SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 16(2), 131–143 (FEBRUARY 1986)

SOFTLIB — A Documentation Management System

I. SOMMERVILLE, R. WELLAND, I. BENNETT AND R. THOMSON
Department of Computer Science, University of Strathclyde, Glasgow, Scotland

SUMMARY

This paper describes a software sysem (SOFTLIB) that has been developed to assist in the management of software documentation generated during systems development projects. It provides facilities to manage large numbers of documents, to file documents when they are complete and to issue them to system developers and maintainers. It also includes an information retrieval facility that allows programming staff to find documents, to examine their contents before issue and to assess the state of the software project documentation. SOFTLIB is explicitly intended to help manage the documentation generated during software development — it is not designed for use by end-users of that software or for managing end-user documentation.

The novel characteristic of this system is the approach that is taken to the consistency and completeness of documentation. The documentation associated with a software system is organized in such a way that it may be detected if document sets are complete (that is, if all documentation which should be provided for a software component is available) and if document sets are likely to be inconsistent. This means that if a document has been changed without a comparable change being made to other associated documents, this is detectable by the librarian system.

In addition, a subsidiary aim of our work was to investigate the utility of menu systems to complex software tools by building a user interface to SOFTLIB. We conclude that menu systems are far from ideal in such situations because of the range of possible options which must be handled by the system.

KEY WORDS Software documentation Software re-use Project management tools Software completeness and consistency

INTRODUCTION

A common characteristic of all large software systems is the prodigious amount of documentation which is generated during software development. Until relatively recently, this documentation was written by software engineers, typed on paper and managed by a project secretary who was allocated the job of documentation librarian. Now, in more and more instances, documentation is written and stored on the computer that is used for software development. This opens up the possibility of using an automated tool to manage the issuing, updating and filing of that documentation, thus relieving a project secretary of an uninteresting, time-consuming and error-prone task.

The notion of an automated management system to support some of the activities of software development is not new. Systems such as CADES,[1] which is a database-centred support system, have been used in the development of a number of large software

0038–0644/86/020131–13$05.30

Received 6 November 1984
Revised 13 May 1985

projects. Several other systems, which are mostly proprietary developments, have also been used and a good deal of research work in this area has been reported.[2] Although systems such as ARGUS[3] provide facilities for creating documentation, the management of documentation as a separate entity seems to have been neglected.

Current thinking in this area is for the provision of a completely integrated project support environments (IPSEs) where automated tool support is provided for all stages of software development. These are based around a project database and include a database management system to control all project information. Such environments are exemplified by the Ada Programming Support Environment.[4] However, such systems are not easy to develop — the last few years have seen the failure of the MCHAPSE project[5] in the U.K. and Intermetrics' Ada Integrated Environment (AIE) project in the United States.

It is unlikely that integrated project support environments will become widely used until the 1990s. We believe that, until then, most large-scale software development will take place using software toolkits rather than integrated environments. An example of a software toolkit is the UNIX† Programmers Workbench System[6] and UNIX-based toolkits seem likely to be the most commonly used software development systems. Software toolkits are distinct from integrated environments in that they do not normally make use of a database management system but rely on the host computer's filing system for storing project information. Thus, the user does not have access to the powerful data manipulation facilities offered by a database management system.

The aim of our work was to develop a component of such a software toolkit, namely a system to manage the documentation associated with a large software project. As well as providing documentation management services, it was our intention that the system should include checking facilities so that a restricted form of documentation consistency and completeness checking might be carried out.

The model on which our system is based is that of a shared 'public' document library with each system user having his or her own private workspace where no restrictions are imposed by the documentation management system. Currently, this is provided using a multi-user UNIX system but we envisage that future development environments might be made up of workstations linked to a central computer hosting the document management system.

