in a large software system. A great deal of care must be taken because of the possible consequences of the change. There is very little help that is currently offered a programmer in this situation.

A program can be inspected manually for the possible consequences of a proposed change. But such manual inspection is time-consuming, tedious and error-prone. A tool that locates occurrences of objects in a program (such as a cross reference tool or even a text editor) provides minimal support for locating sites of potential problems. But such a locating tool provides no assistance in performing analysis to determine the presence or extent of problems due to the proposed change. Furthermore, such a tool must be used in such a way as to locate all references to the object under its own name as well as any aliases it might have (as might occur via a *renaming declaration* or *generic instantiation* in Ada [ALRM]).

Existing analysis tools can provide help for the actual analysis task. There are, unfortunately and as noted above, several problems with existing analysis tools that render them less than ideal. The Dave data flow analysis tool [Fosdick 76, Osterweil 76], for example, is a large, monolithic tool. While its data flow results are useful in discovering problems with interfaces, there is no way to direct the attention of the Dave tool. It always produces an entire analysis of a program. It doesn't use results from its previous runs to reduce the amount of analysis it must do. Since data flow analysis is complicated, the programmer might wait for an unreasonably long period of time for the analysis, and then have to distill the desired information from Dave's voluminous output.

Contrast the above scenario with the critics scenario. In a critics environment, the programmer would make the change to the program and go on to do useful work. The critics would analyze the program and prepare critical reports. The programmer could evaluate the critical reports at leisure and decide whether the change was acceptable. Since critics do incremental analysis, the actual amount of analysis will be reduced. If there were no previous data flow anomalies in the program, the critic could restrict its attention to the modified portions of the program.

3. Architectural and Implementation Considerations

From the foregoing discussion it should be clear that a critic cannot be implemented as an independent tool, but must instead be considered to be a component of a larger collection of tools or tool functions. In the simplest case, a critic might be considered to be a semiautonomous part of a large and complicated tool or tool system. For instance, the garbage collection function of a large system is an active tool capability (although not a critic). Thus a critical capability might be imbedded in a larger software tool or system in a fashion analogous to the way in which a garbage collector is imbedded in a string processing system. Alternatively the critical capability might

be implemented as a tool or tool fragment which is a component of an organized tool system or environment.

In either case, to be of practical use, critics should be as unobtrusive as possible and as efficient as possible. If a critic is not perceived by the user to be efficient, it will be considered a drain on resources, and is likely to be turned off, perhaps permanently. However, the animation of some rules involves elaborate, and therefore time-consuming, analyses. Thus, producing critics might appear to contradict the goal of doing so efficiently. Many of these rules, however, will require the same preparatory analyses of the program (or specification, etc.) before the primary analysis can begin. For instance, most analyses of programs will need the results of lexical analysis, syntactic analysis and some semantic analysis. If each critic were implemented as an individual, monolithic tool, these pre-analyses would be performed by each critic. This would lead to gross inefficiency of the set of critics as a whole, as each critic recomputes results that exist in the system but that are not accessible to it. If we can perform such preliminaries once, and allow the various critics to share the intermediate results, we should gain quite a bit of efficiency over individual, monolithic analyses. This strongly suggests that critics are best implemented as cooperating capabilities, preferably even coordinated with the processing of the passive tools whose products they analyze.

Although this could be effected by imbedding critical capabilities within a single large tool such as an intelligent editor, it seems to us preferable to implement critics within the context of a less tightly integrated family of tools such as a loosely coupled environment. The Odin environment integration architecture offers one example of an architecture which seems to facilitate the efficient integration of critics as outlined above. Odin strongly encourages the user of an Odin-integrated tool environment to think of the environment as a repository of objects created by tools and tool fragments. Thus the Odin command language is best thought of as a language for requesting the creation of the objects which tools can build. In Odin, there is a specification language in which each type of object is described as being derived by the application of a sequence of tool fragments. A pretty-printed program, for example, is produced from the source program and the results of lexical and syntactic analysis. The Odin command interpreter has as its job assuring that the objects which a user requests are created as efficiently as possible through the effective reuse of previously created and retained intermediate objects. Using the Odin framework, then, complex critics could be specified as tools which build upon any number of objects produced as the results of the actions of other tools and tool fragments. In an ideal situation a critic tool might merely draw upon the results produced by other active or passive tools.

