Evaluating Educational Software Environments Using a Model Based on Software Engineering and Instrucfional Design Principles

Betty A. Collis
Marilyn Gore

Abstract: This study suggests a new model for the evaluation of educational software authoring systems and applies this model to the evaluation of a particular authoring system. The model used for the evaluation is based on an integrated set of software engineering and instructional design principles. We argue that the model is appropriate as a basis for the evaluation of authoring environments in that it does not consider authoring materials as isolated entities, but rather, evaluates them in terms of how well they integrate with and facilitate an overall process of courseware design.

Production of educational software is expensive and time-consuming. The majority of educators involved in instructional software development typically lack an appropriate background in computer science and hence require considerable technical or authoring support. In addition, those educators who do have technical expertise also seek specialized authoring support in order to facilitate the software development process in both time and scope. In response to this general need, a considerable amount of research and commercial activity focuses on the design, attributes, production and evaluation of authoring support environments. For example, on the program for the March 1985 ADCIS conference held in Philadelphia there were more than 20 sessions focussing on authoring systems or authoring languages, while the program for the May 1986 Canadian Symposium for Instructional Technology featured approximately 30 such sessions. Many authoring materials are developed by their eventual users, especially in large-scale software production research and development units. However, a continually-increasing number of commercially available authoring packages are being marketed toward the individual educator, who wishes to develop some specialized C.A.I. but who does not have the support of a professional development team. In this paper we present an approach to the evaluation of authoring environments that is particulary appropriate in situations where the educator is not part of a large-scale production group and is not able to be involved in a substantial long-term investment of training time in the package itself.

Betty A. Collis is an Associate Professor in the Department of Psychological Foundations, Faculty of Education, University of Victoria, Victoria, BC and **Marilyn Gore** is Programming Manager for the IBM Canada - University of Victoria Software Engineering Education Cooperative Project. This work was done with the support of the IBM Canada - University of Victoria Software Engineering Education Cooperative Project.

DEFINITION OF 'AUTHORING ENVIRONMENT'

We define the general category of authoring environments as including at least three subcategories of support tools, with considerable overlap between the three groups. Although classifications and nomenclature vary considerably among implementations, these three groups may be defined as a) authoring languages, b) authoring systems, and c) authoring tool kits.

Authoring languages are programming languages used for instructional software development, with at least 26 identified as languages developed specifically for courseware development (Brahan, Farley, & Orchard, 1983). The number of these languages continues to grow. BASIC, C, Pascal, and LISP are examples of general purpose programming languages which are used for educational software authoring, while Pilot and Natal are examples of authoring languages with specialized features for instructional software development. Authoring languages are not based on any specific instructional development model. All of the instructional sequencing in a program, as well as the particular content, needs to be specified by the ‘developer. The price of this flexibility is complexity. Programming languages, including so-called authoring languages, require a considerable investment of technical training and expertise to use at all. To use them intelligently and efficiently, we believe, requires the application of computer science perspectives and software engineering principles.

Authoring systems may be defined as preprogrammed sets of procedures from which courseware can be developed (Cyrs, 1985). These sets of procedures can be rigidly defined around a particular instructional strategy and presentation format or may vary in the flexibility and options available to the author. Their goal is to automate the programming aspect of courseware production by producing an executable program based on the dialogue maintained between the system and the developer (Quick, 1985). Although their particulars vary greatly, commercially available authoring systems share the goals of “allowing novices to easily create computer-based instructional courseware without programming knowledge” (Cyrs, 1985, p. 3) and of reducing the time required to produce executable code. In theory, authoring systems function “as a mediational link between instructional logic and computer logic” (Jonassen, 1985, p. 40), and supposedly allow the instructor to concentrate on the design of interactive instructional experiences which will take effective advantage of the capabilities of the computer. Estimates vary concerning the number of authoring systems available, because new or modified systems “are appearing weekly on the market” (Cyrs, 1985, p. 1) and many independent research projects involving the development of authoring systems are in place in university research and development centers (see for example, Hunka, 1986; Szabo, Jensen & Bagnall, 1986).

