FEEDBACK-DIRECTED DEVELOPMENT OF COMPLEX SOFTWARE SYSTEMS

Jack C. Wileden
Lori A. Clarke

COINS Technical Report 84-06
April 1984

(To appear in: Proceedings of Software Workshop, Runnymede, England, 1984

Software Development Laboratory
Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003

ABSTRACT

The phrase feedback-directed development refers to a research program whose goal is improved approaches to the software development process. In this paper we give an overview of that research program. We discuss shortcomings in existing approaches and motivate the two focal points of our research on feedback-directed development:

- appropriate, consistent sets of abstractions and
- analysis (or feedback) techniques applicable throughout the software development process.

We also indicate how several of our ongoing projects in software technology contribute to the feedback-directed development research program.

1. INTRODUCTION

Powerful and sophisticated software development environments are urgently needed. The ever-increasing demand for complex software systems, particularly those supporting concurrent or distributed processing under real-time constraints, accentuates this need. Existing software development support consists primarily of high-level programming languages, such as FORTRAN, Pascal, or Ada*, and an assortment of diverse, unrelated tools, such as compilers and editors. Programming support environments (PSEs), such as the Programmer's Work Bench [Ivie77] or the proposed Minimal Ada Programming Support Environment (MAPSE) [Buxt80], are a first step toward more sophisticated support for software development activities. But these PSEs only offer support for the program implementation phase of software development. This must be augmented by additional tools supporting the <u>pre-implementation</u> activities of requirements, specification, and design development and the <u>post-implementation</u> activities of code analysis, testing, and maintenance, before the result can appropriately be called a software development environment (SDE).

We do not believe that simply extending the coverage of existing PSEs by adding pre- and post-implementation languages and tools will result in the highly integrated, powerful SDEs required for developing complex software. This is because the standard approaches to software development, on which existing PSEs are based, have two crucial shortcomings:

- Lack of appropriate, consistent sets of abstractions

*Ada is a registered trademark of the U.S. Government (Ada Joint Program Office)

- Inadequate availability of analysis during most phases of the software development process

We view appropriate and consistent sets of abstractions as the key to integration in the software development process. Having appropriate sets of abstract concepts for describing various aspects of a computational system is of crucial importance to software developers. The lack of suitable abstractions for such things as concurrency, real time constraints and exception handling is no doubt due largely to their relatively recent emergence as important features of software systems. Unfortunately, appropriate abstractions are also lacking for other, less recently recognized concepts. An example is visibility control, which is inadequately supported in many languages (e.g., FORTRAN) and inelegantly supported with a hodgepodge of constructs, rules and exceptions to those rules in other languages (e.g., Ada). Even the most appropriate abstractions will be of limited value to software developers, however, unless they can use the same set, or closely related sets, of abstractions for describing a software system during each phase of the development process. Such consistent sets of abstractions are conspicuously lacking in current approaches to software development, where at each phase of development (e.g., specification, design, program) a software system is described in terms of vastly different sets of concepts. Providing consistency among the abstractions used in describing a software system during different phases will minimize discontinuities in the development process. As a result, development can be a smooth, consistent progression through successively less abstract descriptions of an evolving software system. Moreover, appropriate and consistent abstractions will facilitate the development and use of analysis techniques that can be uniformly applied during all phases of software development.

Although there do exist some methods for analyzing programs, and even a few for analyzing pre-implementation descriptions, the systematic use of analysis to guide software development has received little attention. To be most beneficial, analysis techniques should be applied early in the software development process when decisions are first being formulated and when errors are most likely to be introduced. The use of powerful analysis techniques early in the development process enables developers to explore alternative approaches to structuring a complex software system. Such <u>exploratory development</u> is a very attractive paradigm for the construction of complex software, since it can increase the developer's confidence in the quality and eventual success of the software system. To ensure that quality is maintained, it is important that corresponding analysis tools also be available later in the development process when coding details are added or modifications are made.

In an effort to overcome the shortcomings in current approaches to software development, we have been pursuing an approach that we call <u>feedback-directed development</u>. An SDE supporting this approach would employ <u>consistent abstractions</u> and provide tools supporting <u>analysis</u> (i.e., feedback about the quality of the system being produced) <u>throughout the software development process</u>. The consistent abstractions would provide a basis for thorough integration of the languages and tools constituting an SDE. In particular, they would facilitate the application of nearly identical analysis tools to descriptions at various levels of abstraction and detail. By providing analysis capabilities that are available throughout the software development process, the environment would support continual reasoning about the properties of an evolving software system.