The SOFTLIB system uses a document classification scheme which collects together all of the documents which are associated with a particular software component. This component document set is transferred, as a unit, to and from the user's workspace. SOFTLIB also allows access permissions to document sets to be established, document sets to be edited and new SOFTLIB users to be added to the library 'register'. A key feature of the system is a form of completeness and consistency checking, carried out when documents are returned to the library. If a document set is incomplete or inconsistent the user is warned, and certain restrictions are placed on future modifications of these documents.

SOFTLIB is a prototype system whose main purpose is to demonstrate the feasibility of limited completeness and consistency checking and to investigate the requirements for a project information system. In principle, the system can support any number of projects and users but its main function was to demonstrate the feasibility of ideas. It has thus only been used by a small research group on a limited number of projects. The system has been built in a language called S-algol[7] and uses a small document database

†UNIX is a trademark of AT & T, Bell Laboratories.

built on top of the UNIX filestore. The system runs on DEC PDP-11 and VAX computers.

DOCUMENT CLASSIFICATION

The basis of any library system is an effective method of classification. We looked at the organization of some existing software document libraries which were organized into sublibraries. Each sublibrary was associated with a particular software development project and, within the sublibrary, documents were classified as specification documents, design documents, implementation documents, etc. Document dependencies had to be maintained manually and there was no automatic way of ensuring that changes to one document were reflected in all other associated documents.

A classification scheme based on document type (specification, design, etc.) suffers from a number of disadvantages:

1. Documentation associated with a software component is not naturally related. To find all documents associated with a subsystem, the user may have to search the entire project sublibrary.
2. The re-use of software components and associated documentation is not encouraged. Software components with similar functions are distributed across many projects. All project sublibraries must be searched to find a set of components with similar specifications to determine which is the most appropriate for a particular job.
3. Documentation completeness and consistency is difficult to check. It is often difficult to find out if all documentation which should be produced for a software component is in existence and if it is in a consistent state.

Although we retained the notion that it is useful to classify documentation by project, we rejected the idea of classifying documents as design documents, implementation documents, etc. Instead, the basis of our classification scheme is the *software component*.

We consider a software component to be any item of software which has an associated specification. Thus packages or modules implementing abstract data types, subsystems, procedures, functions, etc. are all possible software components. Each software component has associated documents such as a specification, one or more designs, one or more implementations, a set of test data, etc.

Software components may be gathered into *component families* where all the components in a family have some commonality in their specification. Thus there might be a family of sort components, a family of display components, a family of syntax analysis components and so on. This classification into families is intended to assist in finding software for re-use.

To find a component of a particular type (a search component say), the user makes use of the library information system to find out what components are available in the 'search family'. Although these components may have been developed as part of different projects, SOFTLIB cross-references them under both project and component family.

For any component specification, there are usually many possible implementations of that specification. These might be implementations in different programming languages, implementations designed for different computers, implementations which use different algorithms to optimize space, speed, etc. Each of these is termed a *version* of the component. We deliberately restrict the term 'version' to cover only those implementations of a component which have a common abstract specification. Thus, all of the

versions of a search component might search an array of integers but one version might be a linear search written in C, another might be a linear search in Pascal and a third version might search by sorting then binary chopping.

Each version of a component has its own *document set* made up of design documents, programming language code, etc. This version document set is distinct from the component document set which is made up of implementation-independent documents such as specifications and test data.

The classification scheme used in the SOFTLIB system is illustrated in Figure 1.

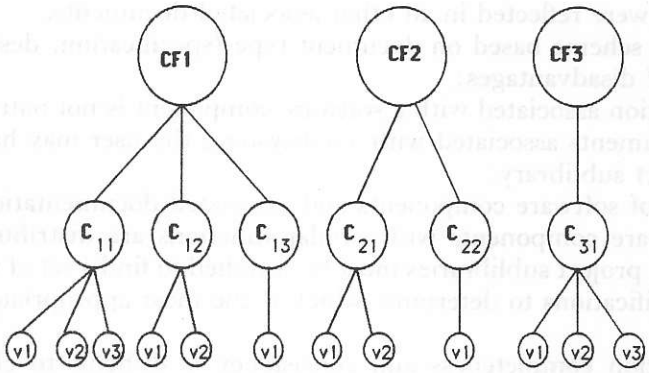

Figure 1. Component classification

In Figure 1, we see that component family 1 (CF_1) is made up of three components which have similar (but not identical) specifications. For example, if CF_1 is a family of search components, C_{11} might be an integer search, C_{12} might be a file search and C_{13} might be a component to search an array of strings. Both C_{11} and C_{12} have several versions but only a single version of C_{13} is provided.