Authoring tools are less well-defined. They may include modules or other utilities provided in subroutine libraries which can be reused in various software development projects, streamlining the particular project by allowing its more predictable components to be based on preprogrammed utility modules (Wilson & McCrum, 1984; Collis Gore, 1986). While this approach is common in large-scale software development centres where the technical experts involved in the projects have the skill and understanding to develop specialized software tools, it is relatively uncommon in commercial materials aimed at the individual educator-author. This is probably because its use requires a level of technical programming knowledge inconsistent with a general goal of automating the production process for the novice courseware developer.

EVALUATING AUTHORING ENVIRONMENTS

Although most of the literature on authoring environments tends to be descriptive in nature (or to have as its purpose the conveyance of an enthusiastic endorsement of a particular authoring approach) a few studies have focussed on the evaluation of authoring languages and systems. Burger (1985) evaluated 12 *authoring languages/systems* on the basis of various attributes relating to lesson delivery, student management and production support. Lesson delivery attributes (evaluated by Burger on a 'yes-no' basis) include provision for automatic line spacing in text displays; availability of colour for text displays; availability of primitive graphic forms, sound, and animation capabilities; availability of various feedback characteristics; acceptance of string and numeric variables; and various answer processing attributes, such as timing, wild card designation and range specification. Student management attributes include security of student records and access, and availability of storage and analysis options relative to student records. Production support criteria include provisions for text and graphics editing, timing and complexity of movement between authoring and testing modes, and thoroughness of documentation. Cyrs (1985) established criteria relative to vendor reputation, purchase and licensing costs, documentation, equipment requirements, re-entry provisions when moving between authoring and testing modes, input manipulation, availability of various test formats and questioning strategies, record keeping provisions, portability, windowing capabilities, graphics, text-related variables, and presumed target audience.

Barker (1986) used a similar attribute-based framework in his checklist of 66 features to consider in the evaluation of "author languages for CAL." Barker and Singh (1983) based an evaluation of a particular authoring language on the ratings of various groups of educators who used the language in a workshop environment and were asked to rate the experience in terms of the value of the demonstration materials, ease of use, and likelihood of wishing to use the language again. Merrill (1985) discussed advantages and disadvantages of authoring systems and languages in a particularly useful and well-informed critique. Merrill's work differs from the other evaluation schemes in that he also discussed how the *frame assumption* inherent in the model of interaction supported by most authoring systems places limitations on the instructional design of the courseware itself. Merrill concluded his'critique by noting that "none of the existing authoring systems provide any assistance in the authoring or instructional design process" (p. 96). Overall, however, most evaluations of authoring support seem to reflect a framework emphasizing an assortment of possible implementation attributes but other than Merrill's, give no particular evidence of being based on a more theoretical consideration of how well the system mediates a particular model of educational software development.

THE COLLABORATIVE MODEL OF EDUCATIONAL SOFTWARE DEVELOPMENT

The collaborative model is one such model for educational software development (Gore & Collis, 1986). This model reflects the integration of three sets of principles:

A) Principles Reflecting the Needs of the Software Engineer

1) Using documentation as a software design medium.

2) Writing software requirements that are as complete as possible.
3) Structuring the development process so that distinct times are provided for planning and evaluation prior to implementation.
4) Partitioning software projects into components that can be developed and tested independently.
5) Designing systems so that they are easily contracted or extended.

B) Instructional Design Principles

1) Identifying intentions of the material relative to learning objectives, target learners and overall instructional strategy.
2) Delimiting the scope and sequence of the instructional content.
3) Specifying the methodological decisions involved in delivering the content.
4) Establishing evaluative processes for both the learner and the designer/instructor.

C) Collaboration Principles

1) Designing through iteration and compromise.
2) Assessing congruency at specific points in the development process.
3) Minimizing the common knowledge necessary for all participants and documenting this shared knowledge in a *collaborative filter.*

The procedure which operationalizes the collaborative model has been expressed by a timeline with seven phases. The model and its 7-phase timeline have been used in a variety of collaborative educational software developments at the University of Victoria under the support of the IBM Canada-University of Victoria Software Engineering Education Cooperative Project and is proving to be an efficient and effective structure for both computer scientists and educators (Gore Collis, 1986).