This ongoing reasoning would guide the SDE user in evaluating decisions, exploring alternatives and smoothly progressing toward a completed software system. In some cases, this reasoning might be performed by the environment itself. In other cases, the environment would simply augment the reasoning abilities of its human user by providing insightful information. But always the environment's consistent abstractions and its omnipresent analysis capabilities would provide the basis for that reasoning, and hence for an exploratory, feedback-directed development process.

2. RELATIONSHIP TO OTHER APPROACHES

Feedback-directed development extends and complements the two other primary approaches to SDE integration, which we refer to as the uniform interface toolset approach and the transformational development approach.

Uniform interface toolsets (e.g., Interlisp, Smalltalk, Programmers Workbench [Ivie77], TOOLPACK [Oste82]) attempt to achieve integration by providing a single standard user interface to a collection of software development tools. The interface may be a common command language or even a graphical interface. Uniform interface toolsets may also employ uniform interfaces among the tools themselves, i.e., common internal representations of the information that the various tools can produce and manipulate. While a uniform and user-friendly interface is an important aspect of an SDE, it will only provide surface level integration. If the tools in these toolsets are not based on consistent sets of abstractions, shifts in perspective and discontinuities in descriptions are likely to arise as development proceeds. These shifts and discontinuities force users to perform the necessary translations, and hence they invite the introduction of errors.

We use the term "transformational development" to encompass a variety of development approaches, all of which are based on the automated transformation of specifications into executable programs. Balzer and Cheatham [Balz81] have pursued such an approach, while similar, if less ambitious, approaches have been dubbed "Very High Level Languages" or "application generators". Transformational development seems very suitable for use in developing software for limited, well-understood problem domains, where standard data representations and processing techniques are known to be both applicable and reasonably efficient. It is not at all clear that transformational development is as suitable for developing novel solutions or for producing software for novel applications, where the major challenge is to discover appropriate data representations and algorithms. The generality of the transformations that would be required to develop reasonable software in the general case of novel applications far exceeds the transformation repertoire that can be anticipated in the foreseeable future. Feedback-directed development, on the other hand, exploits and augments the creativity of human software developers. While letting the developers themselves define the transformations to be performed, the feedback-directed approach supports and guides them by providing feedback (i.e., the outcome of analysis) on the results of the development steps that they choose to make.

3. TOWARD FEEDBACK-DIRECTED DEVELOPMENT

We are presently working toward the construction and evaluation of a prototype SDE supporting the feedback-directed approach to software development. This prototype will be based upon three promising foundations for feedback-directed development that have emerged from our work on software development technology. These are:

- language constructs and analysis tools supporting precise description of modularity and interfaces within a software system [Clar83]
- the constrained expression framework for description and analysis of behavior in concurrent/distributed systems [Avru83b], [Wile82]
- rigorous and systematic testing methods applied throughout the software development process [Rich81,82]

As will be explained in more detail in the remainder of this section, each of these represents an instance of a consistent set of abstractions. Each also provides a suitable basis for analysis techniques and tools that could generate the information needed for feedback-directed development. Thus each of these areas is, in itself, an appropriate vehicle for studying feedback-directed development. In addition, our work on each of the areas will result in tools and techniques that will be of independent interest and value, while the integration of all three areas will permit a more realistic exploration and evaluation of the approach.

In addition to tools specific to the three domains enumerated above, the prototype SDE will necessarily include traditional tools, such as an editor and a compiler. Indeed, one of our objectives is to explore the role of such traditional tools in feedback-directed development. For instance, given a set of consistent abstractions spanning the phases of software development, one can conceive of truly smart editors, which are not simply syntax-directed (e.g., Cornell Program Synthesizer [Teit81], Gandalf [Habe79]) but instead can provide feedback and guidance and are also uniformly applicable throughout the development process. Thus we see inter-operability and tool cooperation as valuable byproducts of our approach.

We conclude our discussion of the feedback-directed development research program by briefly describing our work in each of the three foundations areas enumerated above.

3.1 Modularity and Interfaces

Since Algol60, modern programming languages have primarily relied on nesting as a means of defining modularizations and specifying the interrelationships of entities in a software system. Nesting, however, is both an overly restrictive and an imprecise basis for describing modularization and entity interrelationships, since it imposes a rigid tree-like structure that permits only limited patterns of interrelationships. Moreover, nested software systems are notoriously difficult to maintain. As a result, many recently introduced languages have incorporated additional mechanisms to overcome nesting's shortcomings, while specification languages have generally eschewed nesting altogether.

