CogentHelp: NLG meets SE in a tool for authoring dynamically generated on-line help

Michael White and **David E. Caldwell**
CoGenTex, Inc.
840 Hanshaw Road
Ithaca, NY 14850, USA
{mike,ted}@cogentex.com

Abstract

CogentHelp is a prototype tool for authoring dynamically generated on-line help for applications with graphical user interfaces, embodying the "evolution-friendly" properties of tools in the literate programming tradition. In this paper, we describe CogentHelp, highlighting the usefulness of certain natural language generation techniques in supporting software-engineering goals for help authoring tools — principally, quality and evolvability of help texts.

1 Introduction

CogentHelp is a prototype tool for authoring dynamically generated on-line help for applications with graphical user interfaces (GUIs). In this paper, we describe CogentHelp, highlighting the usefulness of certain natural language generation (NLG) techniques in supporting software-engineering (SE) goals for help authoring tools — principally, quality and evolvability of help texts.

To our knowledge, CogentHelp is unique in that it is the first operational prototype to embody the **"evolution-friendly"** properties of tools in the literate programming tradition (Knuth, 1992) — e.g., the by now well-known `javadoc` utility for generating API documentation from comments embedded in Java source code (Friendly, 1995; cf. also Johnson and Erdem, 1995; Priestly et al., 1996; Korgen, 1996) — in a tool for generating end user–level documentation. CogentHelp is also unusual in that it is (to date) one of the few tools to bring NLG techniques to bear on the problem of **authoring** dynamically generated documents (cf. Paris and Vander Linden, 1996; Knott et al., 1996; Hirst and DiMarco, 1995); traditionally, most applied NLG systems have focused on niches where texts can be generated fully automatically, such as routine reports of various types (e.g. Goldberg et al., 1994; Kukich et al., 1994) or explanations of expert system reasoning (cf. Moore, 1995 and references therein).

While striving to design highly sophisticated, fully automatic systems has undoubtedly led to a deeper understanding of the text generation process, it has had the unfortunate effect (to date) of limiting the use of techniques pioneered in the NLG community to just a few niches where high knowledge acquisition costs stand a chance of being balanced by substantial volume of needed texts (cf. Reiter and Mellish, 1993). By joining the emerging authoring support crowd and endeavoring to create new opportunities in automated documentation, we hope to contribute to the broader acceptance and visibility of NLG technology in the overall computing community.

The rest of the paper is organized as follows. In Section 2 we discuss the software engineering goals for CogentHelp. In Section 3 we provide background on automated documentation and identify where CogentHelp fits in this picture. In Section 4 we give a brief overview of the CogentHelp system. In Section 5, we highlight the NLG techniques used in support of the software engineering goals identified in Section 2. In Section 6 we describe CogentHelp's authoring interface. Finally, in Section 7 we conclude by discussing the outlook for CogentHelp's use and further development.

2 Software Engineering Goals

From a software engineering perspective, we set out to achieve three main goals in designing CogentHelp, each of which has various aspects. The first of these goals is end user–oriented, whereas the latter two are developer-oriented.

The first goal is to promote quality in the resulting help systems, which includes promoting

- **Consistency** — the grouping of material into help pages, the use of formatting devices such as

headings, bullets, and graphics, and the general writing style should be consistent throughout the help system;

- **Navigability** — the use of grouping and formatting should make it easier to find information about a particular GUI component in the help system;
- **Completeness** — all GUI components should be documented;
- **Relevance** — information should be limited to that which is likely to be of current relevance, given the current GUI state;
- **Conciseness** — redundancy should be avoided;
- **Coherence** — information about GUI components should be presented in a logical and contextually appropriate fashion.

The second goal is to facilitate evolution, which includes facilitating

- **Fidelity** — the help author should be assisted in producing complete and up-to-date descriptions of GUI components;
- **Reuse** — wherever possible, the help author should not have to write the same text twice.

The final goal is to lower barriers to adopting the technology, which has principally meant providing an authoring interface which makes the benefits of the system available at a reasonable cost in terms of the understanding and effort required of the help author.

