# Dynamic (Re)Generation of Software Documentation

W. Lewis Johnson
USC / Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292-6695
johnson@isi.edu

**Abstract**

We are developing an authoring tool called I-Doc that will automate the process of generating documentation and user help for software systems. The focus of the tool is on capture of the requirements and design decisions that form the content of software documentation. This information can then be used to generate summaries and explanations of the software on demand. The objective of this research is to provide on-line assistance for software maintainers and other software professionals that can take the place of conventional bulk documents. I-Doc is designed to support the reengineering of software systems, since some of the necessary design information will have to be captured by annotating existing code. Reengineering technology, specifically transformation technology, is employed during the generation process to simplify and reorganize design information when describing software.

## 1 Introduction

Conventional documentation for software systems has surprisingly little value, given the amount of time and effort spent to create it. This is particularly true for large, mature systems. Such systems typically have voluminous design documents, in which it is difficult to find information relevant to any specific maintenance task. If the documentation is not maintained in lock step with the code, it quickly becomes inaccurate, so maintainers cannot rely upon it. There is an increasing need for alternative technologies that can provide maintainers and users of software systems with the information that they need to operate and maintain those systems.

We are developing a documentation authoring tool that will automate the process of generating documentation and user help for software systems. This tool will result in dramatic improvements in the way documentation is developed, maintained, and used. If the design, requirements, and assumptions underlying code are made explicit, generation of documentation can be substantially automated. This underlying knowledge can be acquired and formalized in a natural, incremental fashion that does not overly burden developers.

Documentation will be generated dynamically, in response to specific requests for user information. When the user requests information, the documentation system determines what information content should be presented. It composes a response by combining textual descriptions previously entered in a database, and automatically generating natural language output to fill in the rest. The content of the generated output depends upon the level of expertise of the user, and the history of previous user documentation requests. This fundamentally changes the nature and role of documentation. There will be less need for users to search through bulk documents in order to obtain answers to specific questions. Instead, the documentation system will search its own knowledge base for the information that the user requires, and compose explanations meeting the user's needs.

Other CASE (computer-aided software engineering) tools have been developed to support the authoring of documentation. These tools differ in that they tend to be oriented toward the generation of specific reports, such as those mandated by Department of Defense procurement standards. They make it easier to produce such reports, but that does not make the reports themselves significantly more useful. The I-Doc approach is designed to make such reports unnecessary for most purposes, although it will still be possible to generate bulk reports from I-Doc's design repository.

The approach employs reengineering technology,

and is designed to be compatible with reengineering efforts. Information required to support documentation is entered in a reengineering knowledge base, in the form of annotations on source code parse trees. Transformations are employed to generate simplified and reorganized sections of code that highlight the aspects of the system being documented. Design recovery activities have the effect of bringing documentation up to date, facilitating subsequent software maintenance.

## 2 The State of Current Documentation

Let us examine the problems associated with conventional software documentation, to see how new techniques can alleviate those problems.

The primary emphasis of conventional system documentation is on amassing information. Documentation standards, especially government standards such as MIL-STD-2167A or SDD, require developers systematically to describe all details of a design, such as the inputs and outputs of each function. The structure of such documents is fixed and standardized.

The first problem with such system documentation is that is not sufficiently *activity-oriented*, i.e., it is not designed to support the activities of the intended readership. User manuals are activity-oriented in this sense: such manuals are designed to help people whose activity is to use the software system. System documentation is not, or if it is the set of activities being supported is incomplete. System documentation can potentialy support a number of activities, including the following:

- review by the customer to check that all stated requirements are met in the design,
- design reviews in which the quality and validity of the design is evaluated, and
- maintenance activities, in which maintainers seek to obtain information about the design so that modifications and enhancements can be performed correctly.