While this approach clearly offers the advantages of convenience, flexibility, extensibility and efficiency, it still is not necessarily an efficient enough approach to assure that the critic will be able to keep up with changes being made to the information in the environment in which it works. This will be the case especially if the amount of information or rate of change were large or if the critic embodied a complex rule. In this case, the critic would labor in the background some distance behind the changes being made by the users, until the users slowed their pace of changes, focused their attention on an unrelated object, or perhaps suspended their work (e.g. at the end of a working day). At that time, the critic or critics would be able to catch up and perform their analyses on the more static data repository. Their critical reports would be lodged in the data repository and brought to the attention of the relevant users.

A critic that embodies the rule of thumb concerning length of subprograms would

more or less continuously monitor the lengths of the source text of subprograms as they were developed and updated. This is a straightforward task; the critic could give essentially instantaneous feedback to programmers were this guideline to be violated.

A critic that embodies the guideline that an Ada task should not call one of its own entries is not as straightforward as the previous example. It would require scanning, parsing and some semantic analysis, and for that reason would probably lag some distance behind the code writer in its diagnostic feedback. As noted above, these preliminary analyses might be available to the critic as the results of earlier processing by other tools. If, for example, the Ada code were created by an intelligent editor the critic could begin its analysis by examination of a parse tree, or even a decorated parse tree, which would reduce the time lag for critical results. The more complex a rule a critic animates, the more likely it is that the critic will lag behind. In the case of a critic embodying a data flow rule, the time lag could quite significant. Here the critic might have to reanalyze large numbers of procedures that directly and indirectly invoke a lower level procedure which had been altered.

This issue of efficiency and rapid response seems to be central in considering architectural alternatives for implementing critics. The foregoing indicates that critics can embody rules which may require extensive preparatory analysis of unexpectedly wide ranging procedures. Under these circumstances it seems prudent to offer users control over how often critics are called upon to carry out their analyses. Certainly critics embodying complex rules ought not to be automatically invoked after every keystroke. Thus it seems that if critics were imbedded in intelligent editing systems there ought at least to be some user control over how often critic rules were to be applied. Some early experience with intelligent editors already indicates, for example, that critics embodying static semantic rules ought to be suspended until after the user has completed a multi-line editing procedure embodying a non-trivial change. Certainly it is possible to create an intelligent editor which embodies the ability to define and implement arbitrarily complex critic rules and to specify the rate at which the rules are to be applied. It seems to us that such a system would necessarily have to be implemented as a loose confederation of smaller analytic capabilities all driven by an interactive editor and an intelligent scheme for reuse of intermediate analytic results. We regard this as being functionally equivalent to the Odin-based approach we have already described.

The issue of whether the most effective user interface is the one presented by an intelligent editor or by a looser tool confederation, such as is suggested by Odin, is one well worth exploring, but does not seem central to investigation of the power and desirability of the notion of critics. On the other hand the issue of how to control the effectiveness of critics through control of granularity of application seems central.

A critic can be defined to work at any level of granularity. For instance, a critic whose rule concerns programs could work on an entire program, a subprogram, a block, or a statement at a time. A critic whose rule concerns regular text could work on an entire document, a section or chapter, a paragraph or a sentence at a time. However, if the level chosen is either too fine or too coarse, the resulting critic is not as useful as it might otherwise be. For programs written in a block structured language, for example, a block seems to be the right granularity. A critic on whole programs has too coarse a level of granularity. Such a critic could lag considerably behind the editing changes, because it could not begin until the entire program were written. A critic on individual statements has too fine a level of granularity. Such a critic would be an annoyance because the number of its reports (equal to the number of statements) is

likely to be large. Such a critic may also find relatively many of its results becoming obsolete as a program is edited. In contrast, a critic on blocks would begin its work earlier than a program critic, and would give more immediate feedback. On the other hand, it would begins its work later than a statement critic, and would give less, more relevant feedback.

A critic must do its work in an unobtrusive manner. A critic whose level of granularity is too fine will, most likely, waste resources and will frequently annoy users with small results that are likely to change. A critic whose level of granularity is too coarse will not give feedback as often and immediately as is desirable. Proper levels of granularity will be discovered for critics on various objects as the critics are developed and used.

Critics may be most useful, and in fact may work best, for large programs. Complexity increases with program size, so it is during the production and maintenance of large programs that analysis itself is most useful. Furthermore, it is during the production (or maintenance) of a large software system that the active nature of critics will provide the most leverage. When a system is composed of many objects (subprograms, modules, data types and so forth), it would be easy to neglect requesting the analysis of each object. It would be easy to neglect reanalysis when appropriate. These problems are compounded when a group of people are involved with a single project.