3 Automated Documentation

The main idea of CogentHelp is to have developers or technical writers author the reference-oriented part of an application's help system[1] in small pieces, indexed to the GUI components themselves, instead of in separate documents (or in one monolithic document). CogentHelp then dynamically assembles these pieces into a set of well-structured help pages for the end user to browse.

The principal advantage of this approach is that it makes it possible to keep the reference-oriented part of an on-line help system up-to-date automatically, since both the application GUI and the documentation can be evolved in sync. This benefit of generating both code and documentation from a single source[2] has long been recognized, both in the NLG community (cf. Reiter and Mellish, 1993; Moore, 1995; and references therein) and in the SE community, where it is recognized under the banner of literate programming tradition (Knuth, 1992). Other important benefits stem from supporting the separation of the content of the document to be generated (descriptions of individual components) from the structure of the document (how the content is distributed and formatted): this allows the author to focus on writing accurate component descriptions, and avoid much of the drudgery involved in creating a complex hypertext document manually.

To better understand where CogentHelp fits in, it is instructive to compare it with the closest reference points in the NLG and SE communities. On the NLG side, there is the Drafter system (Paris and Vander Linden, 1996), which generates drafts of instructions in both English and French from a single formal representation of the user's task. Drafter is considerably more ambitious in aiming to automate the production of multilingual, task-oriented help; at the same time, however, Drafter is more limited in that it is not evolution-oriented, aiming only to generate satisfactory initial drafts (whence its name). This is in large part due to the fact that Drafter's input is centered around task representations which are not typically used for GUI-level tasks in software engineering practice; in contrast, nearly every GUI builder provides some form of GUI resource database which could be used as input to CogentHelp.

On the SE side, the nearest reference points are a bit more distant, as CogentHelp has more in common with developer-oriented tools such as the `javadoc` API documentation generator distributed with Sun Microsystems' Java Developers Kit (Friendly, 1995) than with currently available help authoring tools. While several current GUI-development environments include tools which generate an initial, "skeleton" help system for an application (with topics that correspond to the widgets in the GUI), to our knowledge CogentHelp is the first operational prototype to implement the "evolution-friendly" properties of a tool like `javadoc` in a system for generating end user–level documentation. Unlike the "skeleton generators" mentioned above,

[1] By *the reference-oriented part,* we mean the part of a help system which describes the functions of individual windows, widgets, etc., as opposed to more general or more task-oriented information about the application; other than providing an easy way of linking to and from CogentHelp-generated pages, CogentHelp leaves task-oriented help entirely to the author.

[2] Note that this "single source" need only be *virtual* — *physically* single-source code and documentation can have its drawbacks, which are not however inherent to the approach (Priestley et al., 1996).

which require the help author to start from scratch each time the skeleton is regenerated in response to GUI modifications, CogentHelp supports help authoring throughout the software life cycle.

4 System Overview

CogentHelp takes as input various human-written text fragments (or "help snippets") indexed to GUI resource databases, which provide some useful help-related information in the form of types, labels, locations and part-whole relations for GUI widgets. CogentHelp generates HTML help files, which can be displayed on any platform for which a Web browser is available. It is designed to support efficient navigation through the help system, through the use of intelligent, functionally structured layout as well as through an expandable/collapsible table of contents and "thumbnail sketch" applet. These points will be elaborated upon below.

CogentHelp operates in two modes: a "static" mode, which does not make use of run-time information, and a "dynamic" mode, which uses widget run-time status information to produce more specialized, contextually appropriate messages. The static mode is useful in the authoring process, since it displays all available help information and has simpler architectural requirements.