We have been exploring an alternative approach to describing modularization and entity interrelationship, along with associated techniques for analyzing these aspects of a software system's structure [Clar83]. This approach shuns nesting and instead builds upon a carefully selected set of existing encapsulation and import/export concepts. The result is a mechanism capable of describing a software system's modularization and the interrelationships among its entities more precisely and flexibly than is possible with any existing language. Both our subjective assessment and a collection of theorems derived within a formal model of software system structure [Wolf83] testify to the superiority and generality of this approach.

Our mechanism for describing modularization and entity interrelationship is representative of the appropriate and consistent abstractions that are required by feedback-directed development. We believe that it is well-suited for use at every stage in the software development process, from specifications and design to programming and maintenance. Moreover, the associated analysis techniques are uniformly applicable and valuable throughout development. Thus one aspect of our feedback-directed development research program is to explore the use of this mechanism as a basis for feedback-directed development.

3.2 Constrained Expressions

A variety of approaches for describing concurrent/distributed systems have been proposed, from Petri nets to temporal logic to CSP and Ada. Yet none of these has provided a truly unifying perspective on the behavior of concurrent or distributed systems. Each has been primarily applicable to a limited segment of the software development process. Few have been associated with useful analysis methods. In short, neither the appropriate set of consistent abstractions nor the analysis methods necessary to support feedback-directed development of concurrent/distributed software currently exists.

The constrained expression framework that we have been developing holds great promise as a basis for feedback-directed development of this important class of systems. In this framework, the possible behaviors of a concurrent/distributed system are regarded as a set of sequences of events. Typically these events correspond to internal activities of the individual processes in the system or to communication among those processes. A constrained expression is a closed-form representation of this set of event sequences [Wile82]. We have shown that a constrained expression representation can be derived from descriptions of concurrent/distributed systems written in a variety of other notations, such as Petri nets, CSP and the DYMOL design language [Wile80]. Thus constrained expressions can be used as a standard representation of the behavior of concurrent/distributed systems that is largely independent of the language in which they were originally described. We have also used the constrained expression framework as the basis for a high level, abstract specification language supporting stepwise refinement of the description of a concurrent/distributed system's behavior. This language has proven to be valuable not only in developing but also in debugging concurrent/distributed systems [Bate83].

The use of constrained expressions for description is complemented by analysis techniques based on formal manipulation of the expressions. Among these are expression simplification and algebraic techniques for determining whether a particular event or set of events can be realized by the closed-form representation. The algebraic techniques are based upon a collection of rules for iteratively generating a set of inequalities. The generated inequalities describe the number of occurrences of particular events at various stages in the behavior of a concurrent/distributed system [Avru83a]. The rules are based on the events in question and the underlying semantics of the concurrent/distributed system as captured by its constrained expression description. Beginning with the assumption that the particular set of events in question does occur in a behavior of the system, the rules are applied to the constrained expression representation of that behavior to generate inequalities. If at any stage these inequalities are inconsistent, then there is a contradiction, showing that the original assumption was incorrect and the postulated set of events does not occur in any behavior of the system. Otherwise, additional inequalities are generated until there is enough information to construct a behavior with the desired properties.

Having applied these analysis techniques to several distributed software design problems, we believe that they provide a method for generating feedback that holds considerable promise. For example, this technique easily detected a subtle error in a published solution to the distributed mutual exclusion problem [Avru83b].

Like our mechanism for modularization and entity interrelationship description, constrained expressions offer both an appropriate, consistent set of abstractions and also analysis techniques for generating feedback. Thus a second aspect of our program is to explore the use of constrained expressions as a basis for feedback-directed development.

3.3 Rigorous and Systematic Testing

While it is well known that testing, like all validation and verification methods, cannot guarantee correctness, testing remains an important form of analysis. Applied in appropriate combinations, and with suitable automated support, testing techniques should be able to detect a wide class of errors or guarantee their absence. Moreover, with appropriate modification we suspect that many of the known techniques for testing programs can be applied to the more abstract and possibly incomplete descriptions found in specification and design. Hence we view testing itself as an appropriate and consistent abstraction contributing to the feedback-directed development approach, as well as a valuable analysis methodology.

Over the last ten years there has been considerable research addressing the problems of testing. This research has matured, from early efforts predominantly focused on methods for gathering information about programs, to more recent work that is primarily concerned with developing techniques that actually apply this information. Moreover, there has emerged a deeper awareness and understanding of the theoretical limitations of the various techniques. Very little work has been done, however, on understanding the comparative strengths and weaknesses of various testing techniques or their potential interaction. We have recently begun to study these issues and we anticipate that the result will be a systematic and rigorous approach to testing that can serve as one of the foundations for feedback-directed development.