CF_2 and CF_3 are each component families where there are two components in CF_2 but only a single family member in CF_3. In our present system, we insist that component families are disjoint so that it is not possible for a component to be a member of more than one family.

This classification scheme avoids the disadvantages of schemes which organize documentation by type. Documentation associated with a component is accessible as a single unit. Component re-use is simplified by classifying components as family members and by providing facilities to examine all members of a component family. Documentation checking is possible, as discussed in the following section.

On top of this classification scheme, we maintain an additional classification by project. The system keeps track of which component versions are used in which projects. This is illustrated in Figure 2.

Figure 2 shows how projects make use of individual components from different families. Notice that it is possible for different versions of the same component to be used in the same project.

DOCUMENT CONSISTENCY AND COMPLETENESS

Each version of a component in a software system usually exists in a number of different

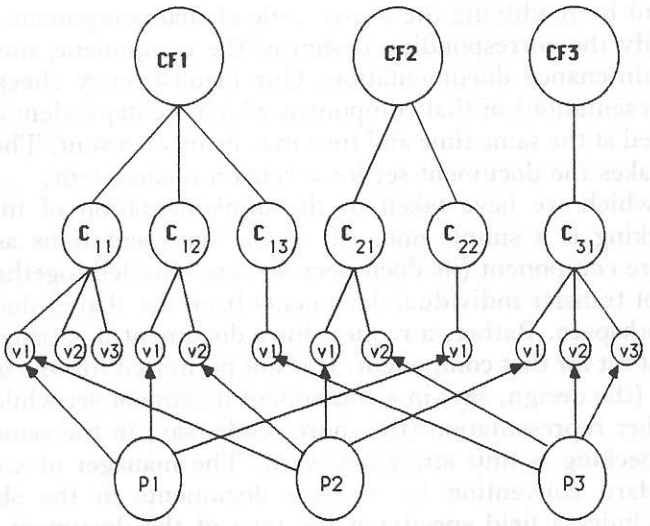

Figure 2. Project use of component implementations

representations. There may be one or more diagrammatic representations of the component design, a design description in some program description language and an implementation in a high or low-level programming language. A common practice, particularly as the project delivery date draws near, is to modify the implementation under time pressure and to delay corresponding modifications to the other representations until later.

Often, later never comes! The different representations of the version of the software component are left in an inconsistent state. They may remain like this for some time after the system has been delivered and may only be discovered when the software must be modified during system maintenance or enhancement. As it is very difficult to retrofit implementation changes to a design, it may be impossible to make the representations consistent without complete rewriting.

Different projects require different representations of a component to be provided. A component representation is said to be *complete* when all representations required by management have been produced, entered in the component document set and placed under the control of the librarian system. A definition of consistency is rather less straightforward, so we operate using the notion of inconsistency. It is not possible to determine, using software tools, if a representation set is logically consistent. That is, we cannot write a formal component specification and check automatically that all manifestations of a component match this specification. Thus, our approach is necessarily a weak one; nevertheless, we believe that it is useful.

However, we can detect the possibility of *inconsistency* between representations by keeping track of modifications to the various component representations. When one representation is modified, all other dependent representations must be changed during the same editing session. This mechanism does not catch logical inconsistencies where all representations are modified but the modifications are not logically consistent. However, it does trap inconsistencies which are a result of changing one representation and omitting to change other corresponding component representations.