In evolving the design of CogentHelp, we have employed a rapid-prototyping approach in working with our TRP/ROAD[3] consortium partners at Raytheon, who are serving as a trial user group. The full CogentHelp component architecture and dependencies reflects the particular requirements of this group, and are as follows. CogentHelp itself is a hypertext server written in Java, making it highly cross-platform. CogentHelp currently works with applications built using the Neuron Data cross-platform GUI builder; while it is not dependent on this product in any conceptually substantial way, using other GUI builders would require a porting effort. To retrieve run-time information, CogentHelp uses Expersoft's PowerBroker ORB for inter-process communication; in comparison to the Neuron Data connection, other IPC methods could be more easily substituted. Finally, CogentHelp displays hypertext in Netscape Navigator, using HTTP to mediate access to the dynamically generated texts; since Netscape Navigator remains the most widely used cross-platform browser, we have yet to investigate using other browsers.

[3] A DARPA-sponsored Technology Reinvestment Program for Rapid Object Application Development, led by Andersen Consulting.

5 NLG Techniques

Although CogentHelp is by no means a typical NLG system — insofar as it is incapable of generating useful texts in the absence of human-authored help snippets — it does employ certain natural language generation techniques in order to support the software engineering goals described above. These techniques fall into two categories, those pertaining to knowledge representation and those pertaining to text planning.

5.1 Knowledge Representation

In developing CogentHelp, we have taken a minimalist approach to knowledge representation following a methodology for building text generators developed over several years at CoGenTex. As will be explained below, this approach has led us to (i) make use of what amounts to a large-grained "phrasal" lexicon and (ii) devise and implement a widget-clustering algorithm for recovering functional groupings, as part of an Intermediate Knowledge Representation System (IKRS).

5.1.1 Phrasal Lexicon

As Milosavljevic et al. (1996) argue, to optimize coverage and cost it makes sense to choose an underlying representation which

- makes precisely those distinctions that are relevant for the intended range of generated texts; and
- is no more abstract than is required for the inference processes which need to be performed over the representation.

They go on to argue that besides eliminating a great deal of unnecessary 'generation from first principles,' this approach complements their use of a phrasal lexicon (Kukich, 1983; Hovy, 1988) at the linguistic level.

Applying essentially the same approach to the design of CogentHelp, we first determined that for the intended range of generated texts it suffices to associate with each widget to be documented a small number of atomic propositions and properties, identifiable by type. Next we determined that since no inference is required beyond checking for equality, these propositions and properties can be conflated with their linguistic realizations — i.e., the indexed, human-authored help snippets CogentHelp takes as input. While we did not originally think of CogentHelp's collection of input help snippets as a phrasal lexicon *a la* Milosavljevic et al., in retrospect it becomes evident that this collection can be viewed as

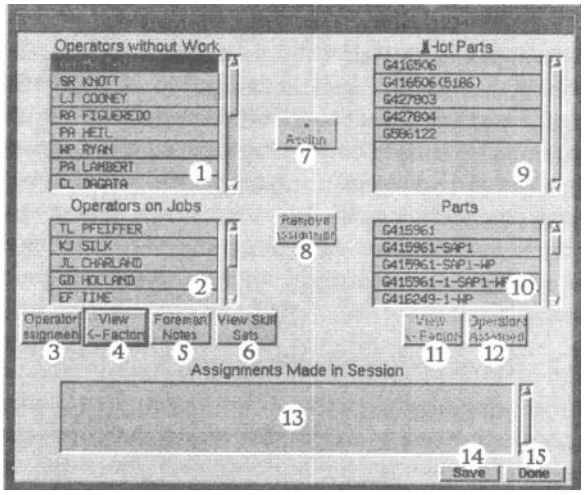

Figure 1: Sample application window

tantamount to one; of course, since these snippets vary in size from phrases to paragraphs, the term "phrasal" is not entirely accurate.

The types of snippets in current use include a one-sentence short description of the relevant GUI component; a paragraph-sized elaboration on the short description; various phrase-sized messages concerning the conditions under which it is visible or enabled, if appropriate; and a list of references to other topics. In the case of the phrase-sized messages, each of these fields is accompanied by a syntactic frame which prompts the author to provide consistent syntax — for example, entries in the `WHEN_ENABLED` field should fit the frame

> This element is enabled when _____.