These activities are very different, yet documents often must support more than one of them. The activity that is least supported, of course, is maintenance. Maintenance manuals are common for physical devices, but are rare for software systems. Of course it is harder to write maintenance manuals for software than it is for devices, because maintenance tasks change as the software evolves. Nevertheless, maintenance activities in general are vastly different from specification and design reviews. Maintainers rarely perform methodical reviews of entire systems; rather, they inspect specific modules in detail in order to determine how they can be modified. Interrelationships between modules can be extremely important. Documents such as design documents, which describe all components in a uniform way, are more suited to design activities than maintenance activities.

In addition to being activity-oriented, good documentation is *task-oriented*, i.e., designed to help readers perform specific tasks. Tutorial user manuals are frequently written in a task-oriented fashion. For example, a word processor manual might have the user work through sample tasks such as composing a business letter or printing mailing labels.

A task-oriented approach centered on hypothetical tasks is not necessarily the best way to design documentation in general. It requires the reader to take the time to work through exercises, whereas document users typically are impatient and skip through the documentation trying to find out what they need so they can get on with their actual job. This is the motivation for the new "minimalist" approach to documentation, which uses overviews, structured exercises, and any information that the user cannot discover through experimentation with the system [3]. However, the minimal approach is not a rejection of task orientation per se, just of manuals that are oriented around lengthy hypothetical exercises and that contain information one can figure out on one's own.

It is certainly true that system documentation can be improved simply by learning lessons from other types of documentation such as user documentation. However, even well-written paper documents suffer from basic limitations. A person writing a document can only make rough guesses about what tasks the reader might be performing, what information he or she might want to know, and the level of expertise of the reader. Detailed exercises can help eliminate the guesswork—if the reader works through an exercise, the writer can try to anticipate what kinds of questions the reader might want to ask at each point in the exercise. This does not work if readers lack the patience to work through the exercises, as the minimalists argue.

The key to a substantial improvement in documentation is an on-line system that can construct presentations dynamically, reducing the reliance on guesswork. Interaction with the system should be in the form of a question-answer dialog; that way, the reader

indicates to the system what he or she wants to know. The system can present information in the context of the reader's activities, simply by asking the reader questions about those activities. If the reader does not understand the descriptions generated by the system, the reader should be able to request a clarification, and have the system adjust its estimate of the reader's level of expertise. The process of supporting documentation then becomes less an activity of writing text and more an activity of providing the system with the information that it needs to produce a range of descriptions of the system.

Such a capability constitutes a clear advance of the state of the art in documentation support. However, the technologies needed to realize such a capability, such as design repositories, hypertext, program analysis and transformation, and natural language generation, are well developed and in a state where they can be brought to bear effectively on the documentation problem.

## 3 An Example from the Reader's Standpoint

The following example illustrates how I-Doc is intended to function, from the viewpoint of the reader, i.e., the person asking questions about the system.

The system in question is real-time embedded control software of a fighter aircraft radar system. This example was studied by Hughes in a research effort sponsored by Wright Patterson Air Force Base [4]. Hughes built a demonstration hypertext documentation system to support a hypothetical maintenance task on this system. We have been using the same example as an initial test case, to design I-Doc so that it can generate descriptions automatically that are similar to what the Hughes group constructed manually in their demonstration.

One of the functions of the radar system software is a Range While Search function, which electronically controls how the aircraft's radar scans the airspace. Normal Range While Search scans a volume of air space that is wider in azimuth than it is in elevation, say 60 degrees to the left and right of the aircraft, and 10 degrees above and below the horizon. The volume is scanned by sweeping back and forth horizontally, top to bottom. The hypothetical maintenance task is to change the code so that it can scan volumes that are wider in elevation than in azimuth, by scanning vertically rather than horizontally.