For example, say a user is working under pressure and creates a new version of a

software component by modifying the source code of that component. However, he or she does not modify the corresponding design of the component, nor does he or she modify system maintenance documentation. Our inconsistency checking system will detect that all representations of that component which are dependent on the code have not all been modified at the same time and thus may be inconsistent. The 'quick fix' to an implementation makes the document set for a version inconsistent.

The approach which we have taken to the implementation of inconsistency and completeness checking is a simple one. All of the representations associated with a version of a software component (its document set) are bundled together under a single name. Users do not transfer individual documents from the shared document database to their private workspace. Rather, a request for a document is satisfied by issuing the complete document set for that component. It is not permitted for one user to work with one representation (the design, say) in a component document set while another user is manipulating another representation (the source code, say) in the same document set.

Completeness checking is thus straightforward. The manager of a software project establishes a standard convention for naming documents in the shared document database which includes a field specifying the type of the document. In the current implementation, we follow the UNIX convention of using suffixes to denote different classes of document. For example, documents which are dataflow diagrams may always have a suffix .dfd, documents which are maintenance descriptions might be suffixed with .mnt and source code might be suffixed .src. The SOFTLIB system enforces this convention and insists that all documents which are entered in the project library have a standard name. Completeness checking is accomplished by ensuring that all documents which should exist for a version of a project component have been entered in the project library.

Completeness checking does not require any global searching nor need the user explicitly link representations. Checking the completeness of any one component simply involves a single directory access to ensure that all representations are provided in that directory.

Inconsistency checking is also straightfoward. We have put a cladding around the system editors so that, on completion of an editing session, a menu of possible types of change is presented to the user. Possible change types are cosmetic changes (formatting, say), code changes (where the fundamental design of a component is unchanged but the code is modified), design changes, interface changes and so on. The onus is placed on the user to record the change type because different types of change require that different documents in the document set be changed correspondingly.

We adopted this approach rather than a date-stamping approach because some changes, such as cosmetic changes, do not make a document set inconsistent and other changes, such as code changes, require the change to be reflected in some but not all dependent documents. Although the user may evade the system checking by marking all changes as cosmetic, we assume that he or she will co-operate in change recording.

After an editing session, the user returns the entire document set to the document library. At this stage, the system checks for possible inconsistency and, if an inconsistency is detected, it flags the document set as potentially inconsistent. When this flag is set, further operations on the document set are restricted.

If a component is returned to the library in an inconsistent state, its name is recorded in an 'inconsistent components' list and that component may only be reissued to the individual who caused the inconsistency, to his or her project manager, or to the owner

of the component. This means that an inconsistent component may not be issued for further unrelated modification. This restriction is necessary in order to keep track of who made the component inconsistent and who must remove the detected inconsistency.

We have adopted this approach rather than forbidding the inclusion of inconsistent document sets in the library because the realities of software development mean that it is sometimes necessary to make a 'quick fix' to a component. We believe that it is unreasonable to expect all component document sets to be consistent at all times and our approach means that inconsistent components may be detected and may not be independently modified until they are in a consistent state.

DATABASE SECURITY

A document classification scheme which is based on projects and software components leads naturally to an effective document security scheme. This is particularly important where the system is built on top of the UNIX filestore. The UNIX system does not support, in an adequate way, controlled sharing of specific files. Although group permissions on a file are allowed, it is not possible for different group members to have different access permissions on a file or a directory. Furthermore, overlapping of group permissions is only possible on some versions of UNIX and is inconvenient on many of them.

Security is enforced at a number of different levels, such as:

1. *The project level.* It is assumed that all system users are working on a specific development project. Thus, to use the librarian system they must log on using a project identifier and password. They are then allowed access to the components associated with that project.
2. *The component level.* For each component in the system, the librarian system maintains a user/use list. This list holds the identifiers of development staff who are allowed to access that component and whether they have read, copy or read/write access. Component level permission allows access to component documents, such as specifications, which are version independent.
3. *The version level.* Each component version also has a user/use list defining allowed access to that version. Permission to access a version implies permission to access all documents associated with that version. Component level permission must be held before access to a version is permitted.