To facilitate equality checking and promote text reuse, a mechanism is provided enabling the author to alias the message for one widget to that of another.

5.1.2 IKRS

To enhance the modularity and robustness of a practical text generator, Korelsky et al. (1993) argue for the use of an Intermediate Knowledge Representation System (IKRS, pronounced "Icarus") to bridge the gap between what the text planner would like to access and what is actually found in the information base of an application program. A remarkably similar idea has been independently developed by Lester and Porter (1996), under the heading of KB Accessors.

One purpose of an IKRS, Korelsky et al. suggest, is to provide a component in which to locate domain-level inferencing not provided by the application program. Note that while this type of inferencing is motivated by text planning needs, it is still about the domain rather than about natural language communication, and thus does not belong in the text planner itself.

In developing CogentHelp, we encountered a need for just this sort of inferencing in order to support sensible layout. The problem we faced was how to logically arrange descriptions of widgets within a help page (or set of help pages) describing a window, which is the basic unit of organization in CogentHelp. As will be explained below, grouping widgets by type was considered inadequate, because doing so would obscure functional relationships between widgets of different types. A naive spatial sorting was likewise considered inadequate, as this would inevitably separate elements of a functional group appearing in a certain area of the window. Unfortunately, since these functional groups are often not explicitly represented in GUI resource databases, we appeared to be at an impasse.

To illustrate the problem, consider the sample application window shown in Figure 1, from a prototype of an application under development by our trial user group at Raytheon. This window, whose purpose is to allow a manufacturing shop floor foreman to assign operators to parts, is organized as follows: on the left there are two list boxes for operators (1, 2), with buttons beneath them to access information about these operators (3, 4, 5, 6); on the right there are two list boxes for parts (9, 10), with buttons beneath them to access information about these parts (11, 12); in the middle there are two buttons for making and removing assignments (7, 8); towards the bottom there is a list box showing the assignments made so far (of which there are none here — 13); and at the bottom there are standard buttons such as Save and Done (14, 15 — the Help button would go here). Given this organization, consider first arranging descriptions by type, and alphabetizing: besides cutting across the implicit functional groupings, arranging descriptions in this way would end up putting the two View K-Factors buttons (4, 11) in sequence, without any indication of which was which! Now consider a simple top-down, left-to-right spatial sort: again, this would inevitably yield a rather incoherent ordering, such as the Operators without Work list box (1), the Hot Parts list box (9), the Assign button (7), the Operators on Jobs list box (2), the Parts list box (10), the Remove Assignment button (8), etc.

The solution to this problem was to develop, as part of our IKRS, a method of recovering these func-

tional groups using spatial cues as heuristics; the reason this approach might be expected to work is that in a well-designed GUI, functionally related widgets are usually clustered together spatially in order to make the end user's life a bit easier. We began with the fairly standard hierarchical agglomerative algorithm found in (Stolcke, 1996). Stolcke's algorithm is an order n^2 one that iteratively merges smaller clusters into bigger ones until only one cluster remains; new clusters are formed out of the two nearest clusters in the current set, ensuring that the results are independent of the order in which clusters are examined. After some experimentation, we modified this algorithm to better suit our needs, resulting in the following three differences: (i) to create clusters with more than two elements, we continue adding elements to the newly created cluster until a certain distance threshold is exceeded; (ii) we represent the new cluster using its bounding box, rather than using an average of its elements; and (iii) we restrict the clustering to not operate across explicit groups, such as those formed using panels.

With any clustering approach, there is always the tricky matter of determining a suitable distance measure. After trying out a variety of features, what we found to work surprisingly well was a simple weighted combination of proximity, alignment and type identity. In particular, in a test suite of around 15 windows provided to us by our trial user group, we obtained reasonable results (no egregiously bad groupings) on all of them without undue sensitivity to the exact weights. In the case of the window shown in Figure 1, the clustering procedure performs exactly as desired, yielding precisely the groupings used in the description of this window given above — i.e.: (((1, 2), (3, 4, 5, 6)), ((9, 10), (11, 12)), (7, 8), 13, (14, 15)).