4. Specification of Critics

Software engineers and programmers need and/or perform many analyses of their programs, specifications, and designs every day. Some of these analyses have tool support (syntax analysis by compilers). Other analyses are done visually if at all (data flow analysis tools exist but are not used routinely). A goal of producing critics is to make analysis readily available and relatively efficient so that analysis will be commonly used as a conceptual lever against complexity. One approach to providing a user community with critics would be to handcraft individual critic fragments. However, producing a large collection of critics would be a very time consuming task; furthermore, any given collection of critics will certainly not contain all useful analysis. A second approach is to use specifications of critics either interpretively or to generate critic fragments.

The specification of critics has several important aspects. The specification of the rules themselves is, of course, a central issue. The design rules animated by a critic must be relatively easy to specify. Equally important are the specification of the applicability (what objects are criticized) and granularity of each rule, the specification of report frequency (interaction of critic and user) and the activation and deactivation of critics (setting a critic into action and turning it off).

Central to the task of designing a critic specification language is to design a notation for specifying the rules themselves. Some thought has been given to this problem. It appears that rules often prohibit or endorse *sequences of events* [Taylor 84]. For instance, a set of data flow rules might include "each variable, **x**, must be defined before it can be used," and "each assignment to a variable, **x**, must be followed by a use of **x** before another definition of **x**." A set of rules concerning stacks might include "each call to **Pop** must be preceded by a call to either **Push** or **IsEmpty?** with no intervening call to **Pop**". Of course, some design rules, such as the rule of thumb

concerning length of subprogram source text, do not describe sequences of events. The specification language must allow the statement of these design rules as well.

In addition to design rules, a critic specification language must allow the specification of several other types of information. Each design rule is applicable to a certain kind of object. The specification language will supply a method of specifying the number and types of the arguments to a critic. Each design rule will best work at a certain level of granularity (as discussed in section 3). The specification language will provide a method of specifying the level of granularity for each design rule.

The specification language discussed so far is one that would be used by the creator of critics. Some aspects of the specification of critics are best left to the control of the eventual user. Thus, there must also be a language whereby a user can control the critics. One aspect of the specification of control that must be included concerns frequency and medium of critical reports. Critics could report their findings immediately, at specific times during the day, via messages left in a mailbox or on demand. Critical reports could appear on a terminal or workstation screen, as automatically produced hard copy, or in a mailbox. Since choosing the most suitable frequency and medium of critical reports depends on the style of each particular user, these control aspects seem appropriate for a user profile.

A second aspect of the specification of critics that should be under the control of the user is the activation and deactivation of individual critics. While the essence of critics is that they are zealous assistants working in the background to bring timeliness to the analysis of software objects, there are legitimate reasons why critics should be turned off. A member of the office staff might be typing a paper in a language not understood by the text critics. Rather than have the spelling critic reject every word, for instance, it would be best to turn that critic off.

5. Conclusions

Software production is a complicated activity, and results in a number of complicated objects. While analysis tools are promising as an approach to the control of this complexity, they have never become commonly used.

Critics are active, incremental, analysis tools. Both features that distinguish critics from more traditional software tools are intended to provide a user with a great deal of analytic power in a relatively effortless fashion. An analytic tool that works in the background makes it possible for a user to continue doing useful work rather than waiting for requested analytic results. An analytic tool that works in an incremental fashion can give more immediate feedback concerning possible problems.

At the University of Colorado, we are at work on a prototype implementation of a critics environment. This environment will be based upon the Odin integration strategy. The prototype version will include a minimal set of critics. From this prototype, we will see the impacts made to the Odin architecture by the addition of active tools. Depending on the impacts seen, this may lead us to a different architecture for future versions. Future additions include an improved user interface and a language for specifying granularity, activation and deactivation times and report frequency.

Our future plans include a metacritic or analyst that looks at the reports of many critics and draws conclusions.

Acknowledgments

We wish to thank J. C. Browne for encouraging us to investigate active tools. We have also benefited greatly from discussions with users and implementors of intelligent editors, as these discussions have helped us to focus more effectively on the importance of the issue of granularity in considering how best to implement critics.

References

[ALRM] Ada Joint Program Office. *Ada Programming Language Reference Manual,* ANSI/MIL-STD-1815A-1983, 1983.

[Bell 77] Bell, T. E., Bixler, D. C., and Dyer, M. E., "An Extendable Approach to Computer-Aided Software Requirements Engineering," *IEEE Transactions on Software Engineering SE-3,* 1 (January 1977), pp. 49-60.