As well as controlling access to document sets, the librarian system also maintains a transaction log. This keeps track of all transactions and holds the name of the user, component, implementation and accessed document. Thus, if unauthorized access to a document is gained, this is detectable using this log.

SOFTLIB FACILITIES

The facilities which are offered by the librarian system fall into two main classes. These are, first, facilities for documentation management and, second, information retrieval facilities. The retrieval facilities are analogous to a library catalogue, which allows SOFTLIB users to browse in the documentation library. In our current system, the SOFTLIB system is implemented as a single stand-alone program with the SOFTLIB command set accessible after the program has been initiated.

Document management facilities

The document management facilities provided by the librarian system allow documents to be filed in and retrieved from the document database, allow project and access permissions to be entered and allow documents to be edited with that edit recorded by the system.

Commands available in SOFTLIB fall into three categories:

(i) commands for logging on and logging off
(ii) commands for moving and editing document sets
(iii) commands for project management and for project administration.

The UNIX user calls SOFTLIB in the usual fashion by typing its name (softlib) and he or she is then asked to log on to the SOFTLIB system. The log-on name is a project identifier. A correct response to the log-on request results in SOFTLIB requesting a password from the user. On completion of a SOFTLIB session, the user returns to UNIX using a logoff command. We adopted this log-on/log-off procedure because it provided a security mechanism and because it is a convenient way of associating programmers with a particular set of project documentation.

When SOFTLIB is used for the issue and editing of documents, the normal mode of working is as follows:

1. The user types a get command which instructs SOFTLIB to fetch a document set from the library and transfer it to the user's workspace. If the user is permitted access to the document set, it is then locked so that it may not be reissued and is transferred to the user's personal workspace. Locked component documentation may still be read using the information retrieval facilities.
2. To edit the component documents, the user issues an edit command which, as described above, initiates the standard system editor and, on completion of the edit, requests modification information from the user. It also date stamps the component document set and makes an entry in a transaction log which holds information about all document modifications.
3. After editing is complete, the user issues a put command to return the document set to the document database. The put command initiates the consistency checking mechanism discussed above and, if the document set is consistent, it is unlocked for subsequent issue.

The SOFTLIB system assumes that software projects are administered by a project manager and provides a number of commands which allow users to be added to the system, permissions on document sets to be changed and so on. Examples of this class of command are the newuser command, which allows new programmers to be added to a project or for user permissions to be modified, the acc command which is also used to change permissions, but at the version level rather than the component level, and the create command which is used to name a new project and set up or change document set requirements for completeness.

SOFTLIB command examples

To illustrate the use of SOFTLIB, the dialogue below is an example of a possible terminal session with SOFTLIB. Output generated by the system is shown in upper case, user input in lower case and a comment is on a line by itself which starts with the character *. The SOFTLIB prompt is >.

```
* initiate the SOFTLIB system
softlib
ENTER PROJECT NAME: compiler
PASSWORD:
* SOFTLIB now displays a command menu — an abbreviated version of this
* is as follows:
     1. GET
     2. PUT
     3. EDIT
     4. NEWUSER
     5. CREATE
     ...
* User selects get from command menu
>1
* SOFTLIB asks which class of document is required
DO YOU WISH TO RETRIEVE
     1. COMPONENT SPECIFICATION
     2. IMPLEMENTATION DOCUMENT
* user selects implementation document, i.e. a version
>2
* system displays available versions
DO YOU WISH TO RETRIEVE
     1. LEX.ANALYSER.PAS
     2. LEX.ANALYSER.LEX
* user selects lex.anlayser.pas
>1
OK
*Command menu displayed, user selects edit
>3
DOCUMENT NAME > lex_analyser.pd1
* normal unix edit session to change document
* system asks user to indicate class of change
* cosmetic changes imply that no other documents need be changed, code
* changes imply that code and detailed design documents must be changed
* design changes imply that all documents must be changed
INDICATE CHANGE TYPE
     1. COSMETIC
     2. CODE
     3. DESIGN
>2
* Command menu displayed, user selects put
>2
WARNING: POTENTIAL INCONSISTENCY — CONTINUE?
>y
OK — LEX.ANALYSER.PAS MARKED INCONSISTENT
* user now acts in the role of system manager and adds a new user to the
* project — selects newuser from command menu (not shown)
>4
```

```
    1. ENTER NEW USER
    2. CHANGE PERMISSIONS
>1
ENTER NAME> john smith
ENTER DIRECTORY> /usr/projects/compiler/js
OK
>logoff
```