Once the IKRS has heuristically recovered clusters of widgets likely to form a functional group, these clusters — as well as any explicitly represented groups, e.g. widgets contained within a panel of a window — can be used as a basis for help layout, as discussed below.

5.2 Text Planning

At the core of CogentHelp is the text planner. The text planner builds up HTML trees starting from an initial goal, using information provided by the IKRS, following the directives coded in CogentHelp's text planning rules. These HTML trees are then linearized into an ascii stream by a separate formatter, so that they can be displayed in a web browser (cf. Section 4).

The text planner is constructed using Exemplars for Java, a lightweight framework for building object-oriented text planners in Java which has been developed in parallel with CogentHelp (White, 1997). In this framework, text planning rules — the exemplars, so-called because they are meant to capture an exemplary way of achieving a communicative goal in a given communicative context — are objects which cooperate to efficiently produce the desired texts. While space precludes a detailed description, it is worth noting that Exemplars for Java supports abstraction, specialization and content-based revisions, all of which have proved useful in the present effort.

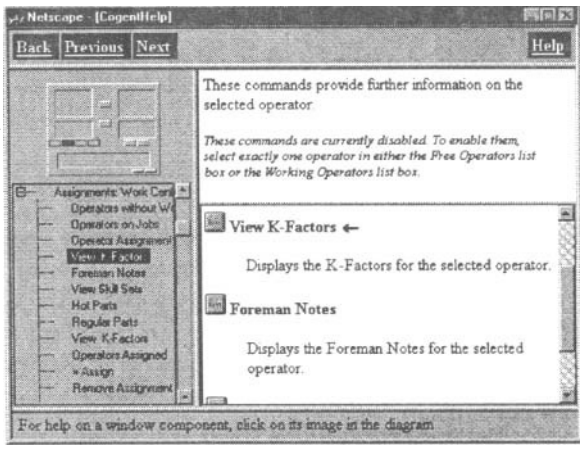

Figure 2: A sample help page

In developing the exemplars for CogentHelp, we have made use of three NLG techniques: structuring texts by traversing domain relations, automatically grouping related information, and using revisions to simplify the handling of constraint interactions. The first two of these make life simpler for the end user, while the third makes life simpler for the developer.

5.2.1 Capitalizing on Domain Structure

When text structure follows domain structure, one can generate text by selectively following appropriate links in the input (Paris, 1988; Sibun, 1992). In the case at hand, we have chosen to use the group and cluster structure combined with a top-down, left-to-right spatial sort: while such a spatial sort alone is insufficient, as we saw above, a spatial sort which respects functional groups turns out to work well.

Returning to the example of Section 5.1.2 (regarding the window shown in Figure 1), traversing the clusters in this way yields a listing which (naturally!) mirrors the order and groupings of the one-sentence description of the window's organiza-

tion we have provided — that is, following a general description of the window, there are descriptions of the two operators list boxes (1, 2), followed by descriptions of the four buttons providing additional information on operators (3, 4, 5, 6), followed next by the part list boxes (9, 10) and the buttons associated with them (11, 12), and so on. This is (partially) illustrated in Figure 2, which shows a sample CogentHelp-generated help topic. Note that the list of widgets in the dynamic TOC on the left side of the page is arranged according to this traversal; consequently, stepping through the contents (using the TOC or the Next button) for this window will lead from widget to widget and cluster to cluster in a sensible fashion. In the particular topic shown, the user has reached the second button (View K-Factors) of the group of four buttons beneath the Operators list boxes, as can be seen from the highlighting in the thumbnail sketch applet (cf. Section 6).