AN EVALUATION PERSPECTIVE BASED ON THE COLLABORATIVE MODEL

If we assume that this integrated set of software engineering, instructional design, and collaboration principles can be taken as an appropriate basis for educational software development, we claim it is also reasonable to use the same set of principles as the theoretical foundation of an evaluation model for authoring environments, The overall criterion for such an evaluation becomes the extent to which the authoring environment -- language, system or toolkit -- supports and mediates a theoretically-based model of educational software development. Unlike evaluative procedures which are primarily based on attribute checklists, this evaluation model is based on software engineering and instructional design principles re-expressed as evaluation criteria, and focuses on authoring support within the larger context of design and development.

The overall question asked by this evaluation model is how well does the authoring environment facilitate or constrain the educator in the overall design and development process.

General Evaluation Criteria

The following questions illustrate this evaluation perspective. The word *system* is used throughout these questions instead of the phrase *authoring environment.* This is done for economy of expression although we are aware that the word *system* can also denote the higher-level interaction of people, machines, software and methodologies involved in software design and development. We expect that an evaluation of the sort we are describing would be more likely to be applied to a commercially available software package described as an *authoring system,* which a consumer may be considering purchasing, than it would to the more general categories of authoring language or toolkits. The questions are asked from the perspective of the educator/user who is not a technical specialist.

A. Principles Reflecting the Needs of the Software Engineer

A.1. Design through documentation. Does the system stimulate or encourage adequate attention to design before using the authoring package for technical implementation? Does the system stimulate the development of appropriate documentation?

A.2. Writing software requirements that are as complete as possible. Does the system stimulate or require the developer to produce the necessary and sufficient amount of documentation for each phase of the development process?

A.3. Structuring the development process so that distinct times are provided for planning and evaluation prior to implementation. For a single-developer, does the system emphasize a distinct time for design and evaluation as opposed to programming and compiling?

A.4. Partitioning software projects into components that can be developed and tested independently. Can the various components of the program be developed and tested independently, and linked together when convenient? Can components (modules) of the program be reused as parts of other programs?

A5. Designing systems so that they are easily contracted or extended. Does the system produce software that can be conveniently altered or extended in response to specific student needs? Does the system facilitate the development of a prototype of the extended program, for evaluation and refinement of design decisions?

B. Instructional Design Principles

Intents. Does the system in any way promote reflection and evaluation concerning the intentions and value of the program itself! Does the system, especially through its documentation, emphasize to the developer that decisions involving basic educational needs, objectives and user characteristics must be adequately addressed before other decisions are confronted?

B.2. Content. Does the system allow data to be treated independently from the main problem?

B.3. Methodology. Does the system allow the designer to postpone secondary methodological decisions until more fundamental decisions are confirmed? Does the system provide tools that help the designer to develop, test and implement his methodological decisions? Does the system unduly influence the designer, or constrain him from implementing his methodological decisions? (As an example, an educator using an authoring system that emphasizes multiple-choice, true-false, or matching templates may base his instructional plans around one of these methodologies without fully considering the

relevance of other strategies, such as simulations). Do the *secrets* of implementation prematurely or otherwise inappropriately restrict or constrain the conceptual development of the program by the educator?

B.4. Evaluation. Does the system support the variety and quality of feedback appropriate for the intended user? Does the system support the capture of information on student performance that the teacher defines as useful for subsequent analysis? Does the system facilitate testing and editing during the development process?

C. Collaboration Principles

C.l. Iteration and compromise. Does the system facilitate iterative and on-going compromises between various participants in a collaborative environment?

C.2. Congruency. Which of the participants in a collaborative team is the intended user of the system -- the educator or the computer scientist? Is the system appropriate for the intended user? How much training is required for the user of the system before it can be used as a facilitating tool rather than a focus of attention and effort?