This hypothetical terminal session shows how the system makes extensive use of menus to initiate commands. This menu interface is discussed in more detail in a later section of this paper.

INFORMATION RETRIEVAL

The information retrieval facilities of SOFTLIB are implemented as a separate subsystem which is initiated by issuing a piqs command where piqs is shorthand for 'project information query system'. The object of providing information retrieval facilities was to provide the automated analogue of a library catalogue. However, rather than provide an electronic card index, we have built a more powerful system which allows documents to be retrieved and which allows the user to find out what documents are available in the system. In addition, the user may also query the state of project development and document sets, discover what transactions have taken place and find out what components of a particular class are available.

A decision which was made at an early stage in this project was that the information retrieval facilities should be 'read-only'. Using these facilities, the user may look at documents, but to change any document the documentation management facilities must be used. This allows unfettered use of the information retrieval system to be made without the possibility of inconsistency being introduced into any document sets.

The information retrieval system (PIQS) offers access to information at a number of levels:

1. *The component family level.* The user may find out what components in a particular family are available, what the specifications of these components are, who the owner is, etc.
2. *The component level.* The user of PIQS can find out what component documents are available, who produced them, whether the component document set is inconsistent, etc.
3. *The version level.* For any component version, the user may query the availability of documents in the document set and may retrieve any document in that set, provided, of course, that he or she has access permission.
4. *The project level.* For all projects, information about the state of the project, the components used, the project members, etc. may be retrieved.
5. *The transaction level.* The system maintains a transaction log which records details of who has done what. This may be queried via PIQS. Thus details of who was responsible for changes to a software component may be recalled by project management.

The initial menu presented to the user is

RETRIEVE MENU
1. Management information
2. Project information
3. Component family information
4. Component information
5. Version information
6. Document set information

The user selects the documentation level of interest to him. Suppose that number 2, project information, is selected. The user is then presented with the following menu:

DOCUMENTATION MENU
1. List all projects
2. View a specific project
3. Project responsibility
4. Project state
5. List all using component
6. List all associated components
7. List all inconsistent components
8. Complex enquiry

Numbers 5–7 above require additional input from the user. Option 8 is an escape mechanism to allow access to a query language, as we found it very difficult indeed to express complex queries using a menu-based system. The problems which we encountered with menu systems are discussed in the following section.

PIQS is aware of the document classification scheme which is used by the librarian and provides simple commands to navigate through the document organization. Thus when working at menu level N (say), the user can issue a command d to go down a level and a command u to go up a level. Thus, if working with components, d gives the version menu and u gives the component family menu. Each level has its own command menu. So, if working at the component level and if d is typed, the 'version menu' (shown below) is displayed. This displays enquiries which may be made on all of the versions of a component.

1. Select a component version
2. Find out about version owners
3. List all associated documents
4. Complex enquiry

The interactive session proceeds with the user moving up and down menus as necessary to retrieve the required information.