Figure 1 shows the window that the user interacts

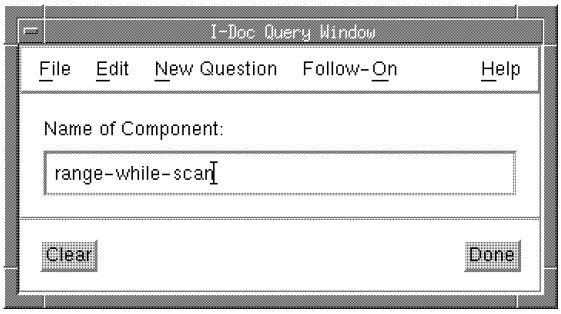


Figure 1: I-Doc Query Window

with in order to initiate the query. The user types in the name of the component that he or she is interested in. Mechanisms for selecting components from a menu of alternatives will also be provided.

Before I-Doc can accept a query, however, it first requests information about the user and the task being performed. The user parameters are input via a menu such as that shown in Figure 2. The user is requested to indicate what role the user plays on the project, and indicates Maintainer. I-Doc will therefore include in the system descriptions that it generates information relevant to maintenance, e.g., inputs, outputs, and functional decomposition of each module. If the user had chosen a different selection, such as User, descriptions would be more functional in nature, and limited to those aspects of functionality that would be visible to the user (in this case the pilot or radar intercept officer responsible for controlling and monitoring the radar).

Additionally I-Doc requires an initial estimate of the user's degree of familiarity with the system. In this case the user selects Low, which causes I-Doc to limit the extent to which it refers to implementation details such as data representations.

Next, I-Doc requests a characterization of the task the user is performing. Four types of activities are known relating to system maintenance: adding functionality, fixing bugs, optimizing, and validating documentation. Add Functionality is the choice in this case. In this context, it causes I-Doc to generate high-level overviews of the functionality in question. If Fix Bug or Optimize were chosen, the description would focus more narrowly on those system components involved in generating the behavior that must be optimized. Validate Documentation is chosen when the user (typically a developer) wishes to see a variety of descriptions generated by I-Doc, to verify that the system can generate valid documentation in each case.


Figure 2: User Parameter Window

Figure 3 shows a sample output, based upon the parameters selected above. I-Doc cannot yet generate this output, as the project is just getting started; this is merely an illustration of the type of output that will be generated. The figure is a display generated by the Mosaic hypertext system [2], which is the hypertext system used as an output interface by I-Doc. The figure contains a simplified decomposition diagram showing the major components of Range-While-Scan: Scan Generation and Output Processing. It summarizes the function of each component, and the main inputs and outputs of each.

Several points are illustrated by this output example. First, the output is selective both in terms of what components of Range-While-Scan are described, and what properties of those components are mentioned. In this overview the major components of Range-While-Scan are shown, but not those components responsible for checking and reporting errors. (For example, if Output Processing detects erroneous radar input, an error is signaled.) It characterizes the function of the components (e.g, Scan-Generation creates a scan pattern), and the inputs and outputs of each component. If the task or user parameters were different, the summary would have changed accordingly, perhaps including more detailed information about the system.

Because the presentation medium is hypertext, it is not necessary to enumerate all relevant properties of Range-While-Scan. It is sufficient to provide hypertext links which, if selected, will permit the reader to obtain further information. Some of these buttonable items are interspersed through the text, and appear underlined in the figure. Other items appear at the bottom. Because Range-While-Scan's performance requirements are particularly important for anyone attempting to add functionality to it, a special hypertext link is included to access this information. Other relevant topics are included at the bottom. The links named "electronically controlled radars" and "radar data processing" provide background about the application domain that might be useful to a maintainer who is unfamiliar with the application. Below are listed links for obtaining more information about Range-While-Scan's components. Further down, below the bottom of the scrolling window in this example, are pointers that allow the reader to see the source code from which this description is derived, either the full text or a simplified version corresponding to what appears in this hypertext description.