APPLICATION OF THE EVALUATION CRITERIA TO A PARTICULAR AUTHORING SYSTEM

CSR Trainer 4000 is an authoring system produced by Computer Systems Research, Inc., 40 Darling Drive, Avon Park South, Avon, Connecticut 06001. This system was purchased for software development because it promised to be a "complete instructional system designed to meet the needs of CAI authors at all levels.,.flexible, structured, use, sophisticated, and a pleasure to work with" (Computer Systems Research, 1985, p. i). The overview of the system described it as "state of the art" and further indicated that novice users would be "amazed" at the ease of using the system while advanced authors "will be amazed at its power and flexibility." In order to evaluate their claims of ease and simplicity, we used the system as *is* and did not seek assistance from the Trainer 4000 office after an initial visit from their representatives. We felt that this would more realistically reflect the situation of the individual purchaser of a commercial package such as this which bases its appeal on being easy to use and appropriate for the novice user. The system includes five disks for authoring, presentation and administration and two manuals. Only one backup copy may be made of each of the disks. The system requires an MS-DOS environment with a minimum of 256K memory and two disk drives (or one drive and a fixed disk).

The system was used first by one of the researchers, an educator with experience in the design and development of educational software using both authoring languages and systems. Working together with a computer scientist , a short tutorial and drill program was designed and developed using the collaborative model and the Trainer 4000 system. Following this experience, nine students involved in a senior-level faculty of education course in educational software design and development were given a summary of the steps involved in using Trainer 4000 to generate a simple instructional program. The education students worked with the instructor and three computer scientists to translate a simple design into a program using Trainer 4000. All nine education students were experienced teachers and computer users and had previous experience in designing and developing educational software using the BASIC language.

Following this guided practice, the education students were divided into four groups and, as a course assignment, were required to design, a short educational program using the

documentation requirements and seven-phase timeline prescribed by the collaborative model, and to develop the program using only the Trainer 4000 system. The groups of students were given three weeks for this assignment. Each group of education students worked with a computer scientist during the production phases of the model. Students were encouraged to be modest in their plans for the program; four or five tutorial sequences each with one or two accompanying drill questions was suggested as a reasonable expectation of program scope given the shortness of course time available for the assignment. The students were experienced in designing simple programs and moved quickly through the planning phases of the model. The use of Trainer 4000, however, presented all the participants with considerable difficulties. Rather than itemizing the many particular difficulties however, the Trainer 4000 system will be evaluated using the criteria described previously in this paper.

A.1. and A.2. Appropriate Documentation

Only one sentence in the manuals of Trainer 4000 seems to suggest the concept of design through documentation -- "It will be helpful to completely outline your course." This admonition is not expanded, nor are the ideas of documentation and preplanning mentioned again.

Although principles A.1 and A.2 relate to the use of documentation during the development process, it is convenient to discuss documentation for the end user in the context of these principles. As a model of documentation in itself, the manual of Trainer 4000 is seriously flawed and was the basis of much of the frustration experienced by all who used the system. Although the manual was lengthy (approximately 400 pages of text in two looseleaf binders), pertinent information was frequently lacking or difficult to find. No complete example programs were provided and critical steps in the authoring procedure were not well-described. The "amazing flexibility and power" of the system seems to rely in part on its ability to utilize a particular mainframe authoring system (not available at our institution) and on the ability to build macros using something called the CSF language. This language, although alluded to frequently and necessary for designs above the template level, was not referenced or defined in the documentation. The documentation provided no useful progression of instructions for the user and was extremely difficult to interpret. We felt that instructions were unnecessarily verbose and repetitious in some instances and incomplete or missing entirely in others. We find it highly unlikely that any individual without extensive technical skill could interpret the manual or use the system beyond its most simplistic It proved a time-consuming enterprise for experienced software developers and computer scientists, and many major components of the system never were deciphered or operationalized. As models or stimulants of appropriate documentation, the Trainer 4000 manuals were, in our opinion, failures.

A.3. Premature Implementation

The Trainer 4000 system, used at its *novice* level, severely constrains the author to a pretest-tutorial-posttest model without options for branching or randomized question selection. The latter two options are available using the *more advanced* (and; to us, undecipherable) features of the system. For users who wish to utilize more than the novice features of the system, various *secrets* of implementation must be penetrated before production can continue, and this penetration requires working in a programming language environment which is neither conceptually easy or well-defined. Although the development system is organized around a hierarchical set of menus, the use of most of the options

available from the menus involves immediate grappling with the *secrets* of implementation. Because of their interdependent complexity, we fell the implementation features of Trainer 4000 may severely constrain the author in the development of the program itself because of the technical complexities they present throughout the development process.