The use of domain structure–driven text planning is central to supporting the software engineering goals identified in Section 2. Rather obviously, generating-by-rule helps to achieve consistency, completeness and fidelity, eliminating much mind-numbing drudgery along the way. A bit less obvious, perhaps, is the fact that this technique should help to achieve navigability and coherence: by presenting descriptions of widgets in a natural order — i.e., in the order in which the user is apt to encounter them in scanning the GUI — we hope to make it easier for the user to find desired information; and, by keeping together descriptions of widgets which are heuristically determined to be functionally related, we hope to make it easier for the user to quickly grasp the organization of both the interface and the on-line help.

5.2.2 Grouping

Grouping related information and presenting shared parts just once is a well-known NLG technique for achieving conciseness and coherence (Reiter and Mellish, 1993). In a reference-oriented document such as an on-line help system, similar or identical descriptions will often be appropriate for elements which have similar or identical functions. To indicate these similarities, as well as to save space, it makes sense to group these descriptions when possible.

As mentioned in Section 5.1.1, we allow developers to alias messages to promote text reuse, as well as to facilitate equality checking. When the text planner detects (via the IKRS) that a phrase-sized message (such as `TO_ENABLE`) is the same for a group of widgets, it generates a description that applies to the whole group, rather than repeating the same description several times in close proximity. Note that this group description is made possible by the use of a phrasal lexicon, which has been designed to allow the author's messages to make sense in a variety of contexts.

To illustrate, let us have another look at Figure 2. In the upper right frame of the page, note that there is the following description of how to enable all of the four buttons below the operators list boxes, rather than a repetition of the same message four times in close proximity:

> *These commands are currently disabled. To enable them, select exactly one operator in either the Free Operators list box or the Working Operators list box.*

This group-level description appears here because (i) the author, realizing that these buttons are enabled under the same conditions, entered the `TO_ENABLE` message "select exactly one operator in either the Free Operators list box or the Working Operators list box" for one button, and aliased this message for the other; and (ii) the text planner, detecting (via the IKRS) that the `TO_ENABLE` messages for the entire group were the same, and that these buttons were currently disabled (a run-time fact), prepended "These commands are currently disabled. To enable them," to the shared message to yield what appears in Figure 2.

5.2.3 Revisions

While the central use of domain structure–driven text planning makes it possible to generate text in a relatively straightforward, top-down fashion, as variations are added a one-pass top-down approach can become cumbersome. The reason why is this: In evolving the text planning rule base, it makes sense to localize decisions as much as possible; however, to handle rule interactions in a single pass, one is forced to centralize these decisions (which can become cumbersome). To simplify matters, it is often appropriate to generate an initial version naively, then carry out revisions on it in a subsequent pass (cf. Robin, 1994; Wanner and Hovy, 1996).

In CogentHelp, interactions that are cumbersome to anticipate arise in dealing with the various optional phrase-sized messages whose inclusion conditions differ between the static and dynamic mode. To elaborate, let us consider once more the help page shown in Figure 2. Had the four buttons described in the lower right frame been enabled rather than disabled (at run-time), the group-level `TO_ENABLE` message would have simply been left out, in order to en-

hance relevance and conciseness;[4] on the other hand, had this page been generated in static mode (where such run-time conditions are not known), the text planner would have again included the description, though this time with the less specific "To enable these commands," prepended instead. Now, since the various messages associated with a widget have slightly different inclusion conditions, it makes sense to localize these inclusion conditions to a text planning rule for each message (the common parts of these conditions are shared via inheritance). At the same time, however, there is a need to know whether any of these messages will in fact appear, in order to decide whether to include the second paragraph in the upper right frame, as well as the italics element. In a one-pass top-down approach, this need would force the various inclusion conditions to be cumbersomely centralized; with revisions, in contrast, one can simply add the paragraph and italics element during the first pass, then check during a second pass whether any of the optional messages for this paragraph did in fact appear, removing the superfluous HTML elements if not.

6 Authoring

In informal usability tests, we have gathered much useful information about areas in which CogentHelp could be improved — the most important of these being ease of authoring. A previous version of the authoring interface, which relied on the resource editor of the Neuron Data GUI builder, proved unsatisfactory, as it (i) required excessive clicking for the author to navigate from snippet to snippet, and (ii) failed to provide sufficient context, making it unnecessarily difficult for the author to adhere to the CogentHelp authoring guidelines.