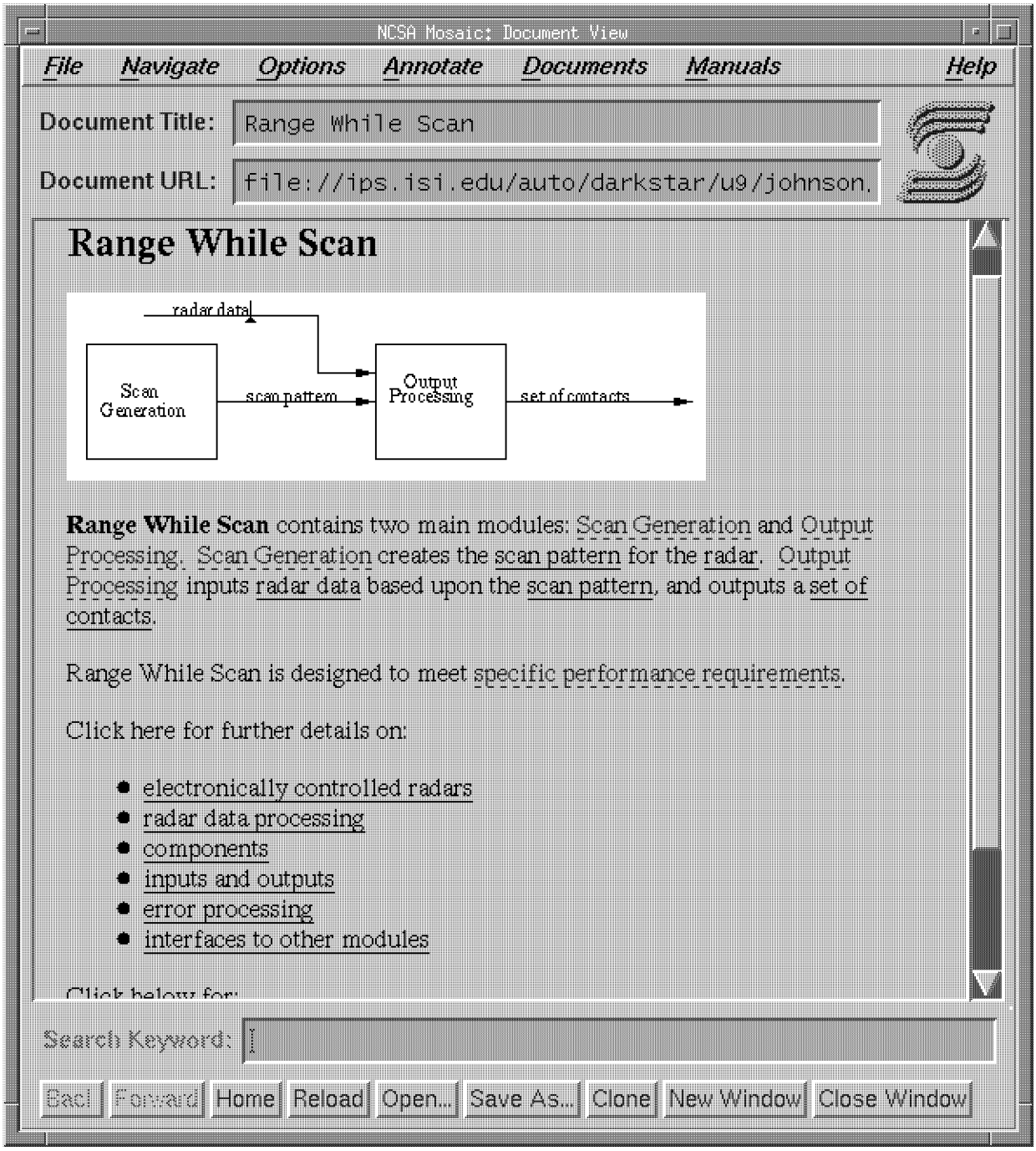


Figure 3: Hypertext Description of Range-While-Scan

## 4 System Architecture

I-Doc contains the following major functions.