A.4. and A.5. Partitioning and modifiability

As far as we could determine, and at the level at which we were able to use the system without additional documentation, the Trainer 4000 system requires the developer to construct his program as a linear sequence of frames before program generation and real testing can take place. Additional frames cannot be inserted within the sequence, and once the *final quiz* option is invoked, it does not seem possible (at least at the levels explained in the documentation) to add additional frames to the lesson. This constraint violates software engineering principles relating to modularization, reusability, and contractability. The various components of the program cannot be developed and tested as independent entities, nor can any of the text screens be reused in other programs.

These are, however, some aspects of the development process which do reflect the software engineering criteria relating to modularity and modifiability. Graphics screens are created separately, under a different editor, and saved as separate entities on a disk. They are called when required from within frames of the program itself. This means that each graphics screen is developed and tested independently and is available for subsequent reuse within the program or within other programs. In addition, Trainer 4000 reflects one aspect of modularity in that the developer can initially create each frame as an essentially empty placeholder and can then develop and edit each frame independently. (Testing,however, can only take place by going through the frames sequentially). This approach does allow a form of prototype to be generated and tested before full development of each frame takes place. However, the extraordinarily cumbersome compiling process, requiring an extra three data disks and approximately 10 to 15 minutes of disk swapping in a two-drive system, serves as a real and practical deterrent to any iterative refinement or evaluative efforts. Overall, therefore, we feel Trainer 4000 does not support the principles of modularity or of modifiability in a way which would be satisfactory to either a software engineer or an educator.

B.1. Intents of Instruction

Only two pages of the manuals of Trainer 4000 include specific comments on the value of making instructional decisions before using the authoring system. The documentation is entirely lacking in any sense of educational context or evaluation. No example programs are included and no recommendations made regarding specification of learning goals or characteristics of the intended users. While we realize that no authoring system can make these sorts of decisions for the developer, we still feel that some recognition of the primary importance of these considerations ought to be included and perhaps illustrated in the context of an example program.

B.2. Contents of Instruction

At the novice user's level, data cannot be stored independently of the program. This is a serious constraint, especially given the technical difficulties involved in editing and regenerating lessons. At the more advanced levels, it is possible to create independent question pools which can be accessed randomly or sequentially at specific points in the program. However, the same question pool does not appear to be reutilizable within the program, and

editing of the question pools apparently requires editing of the overall lesson itself.

B.3. Methods of Instructional Delivery

It is within *methodology* that particularly frustrating problems occurred in the Trainer 4000 environment. For example, the graphics editor provided a number of useful features for the construction of attractive visual displays, but did not allow the development of an interactive graphics screen. Thus, a student could look at an attractive visual (and animation was reasonably easy to include) but any input situation required clearing the graphic from the screen and replacing it with a text screen. This proved to be a serious pedagogical constraint for every one of our development projects, in that interactive involvement with the material displayed on a graphic screen was always desired. Having to ask questions with the graphics screen no longer available was wrong pedagogically in all of our instructional designs. In addition, Trainer 4000 apparently does not even allow the option of moving back in a program to examine a previous screen and then returning to the question frame. More generally, the inability to accommodate branching prevented implementation of most of our designs. We were unable to structure lessons so that the student could control his own movement; standard features in computer-based instruction, such as options menus and optional help screens did not seem to be feasible within the Trainer environment, at least as far as we could interpret from the documentation for the system.

B.4. Evaluation Facilities

Trainer 4000 was well-structured with regard to evaluating student responses to multiple-choice, true-false and matching questions, However, passing through the lengthy series of prompts each time a question was built or edited was time-consuming and often inefficient for the author. The administration system did seem to allow the educator to design a variety of data-capturing and display procedures for monitoring student activity and responses to quizzes. However, because of the many technical difficulties we had with the authoring system itself, we did not have the time to use the administration utility. One management aspect did appear to be built into the student presentation system.' A student returning to a lesson is automatically reentered at the place in which he or she exited. We found this unacceptable, as we believe both the student and the instructor should have the choice of reviewing previous material if desired. This pedagogical decision of restarting a student where he or she left off should be an option available to the educator or student, not a requirement of the system.