The current authoring interface, shown in Figure 3, uses CogentHelp's existing architecture (together with the HTTP forms protocol) to allow the user to edit the text snippets for each widget in substantially the same context they would inhabit in generated help topics. This design provides maximal realism for the author, especially since one can switch between editing and browsing mode at the click of a button to preview the generated help.

Another feature illustrated in Figure 3, as well as Figure 2, owes its inspiration to our trial user group at Raytheon. Our users were concerned about the ease of visual navigation of help pages, and had experimented with using manually coded (and

[4]With a perfectly intuitive GUI, the user would never need to know this information, as commands would always be enabled when the user expects them to be.

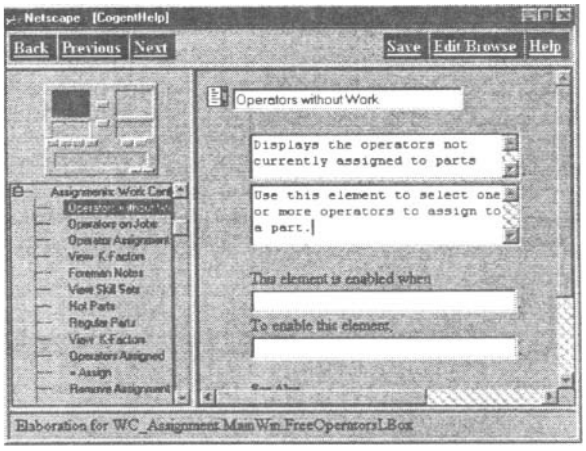

Figure 3: CogentHelp in authoring mode

thus difficult to evolve) image maps superimposed over bitmaps of each application window. This concern prompted us to develop an automatically generated "thumbnail sketch" of the current GUI window, which appears in the upper left corner of the help window (in a Java applet along with the table of contents) and contains hyperlinks for each of the widgets on the window (these hyperlinks display the corresponding help topics on the right-hand side). The automatically generated thumbnail images require no intervention on the part of the help author, and thus are guaranteed to be up-to-date; furthermore, their abstract nature gives them certain advantages over actual bitmaps: they do not present information which is redundant (since the actual window in question will usually be visible) or inconsistent (static bitmaps fail to capture widgets which are enabled/disabled or change their labels in certain situations).

7 Outlook

To date we have gathered substantial feedback on CogentHelp functionality from our trial user group at Raytheon, especially on the need for authoring support and visual navigation aids. We are optimistic that this group will find CogentHelp suitable for actual use in developing a production-quality help system by the end of our Rome Laboratory–sponsored software documentation SBIR project, in mid-1997. Also by project end, we hope to port CogentHelp to a more affordable, Java-based GUI builder, in order to make it useful to a much broader community of developers.

Acknowledgements

We gratefully acknowledge the helpful comments and advice of Ehud Reiter, Philip Resnik, Keith Vander Linden, Terri SooHoo, Marsha Nolan, Doug White, Colin Scott, Owen Rambow, Tanya Korelsky, Benoit Lavoie and Daryl McCullough. This work has been supported by SBIR award F30602-94-C-0124 from Rome Laboratory (USAF) and by the TRP/ROAD cooperative agreement F30602-95-2-0005 with the sponsorship of DARPA and Rome Laboratory.

References

Lisa Friendly. 1995. The design of distributed hyperlinked programming documentation. In *International Workshop on Hypermedia Design.*

Eli Goldberg, Norbert Driedger, and Richard Kittredge. 1994. Using natural-language processing to produce weather forecasts. *IEEE Expert*, pages 45–53.

Graeme Hirst and Chrysanne DiMarco. 1995. HealthDoc: Customizing patient information and health education by medical condition and personal characteristics. In *AI in Patient Education Workshop*, Glasgow, Scotland, August.