- An acquisition interface is used to input the annotations necessary to generate system descriptions.
- This information is stored in a repository, and in annotations embedded within the source code itself.
- A query interface, as shown in Figure 1, is used to input queries from the user.
- The source code and repository are processed to extract the information to be presented.
- A presentation layout for the information is constructed.
- The presentation layout is displayed as hypertext. Requests to traverse hypertext links are intercepted and passed back to the extraction and layout subsystems to generate new presentations.

These components will be described in further detail below, but first the information content that these components operate on will be discussed.

## 5 Underlying Knowledge

In order to generate appropriate software descriptions, I-Doc requires a variety of information about the software and its design. Some of this information can be extracted directly from the code and from CASE repositories. Other information must be added to the design in the form of annotations.

First, a hierarchical decomposition of the design into functional components is required, together with the types of the components' inputs and outputs. The pattern of data flow among components is necessary as well.

In order to present the information flow between modules in natural language, some additional characterizations of the data and the operations on the data are required. First, it is useful to classify the type of operation being performed by the component. Classifications that have been identified so far as useful include create, destroy, filter, insert, remove, retrieve, process, and validate. For example, the module Scan-Generation is classified as creating scan patterns. Modules whose function is to validate data were omitted from the summary in Figure 3, under the assumption that initial descriptions should assume that all data is valid, and methods for handling exceptional data will be described later. The content and use of data structures is characterized as well. Data structures are categorized as to whether they represent objects, aggregates of objects (sets, sequences, etc.), or names of objects. This enables I-Doc to refer to the output of Output-Processing as a set of contacts, regardless of the actual data representation (e.g., an array of pointers to contact objects).

Function categorizations can be applied to segments of components or groups of components, as well as to individual components. For example, a set of routines may be employed to process radar data, or a set of statements may be employed to validate the data.

Another type of information that plays a prominent role in I-Doc descriptions is information about requirements, particularly nonfunctional requirements. A set of attributes such as speed requirements or accuracy requirements may be associated with functional components and data.

In order to determine how to render data dictionary elements most effectively in natural language, I-Doc uses grammatical annotations. The annotations used in I-Doc are based on those used in the ARIES requirements acquisition system for annotating specifications [7]. Data expressing relationships between objects are categorized as to whether they are attributes, actions, circumstances, classes, or relations. Attributes describe properties of the object, actions describe actions that involve the object, circumstances describe states of the object, classes identify categories to which the object belongs, and relations are default data relationships. Objects participating in such relationships can assume one of several grammatical categories, e.g., actor, goal, location, or beneficiary. These categories are drawn from case grammars for natural language [5].

Finally, descriptions of classes of behavior, called *scenarios*, are useful in the description process. Scenarios are useful for defining system requirements, as a way of describing types of behavior that a system should or should not exhibit [1], which can be used to validate system specifications. They are intended to serve two roles within I-Doc. First, scenarios can be used to illustrate system behavior. Second, scenarios provide the context in which to describe systems. If the I-Doc user is trying to fix a bug, for example, then if a scenario illustrating the bug is available it can be used by I-Doc to focus on describing those components of the system that are relevant to the bug.

## 6 Knowledge Acquisition

The information described above is acquired from a variety of sources. These sources will be described below. It is important to emphasize, though, that I-Doc can still generate comprehensible system descriptions without much of this information. The added design information is used to improve the quality of system descriptions, and can be acquired when and as appropriate.

Some information is available in front-end CASE tools such as Software through Pictures [6]. I-Doc will have the ability to query one or more such CASE repositories in order to extract such information if available.

Another means of acquiring the documentary information is through a special acquisition interface. This method is used especially for inputting grammatical annotations and design component classifications. These annotations need not be selected directly; instead, the person entering the information can request that I-Doc attempt to provide the annotations automatically, and present samples of natural language output based upon those annotations. Although the grammatical annotations are based on linguistic concepts that may be unfamiliar to software engineers, it is easy to see when the generated natural language is awkward or incorrect. Once the user has selected from among alternative descriptions generated by I-Doc, I-Doc saves the annotations used to produce that sample output.