C. Collaboration Principles

Trainer 4000 did facilitate collaboration between educators and computer scientists involved in a software development project but the communication occurred as a reaction to problems rather than being motivated by a desire for constructive collaboration. It is our opinion that the technical demands of the system made it virtually unusable for independent educator-authors. This reflects a fundamental criticism of the Trainer 4000 system and also of many authoring environments in general -- who was this system designed for? Who is the intended user? If designed for a novice educator-author or member of a production team with limited time available for technical support, the system was totally inaccessible. If designed for a more experienced educator/author, the system did not appropriately provide adequate support for implementing many of the features that an individual might wish to include in his or her program design. If (and since) the package requires technical support for

its use, then the system was not described or organized in a way appropriate to the computer scientist who will become involved, in that the structures provided were not only poorly designed and documented, but also were inhibiting.

CONCLUSION

We feel an evaluation mode1 based on the software engineering and instructional design principles which underlie the collaborative model of software development provides a useful framework for the evaluation of a particular authoring system. By applying the evaluation criteria specified by our mode1 to our experiences with a particular authoring system, we can organize our criticism of the system within a theoretical framework rather than utilize an approach which would discuss the presence or absence of particular attributes which may be important in one program design but not in another. We feel the *fit* of any authoring environment within the broader design and development process is critical, as otherwise too much attention is given to a tool and too little to the overall system in which the tool is to be applied. Our evaluation criteria are sensitive to this larger system. We feel the design and development process, as well as any evaluation of this overall process or its component parts, can be enhanced by applying an orientation which is based on both software engineering and instructional design principles.

REFERENCES

Barker, P. G. (1986). *Author languages for CAL.* Unpublished report.

Barker, P. G., & Singh, R. (1983). A practical introduction to authoring for computer-assisted instruction, Part 2: PILOT. *British Journal of Educational Technology, 14* (3), 174-193.

Brahan, J. W., Farley, B., & Orchard, R. A. (1983). *A set of tools for courseware development.* Ottawa, ON: National Research Council, Associate Committee on Instructional Technology.

Burger, M. L. (1985, March). *Authoring languageslsystems comparisons.* Paper presented at the Fifth Annual Microcomputers in Education Conference, Arizona State University, Tempe, Arizona.

Collis, B., & Gore, M. (in press). Combining software engineering and instructional design principles in a new type of course for educators. *Association for Educational Data Systems Journal.*

Computer Systems Research, Inc. (1985). *CSR trainer 4000.* Avon, CT: Author.

Cyrs, T. E. (1985, March). *Selection criteria for CBI authoring systems.* Paper presented at the Fifth Annual Microcomputers in Education Conference, Arizona State University, Tempe, Arizona.

Gore, M., & Collis, B. (1986, May). *A collaborative model involving computer scientists and educators for the development of educational software.* Paper presented at the Fifth Canadian Symposium on Instructional Technology, Ottawa, Ontario.

S. (1986, May). *CAL authoring system.* Paper presented at the Fifth Canadian Symposium on Instructional Technology, Ottawa, Ontario.

Jonassen, D. H. (1985). The REAL case for using authoring systems to develop courseware. *Educational Technology, 25* (2), 39-41.

Merrill, M. D. (1985). Where is the author in authoring systems? *Journal of Computer-based Instruction,* 12 (4), 90-96.

Quick, G. E. (1985). ESTOP: Software tools for computer-based education. *Research in Education,* 33,49-56.

Szabo, M., Jensen, W. A., & Bagnall, K. (1986, May). *Productivity in courseware development: Validation of an authoring system.* Paper presented at the Fifth Canadian Symposium on Instructional Technology, Ottawa, Ontario.

Wilson, R. N., & McCrum, E. (1984). Use of modular design in the production of portable CAL software: A case study. *Computer Education,* 8 (2), 229-237.