Eduard H. Hovy. 1988. Generating language with a phrasal lexicon. In D. D. McDonald and L. Bolc, editors, *Natural Language Generation Systems*, pages 353–384. Springer-Verlag, New York.

W. Lewis Johnson and Ali Erdem. 1995. Interactive explanation of software systems. In *Proceedings of the Tenth Knowledge-Based Software Engineering Conference (KBSE-95)*, pages 155–164, Boston, Mass.

Alistair Knott, Chris Mellish, Jon Oberlander, and Michael O'Donnell. 1996. Sources of flexibility in dynamic hypertext generation. In *Proceedings of the Eighth International Natural Language Generation Workshop (INLG-96)*, pages 151–160, Herstmonceux Castle, Sussex, UK.

D. E. Knuth, editor. 1992. *Literate Programming.* CSLI.

Tanya Korelsky, Daryl McCullough, and Owen Rambow. 1993. Knowledge requirements for the automatic generation of project management reports. In *Proceedings of the Eighth Knowledge-Based Software Engineering Conference (KBSE-93*, pages 2–9, Chicago, Illinois.

Susan Korgen. 1996. Object-oriented, single-source, online documents that update themselves. In *Proceedings of The 14th Annual International Conference on Computer Documentation (SIGDOC-96)*, pages 229–238.

K. Kukich, K. McKeown, J. Shaw, J. Robin, N. Morgan, and J. Phillips. 1994. User-needs analysis and design methodology for an automated document generator. In A. Zampolli, N. Calzolari, and M. Palmer, editors, *Current Issues in Computational Linguistics: In Honour of Don Walker.* Kluwer Academic Press, Boston.

Karen Kukich. 1983. Design of a knowledge-based report generator. In *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics*, Cambridge, Mass.

James C. Lester and Bruce W. Porter. 1996. Scaling up explanation generation: Large-scale knowledge bases and empirical studies. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-96)*, Portland, Oregon.

Maria Milosavljevic, Adrian Tulloch, and Robert Dale. 1996. Text generation in a dynamic hypertext environment. In *Proceedings of the 19th Australasian Computer Science Conference*, pages 229–238, Melbourne, Australia.

Johanna Moore. 1995. *Participating in Explanatory Dialogues.* MIT Press.

Cecile Paris and Keith Vander Linden. 1996. Drafter: An interactive support tool for writing. *IEEE Computer, Special Issue on Interactive Natural Language Processing*, July.

Cecile Paris. 1988. Tailoring object descriptions to the user's level of expertise. *Computational Linguistics*, 11(3):64–78.

Michael Priestley, Luc Chamberland, and Julian Jones. 1996. Rethinking the reference manual: Using database technology on the www to provide complete, high-volume reference information without overwhelming your readers. In *Proceedings of The 14th Annual International Conference on Computer Documentation (SIGDOC-96)*, pages 23–28.

Owen Rambow and Tanya Korelsky. 1992. Applied text generation. In *Third Conference on Applied Natural Language Processing*, pages 40–47, Trento, Italy.

Ehud Reiter and Chris Mellish. 1993. Optimizing the costs and benefits of natural language generation. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI-93)*, volume 2, pages 1164–1169.

Jacques Robin. 1994. *Revision-Based Generation of Natural Language Summaries Providing Historical Background.* Ph.D. thesis, Columbia University.

Penelope Sibun. 1992. Generating text without trees. *Computational Intelligence*, 8(1):102–22.

Andreas Stolcke. 1996. Cluster 2.9. URL gopher://gopher.icsi.berkeley.edu/1/usr/local/ftp/ai/stolcke/software/cluster-2_9_tar.Z.

Leo Wanner and Eduard Hovy. 1996. The HealthDoc sentence planner. In *Proceedings of the Eighth International Natural Language Generation Workshop (INLG-96)*, pages 1–10, Herstmonceux Castle, Sussex, UK.

Michael White. 1997. Exemplars for Java: a lightweight framework for building object-oriented text planners in Java. Technical report, CoGenTex, Inc.