There are three other ways in which information for constructing system descriptions is obtained. One means is through the use of object hierarchies. If an object class has a particular set of attributes, its specializations are likely to have similar attributes. A second approach is to annotate the models used in automated program synthesis systems. The knowledge bases of specialized knowledge-based synthesis systems, such as user interface development systems, can be augmented to support the generation of documentation and help as well [12]. Integration of I-Doc with one or more such systems is an option being considered for future development.

The third source of design information for documentation is code analysis. Analysis routines can detect components that appear to be creating objects, inserting into or removing from data aggregates, validating data, etc. Such analysis is further facilitated when some design components are already annotated; e.g., when data is designated as an aggregate, it makes sense to look for routines that add and remove elements from that aggregate. The analysis capabilities of Reasoning Systems' reengineering technology, in particular Refine/Ada, are being used as the basis for such analysis capability.

Use of the above capabilities for extracting relevant design annotations is one way in which reengineering technology plays a prominent role in the construction of documentation in I-Doc. In fact, in order for automated generation of documentation to succeed it must be viewed at least partly as a reverse engineering enterprise. Preparing a design for document generation involves adding design information that is not present in the original design. Front-end CASE tools, executable specification languages, and other advanced forward engineering technologies can reduce the amount of information that must be recaptured, but even then some information must be made explicit that is implicit in the design. Thus a combination of interactive design capture and design recovery techniques appear to be essential.

## 7 Repository Storage and Maintenance

As design information is acquired it must be associated with the code and maintained. The basic mechanism being used to achieve this is to add attributes to the Refine representation of program parse trees. In order to permanently associate these attributes with the code, the following techniques are planned. First, Ada pragmas will be used to insert the information directly into the code. Pragmas were originally intended to record information to guide compilers in generating object code. In an analogous fashion they can be used to guide the generation of system descriptions.

In the longer term, it may be appropriate to extend the data model of a CASE tool such as Software through Pictures in order to record the design information. However, that should not be a substitute for inserting design information directly into the source code. In order for the captured design information to be useful, it must continue to be associated with the code as it is maintained over time. At the present time the best way of ensuring this is to integrate the design information into the source code, so that maintenance using a front-end CASE tool is not required. For languages that do not have constructs similar to pragmas, the approach will be to add grammatical extensions to the source language to provide such constructs, which can then be removed from the source code via transformations, using a tool such as Refine.

## 8 Inputting User Queries

If the design knowledge associated with a design is sufficiently rich, it can be used as the basis for answering a variety of queries. Following the approach taken in Lehnert's original work on question answering [8] and further developed by research in expert system explanation, such as the work of Moore and Swartout [10], common questions about software have been categorized into types. In the initial version of I-Doc, these question types will be available to the user as explicit choices from which the user can choose. Examples of question types include Describe Function, Describe Design, Describe Interface, and Describe Use. In subsequent versions these choices will be automated to a greater extent, so that the user can simply request "Describe" and the system will construct a combined description appropriate to the context.

The main difference between question input in I-Doc and question input in expert system explanation is the use of hypertext as a medium for posing questions. Question-answering systems such as Moore and Swartout's facility in the Explainable Expert System (EES) system allow the user to point to an element of a description and ask one of several follow-up questions about it. In hypertext, the interaction is more limited—the user clicks on a section of text where a link begins, causing the system to jump to the other end of link. The user does not have the option of selecting one of several operations to perform on the link. In order to overcome this difficulty, we are experimenting with making choices explicit as lists of selectable items at the bottom of the hypertext display, as shown in Figure 3. It remains to be seen how effective this technique is in comparison to more ordinary menu-based approaches.