THE USER INTERFACE

The main aim of this project was to provide a documentation librarian, but we also had

an important subsidiary aim. We believe that personal workstations equipped with high-resolution displays and 'mice' will become the normal working tool for software engineers. An attractive user interface on such systems is menu-driven with the user pointing at his or her menu choice. We therefore decided to investigate the utility of menu-driven interfaces to a system which allowed fairly complex operations.

We make extensive use of menus in both the documentation management facilities and in the information retrieval system. We found that for the types of operation supported by the documentation management system, menus were a convenient form of interaction. Given a better method of menu selection (we were constrained to work with a character terminal), we believe that our interface would be easier and faster to use than a command-driven interface.

However, we found that a menu-driven approach was less suitable for the information retrieval system. The problems which we encountered with this menu-based approach came about because we wished to allow the user to input complex conjuctive queries. Examples of such queries are:

(a) List all components written in Pascal which are part of project P and which are owned by Y.

(b) List all versions of component X created after data DDMMYY

In our original implementation, we implemented such queries using a menu hierarchy, where a new menu was introduced for each parameter of the request. This was inconvenient in that the user had to wait for menu display and in that the user got lost in a menu maze and had difficulty in correcting errors or, sometimes, navigating his or her way out of the command.

A deep hierarchy of menus was necessary and the user, in spite of being familiar with the system, readily lost track of where he or she was in this hierarchy. Ultimately, we abandoned the notion of developing an interface which was purely menu-driven and developed an interface which was partly menu-driven and partly based on a simple command language.

Therefore, each menu in the query system has an escape option which initiates a simple command language interpreter which operates at that menu level. Thus the above queries might be posed as follows:

```
list components, project=P, language=Pascal, owner=Y
list versions, component=X, cdate > DDMMYY
```

Although some of the problems which we had with menu systems would disappear if we had more suitable hardware, our general conclusion was that a command language was more suitable for posing complex queries.

CONCLUSIONS

The SOFTLIB system seems to be a useful aid to documentation management particularly when the components and their associated documentation have clearly defined functions. The idea of working with document sets rather than single documents helps to detect document inconsistency and is an effective way of ensuring that all required documents are provided. Although our notions of consistency and completeness are weak, they have the advantage that they may be easily understood and readily implemented. Comments from software management in industrial software develop-

ment organizations have suggested that documentation inconsistency is one of their most serious problems. Our system goes some way towards a solution to this problem.

Our other important conclusion is that menu systems are not really suitable for the complex tasks undertaken during software development. Although they allow fast and effective access to systems which have few options, systems which allow complex combinations of options are best supported using a query language. Our work has highlighted a need for much more research in the general area of the user interface to complex software development tools, where tool users are experienced rather than naïve.

Our system works best for components which have a clearly defined function and for component documentation which concerns a single component. It was not intended to handle user documentation (say) where the system as a whole rather than as a set of individual components is described. However, the problems of consistency and completeness apply equally to that type of documentation. There is a need for a notation to specify interdependencies between parts of such documentation and component documentation sets so that changes to one section may be reflected in other related documentation.

REFERENCES

1. R. W. McGuffin, A. E. Elliston, B. R. Tranter and P. N. Westmacott, 'CADES — software engineering in practice', *Proc. 4th Int. Conf. on Software Engineering,* Munich (1979).
2. H. Hunke (ed.), *Software Engineering Environments*, North-Holland, Amsterdam, 1981.
3. L. G. Stucki and H. D. Walker, 'Concepts and prototypes of ARGUS', in Reference 2.
4. *Requirements for Ada Programming Support Environments: Stoneman*, U.S. Department of Defense, 1980.
5. J. G. P. Barnes, 'The UK M-Chapse', *Ada UK News,* **4**, (1983).
6. T. A. Dolotta, R. C. Haight and J. R. Mashey, 'The programmer's workbench', *Bell Syst. Tech. J.*, **57**, (6), 2177–2200 (1978).
7. A. J. Cole and R. Morrison, *An Introduction to Programming with S-algol,* Cambridge University Press, Cambridge, 1982.