[Clemm 84] Clemm, G. M., "ODIN - An Extensible Software Environment Report and User's Manual", University of Colorado at Boulder, Computer Science Department Technical Report CU-CS-262-84, (May 1984).

[Cline 83] Cline, A. K., and Rich, E., "Building and Evaluating Abstract Data Types", University of Texas at Austin, Computer Science Department Technical Report TR-83-26, (December 1983).

[Cooprider 78] Cooprider, L. W., "Representation of families of Software Systems", Ph.D. Dissertation, Carnegie-Mellon University, 1978.

[Delisle 84] Delisle, N. M., Menicosy, D. E., and Schwartz, M. D., "Viewing a Programming Environment as a Single Tool", *Proceedings of the Software Engineering Symposium on Practical Software* Development Environments, Pittsburgh, (April 1984), pp. 49-56.

[Fischer 84] Fischer, G., Lemke, A., and Schwab, T., "Active Help Systems" in Green, T., Tauber, M., and van der Vee, G. (editors) *Proceedings of the Second European Conference on Cognitive Ergonomics - Mind and Computers,* Springer Verlag, Heidelberg - Berlin - New York, 1984.

[Fosdick 76] Fosdick, L. D., and Osterweil, L. J., "Data flow analysis in software reliability", *Computing Surveys 8,* 3 (September 1976), pp. 305-330.

[Johnson] Johnson, S.C., "Lint, A C Program Checker", *UNIX Programmer's Manual.*

[Leblang 84] Leblang, D. B., and Chase, Jr., R. P., "Computer-Aided Software Engineering in a Distributed Workstation Environment", *Proceedings of the ACM Symposium on Practical Software Development Environments,* Pittsburgh, (April 1984), pp. 104-112

[Luckham 84] Luckham, D. C., and von Henke, F. W., "An Overview of Anna, a Specification Language for Ada," *Proceedings of the Conference on Ada Applications and Environments,* St. Paul, (October 1984), pp. 116-127.

[Osterweil 76] Osterweil, L. J., and Fosdick, L. D., "DAVE - A validation, error detection and documentation system for Fortran programs", *Software Practice and Experience 6,* (1976), pp. 473-486.

[Osterweil 82] Osterweil, L. J., "Toolpack - An Experimental Software Development Environment", *Proceedings of the Sixth International Conference on Software Engineering,* Tokyo, (1982), pp. 166-175.

[Osterweil 83] Osterweil, L. J., "Toolpack - An Experimental Software Development

Environment Research Project", *IEEE Transactions on Software Engineering SE-9,* 6 (November 1983), pp. 673-685.

[Reiss 84] Reiss, S. P., "Graphical Program Development with PECAN Program Development Systems", *Proceedings of the ACM Symposium on Practical Software Development Environments,* Pittsburgh, (April 1984), pp 30-41.

[Reps 83] Reps, T., Teitelbaum, T., and Demers, A., "Incremental Context-Dependent Analysis for Language-Based Editors", *ACM Transactions on Programming Languages and Systems 5,* 3 (July 1983), pp. 449-477.

[Taylor 84] Taylor, R. N., and Osterweil, L. J., "Analysis and Testing Based on Sequencing Specifications", *Proceedings of the 4th Jerusalem Conference on Information Technology,* Jerusalem, Israel, (May 1984).

[Taylor 83] Taylor, R. N., "A General-Purpose Algorithm for Analyzing Concurrent Programs", *Communications of the ACM 26,* 5 (May 1983), pp. 362-376.

[Taylor 80] Taylor, R. N., and Osterweil, L. J., "Anomoly detection in concurrent software by static data flow analysis", *IEEE Transactions on Software Engineering SE-6,* 3 (May 1980), pp. 265-278.

[Teichroew 77] Teichroew, D., and Hershey III, E. A., "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems", *IEEE Transactions on Software Engineering SE-3,* 1 (January 1977), pp. 41-48.

[Teitelbaum 81] Teitelbaum, T., and Reps, T., "The Cornell Program Synthesizer: A syntax-directed programming environment," *Communications of the ACM 24,* 9 (September 1981), pp. 563-573.

[Teitelman 81] Teitelman, W., and Masinter, L., "The Interlisp Programming Environment", *Computer 14,* 4 (April 1981), pp. 25-34.

[Waters 82] Waters, R. C., "The Programmer's Apprentice: Knowledge Based Program Editing", *IEEE Transactions on Software Engineering SE-8,* 1 (January 1982), pp. 1-12.