## 9 Extracting Relevant Information

Once the type and the object of the query have been chosen, the information suitable for inclusion in the presentation must be retrieved and presented. Retrieval of relevant information can be easily accomplished using the retrieval and query mechanisms available in Refine, Software through Pictures, and other tools. Difficulties arise, though, when the amount of retrieved information is too much to present in such a way that the reader can assimilate it. In such cases filtering techniques may be employed to focus on the information of greatest potential interest to the reader.

The filtering procedures in I-Doc proceed by marking code as either central, interesting, or ignored. Central components are the primary focus of the description, and consist of all code satisfying a particular criterion, such as code matching steps a scenario. Components are designated interesting because of their interactions with the central components. Ignored components are removed because they can be explicitly filtered out, or because they are not found to be central or interesting. The following are some criteria that have been found in pencil-and-paper studies to be useful in determining code to be central or ignored; others are expected to be identified.

- Exceptional case handling. In some descriptions, such as the one shown in Figure 3, code for handling exceptional cases is unimportant; in other cases (such as when fixing bugs) such code may be central. Either way, such can be detected in a significant number of cases. Ada has explicit constructs for raising exceptions. Additionally, if a variable is being used as an error flag, tests of the variable can be detected automatically, and branches of the code marked accordingly.

- Code that sets and/or reads a variable or attribute.

- Code that assigns an attribute or variable to a specific value, or checks that the attribute or variable has a specific value.

Once components are found be central, surrounding code is often marked as interesting and is therefore included. The motivation for this is to provide some context so that the focussed operations are more easily understood. For example, if a routine sets an interesting attribute of an object, assignments to other attributes of the same object in the same procedure may be included as well.

The transformation facilities in Refine are to be employed to implement this filtering process. The transformation pattern language can be used in some cases to detect the code that is of interest. In other cases, transformations can be used to perform simple expression simplifications in order to facilitate pattern matching. A tool that could recognize all instances of potentially interesting code would require a more powerful simplifier than is envisioned for I-Doc. Instead, I-Doc will either inform the reader when the view being presented is possibly incomplete, or simply provide a view that is somewhat broader than is strictly necessary.

## 10 Presentation Layout

Once relevant information has been identified, I-Doc must determine what media to use to present the information, and how to present the information using those media. In the long term, we hope to be able to employ a variety of presentation media, including mixtures of text and diagrams. At first, though, the focus will be on natural language generation. Such generated text may be supplemented with diagrams if such diagrams have already been constructed and are available, as is currently the case with CASE-generated documents.

The overall structure of each description that is generated will be determined by a presentation template, a library of which will be included in I-Doc's knowledge base. These templates provide standard ways of presenting different aspects of a system. Each template will have a set of selection criteria, based upon the amount of various types of information that is to be presented, and the assumed level of expertise of the user. Slots within the template are then filled out using a natural language generator.

The natural language generator consists of two components. The basic generation work is performed by a Functional Unification Generator, which can construct arbitrary sentences from attribute-value descriptions of the content to be expressed. A second phrase selection component constructs attribute-value patterns in a form suitable for input to the Functional Unification Generator, according to directives contained within the system description templates. This architecture was used successfully in ARIES and the KBSA Concept Demonstration [11] to generate text descriptions rapidly. Functional Unification Grammar has been evaluated against competing methods for natural language generation, and has been found to be both flexible and efficient [9].

## 11 Presentation Delivery

As indicated in Section 3, Mosaic is being employed as the hypertext delivery mechanism for I-Doc. Although there are various commercial products available that provide hypertext capability, Mosaic has a set of features that make it particularly suitable to dynamic documentation generation.

- Mosaic provides interfaces to external programs, so that the user buttoning on a hypertext link can cause a program to be invoked.
- It interprets hypertext formatting commands dynamically, as needed; this makes it possible to construct a hypertext document dynamically and display it.
- It runs on a variety of platforms, including X Windows, Microsoft Windows, and Macintosh.
- It is public domain, and the source code is readily available, making it possible to customize the system for use within I-Doc.