The application of testing methods to pre-implementation stages of the software development process is another area that has received limited attention to date. In our research on the Partition Analysis Method [Rich81,Rich82], however, we have been investigating the use of information from both the specification (or design) and the implementation to verify consistency between the two descriptions and to astutely select test data to exercise the important characteristics of both. The basic technique involves symbolically executing both a procedure's implementation and its specification (design). The results from these symbolic executions provide two closed-form representations of the same procedure. Each representation gives a decomposition of the procedure's input domain. By intersecting the two decompositions, procedure subdomains are formed. It can be argued that these subdomains are the largest units that can be analyzed independently and yet are the smallest units into which a procedure can be practically decomposed. By comparing the computations associated with each subdomain and by evaluating the boundaries of these subdomains, error sensitive test data is selected.

An initial evaluation of this method has shown it to be very effective for detecting errors [Rich82]. It has several additional benefits as well. For one, it is one of the few techniques to combine testing and verification. Also, unlike typical verification methods, partition analysis is applicable to a number of different pre-implementation languages. It can be applied to many of the more esoteric, high-level specification languages as well as to low-level designs. For suitable languages, it can be applied to any two descriptions of a software system such as a specification and design or high- and low-level design descriptions.

Here again we have both a consistent abstraction and a class of analysis techniques for generating feedback. Thus the third aspect of our research program is to explore the use of rigorous and systematic testing as a basis for feedback-directed development.

4. SUMMARY

In this paper we have described the feedback-directed development research program, compared it to other approaches aimed at integrated software development environments and discussed the foundations of our current work on feedback-directed development. Construction of a prototype SDE based on the approach, and subsequent experience with its use, are expected to provide us with more insight regarding the potential value of the feedback-directed approach to software development.

REFERENCES

[Avru83a] Avrunin, G.S. and Wileden, J.C., "Algebraic Techniques for the Analysis of Concurrent Systems," Proceedings of the Sixteenth Hawaii International Conference on Systems Sciences, Vol.1, January 1983, pp.51-57.

[Avru83b] Avrunin, G.S. and Wileden, J.C., "Describing and Analyzing Distributed Software System Designs," University of Massachusetts, Department of Computer and Information Science, Technical Repaort 83-28, August 1983, (submitted for publication).

[Balz81] Balzer, R.M. and Cheatham, T.E., "Editorial: Program Transformations," IEEE Transactions on Software Engineering, SE.7, 1, January 1981.

[Bate83] Bates, P.C., Wileden, J.C., and Lesser, V.R., "A Debugging Tool for Distributed Systems," Proceedings of the Phoenix Conference on Computers and Communications, March 1983.

[Buxt80] Buxton, J., "Requirements for Ada Programming Support Environments," ("Stoneman"), Department of Defense, February 1980.

[Clar83] Clarke, L.A., Wileden, J.C. and Wolf, A.L., "Introduction to a Unified Treatment of Interface Control and Program Structure," University of Massachusetts, Department of Computer and Information Science, Technical Report 82-32, January 1983.

[Habe79] Haberman, A.N. "An Overview of the Gandalf Project," Computer Science Research Review, 1978-79, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1979.

[Ivie77] Ivie, E., "The Programmer's Workbench - A Machine for Software Development," Communications of the ACM, 20,10, October 1977, pp.746-753.

[Oste82] Osterweil, L.J., "TOOLPACK — A Software Tool Environment Research Prototype," The Fourth Israel Conference on Quality Assurance, October 1982.

[Rich81] Richardson, D.J. and Clarke, L.A., "A Partition Analysis Method to Increase Program Reliability", Proceedings of the Fifth International Conference on Software Engineering, March 1981.

[Rich82] Richardson, D.J. and Clarke, L.A., "On the Effectiveness of the Partition Analysis Method," Proceedings of the IEEE Sixth International Computer Software and Applications Conference, November 1982.

[Teit81] 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.

[Wile80] Wileden, J.C., "Techniques for Modelling Parallel Systems with Dynamic Structure," *Journal of Digital Systems*, 4,2, Summer 1980, 177-197.

[Wile82] Wileden, J.C., "Constrained Expressions and the Analysis of Designs for Dynamically-Structured Distributed Systems," *Proceedings of the International Conference on Parallel Processing*, August 1982.

[Wolf83] Wolf, A., Clarke, L.A. and Wileden, J.C., "A Formalism for Describing and Evaluating Visibility Control Mechanisms," University of Massachusetts, Department of Computer and Information Science, Technical Report 83-34, October 1983.