## 12 Further Plans and Prospectus

The I-Doc project has just begun; it is expected to continue for another three years. The project plans to make available intermediate versions of the system on a periodic basis. In the near term, the capability will be demonstrated on a military application selected in collaboration with the US Air Force's Wright Laboratory.

If resources are available, I-Doc will be extended so that it can describe systems to other classes of users besides maintainers, such as clients and end-users. This will further enhance the value of documentation generation technology to software development projects.

In the long run, dynamic generation of documentation will prove successful if the perceived benefits outweigh the costs of additional design capture. The benefits will be particularly apparent in projects that undergo periodic design reviews, since automated documentation should prove to be a great benefit to the design review process. Reengineering and program synthesis technology will gradually reduce the amount of interaction between developers and I-Doc necessary for design capture. In the near term, making enhanced hypertext capabilities available to developers is likely to bring benefits in itself. Hypertext is gradually being adopted in software development practice, and the activities of the I-Doc project will aim to help accelerate this trend.

## 13 Acknowledgements

The author wishes to thank Bill Swartout and Richard Angros for their contributions to this effort, and John Salasin and Marc Pitarys for their support. Sheila Coyazo assisted with preparation of the article. This work is sponsored by the Advanced Research

Projects Agency and administered by Wright Laboratory, Air Force Materiel Command, under Contract No. F33615-94-1-1402. Views and conclusions contained in this paper are the author's and should not be interpreted as representing the official opinion or policy of the U.S. Government or any agency thereof.

## References

[1] K. Benner, M.S. Feather, W.L. Johnson, and L. Zorman. The role of scenarios in the software development process. In *Proceedings of the IFIP W8.1 Working Conference on Information System Development Process*, 1993. To appear.

[2] E. Bina and M. Andreessen. NCSA mosaic home page. Available from World Wide Web server www.ncsa.uiuc.edu.

[3] J.M. Carroll. The minimal manual. *Human-Computer Interaction*, 3(3):123–153, 1988.

[4] R.A. Falcioni and R.L. Buvel. Modular embedded computer software (MECS): Interim report. Technical Report WL-TR-92-1113, Wright Laboratory, Wright Patterson AFB, OH, 1990.

[5] C.J. Fillmore. The case for case. In *Universals in Linguistic Theory*, pages 1–88. Holt, Reinhart and Winston, New York, NY, 1968.

[6] Interactive Development Environments. *Software through Pictures: Fundamentals of StP*, 1993.

[7] W.L. Johnson, M.S. Feather, and D.R. Harris. Representation and presentation of requirements knowledge. *IEEE Trans. on Software Engineering*, 18(10):853–869, October 1992.

[8] W.G. Lehnert. *The Process of Question Answering*. Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1978.

[9] K.R. McKeown and M. Elhadad. *A Contrastive Evaluation of Functional Unification Grammar for Surface Language Generation: A Case Study in the Choice of Connectives*, pages 351–392. Kluwer Academic Publishers, Norwell, MA, 1991.

[10] J.D. Moore and W.R. Swartout. *A Reactive Approach to Explanation: Taking the User's Feedback into Account*, pages 3–44. Kluwer Academic Publishers, Norwell, MA, 1991.

[11] J.J. Myers and G. Williams. Exploiting metamodel correspondences to provide paraphrasing capabilities for the concept demonstration. In *Proceedings of the 5th KBSA Conference*, pages 331–345, Syracuse, NY, September 1990. Defense Technical Information Center.

[12] P. Szekely, P. Luo, and R. Neches. Facilitating the exploration of interface design alternatives: The HUMANOID model of interface design. In *Proceedings of CHI'92, The National Conference on Computer-Human Interaction*, pages 507–515, May 1992.