SOFTWARE REVIEWS

Editor: Paul Oman
Computer Science Dept.
University of Idaho
c/o 1350 NW 19th St.
Corvallis, OR 97330
Compmail+: p.oman

Software Reviews evaluates operating systems, applications, and utility software in widespread use throughout industry and academia. It points out the benefits and shortcomings of available software products, and contrasts the products with competitors. The products are critically examined as is — reviews are not influenced by promises of upgrades, enhancements, and fixes. Our goal is to give you a fair review of software on hand written by a reviewer with experience in the problem and application domains. We welcome your reviews and comments.

Desktop? Yes. Publishing? Not quite.

Galen Gruman, Assistant Editor

Now that desktop publishing has come out of the Apple Macintosh closet and into the world of MS-DOS, it is being touted as *the* big thing in personal computing. This technology eclipsed everything at Comdex last November and computing magazines have given it saturated coverage.

Everyone, it seems, wants to get on board. Several colleagues who work in diverse places — a CAD software firm, a government agency, a manufacturing firm, and a publicity office — report that their firms are committing to desktop publishing, sight unseen. They are asking not "*Should* we explore this technology?" but "*Which* program should we buy? Should we get an IBM or a Mac?"

I spent nearly a month experimenting with the three top MS-DOS programs, Pagemaker, Ventura Publisher, and Harvard Professional Publisher. I used a Leading Edge Model D with Hercules-compatible graphics, 640K bytes of RAM, a 20M-byte hard disk, one 360K-byte floppy drive, Leading Edge amber monitor, an Apple Laserwriter, and a mouse.

Criteria. What should a desktop publishing program do? It should:

- Lay out text in a rich variety of fonts, sizes, and widths on several page sizes accurately — what I see had better be what I get.
- Hyphenate and justify text accurately.
- Insert, move, size, and crop art elements.
- Wrap and flow text.
- Handle mundane details like page numbers (folios) and assure proper margins.
- Accept formatting from my word processor.
- Edit text and back-annotate so my master version is updated.
- Quickly move, delete, and add elements. I want the freedom to experiment, to change my mind, to try out ideas.
- Be easier or cheaper or higher in quality than my current method. It should do at least one task so much better (and the others no worse) that the expense (in money and time) is worth the effort.

I used the article by Rudolph Seviora on p. 20 as my test file. This is a good benchmark because it is straightforward, free from obscure math symbols, and uses basic magazine elements like a headline, deck, large initial capital letter (drop cap), and subheads. *IEEE Software* uses basic fonts: English Times (similar to Palatino), Souvenir, and Helios (a Helvetica clone).

Pagemaker

After six months of hoopla and a two-month delay in shipping, I was disappointed with Aldus Corp.'s Pagemaker. It has many nice functions, but trips over its abilities. The more exotic functions make the straightforward functions harder to use.

Layout. I had a tough time getting columns to align on the top and bottom because our subheads (like most) are a different size than our text. That throws off the depth of the column by a few points. When this happens, we cut the type above the subhead and manually add or delete space between paragraphs. It takes a few seconds.

Pagemaker's manual suggests you change the space between lines of type (the leading). That's bad design, because each column's leading will be different. If you store the text under each subhead as if it was a separate story, you'll lose the advantage offered by automatic flow (you might get gaps between sections) *and* you'll have to manually move the sections — as you do on paper — when text wraps.

An option not suggested by the manual is to change the leading on the line immediately before the subhead by the number of points needed to even out the column. That means you have to print out the page, measure the point difference, and then change the leading. That gets real tough when you want to dis-

IN BRIEF

Aldus recommends that you run Pagemaker on an IBM PC AT or true compatible. Xerox recommends an XT-class or AT-class machine for Ventura Publisher, but prefers that you use an AT-class computer. Harvard Professional Publisher requires that you use an XT-class machine. All require a hard disk (10M bytes minimum, but 30M bytes is better), 640K bytes of RAM, and a graphics card (Hercules monochrome or Enhanced Graphics Adapter color).

Pagemaker requires a mouse, Ventura will work without one but should be equipped with one, and Harvard almost works better without one. Pagemaker requires the Microsoft Windows operating environment ($99); Ventura requires the Gem environment ($299), but the minimum subset of Gem needed to run Ventura comes with the program.

For more information, circle the reader service numbers (from the list below) from the card at the back of the magazine.

Harvard Professional Publisher, $695, RS 11
Pagemaker, $695, RS 12
Ventura Publisher, $895, RS 13

tribute that space among the column's paragraphs. And if subsequent changes make the paragraphs wrap to another column, you'll have to start over.

Also, Pagemaker will overprint text (as the manual admits) from one text block onto a second if the second text block was created after the first. I consider this a software failure that needs more than a warning in a manual. Chances are that you will add elements from front to back (as the manual suggests you do). So when you lengthen elements early in the layout you will print over later elements. No thanks.

The large initial caps (drop caps) that lead our articles are very tough to do: The drop cap, the text it rests in, and the subsequent text must be separate elements. This is not only time-consuming, it also means you must manually align the three blocks so your leading isn't inconsistent. Why not do as Ventura does and treat drop caps as an attribute (like italics) rather than as a collection of elements? Or invoke a macro as we do in typesetting?

Typesetting. OK, so the layout task isn't easier than the traditional method. What about the output? Does it come close to typesetting? Pagemaker fares better here, once you override the defaults.

Although its hyphenation and justification work very well, the default parameters for spacing result in airy type. So I adjusted the letterspace and wordspace settings to simulate what our typesetter produces.

The default for type size and leading are also too big. The default 12-point type is pica typewriter size — much bigger than you'll see in magazines. Use nine or 10 points. The default for leading is 120 percent of point size, which means your leading gets proportionally larger as your type size increases. But leading is not usually proportional. One point lead for text below nine points and two points for text between nine and 14 points is more appropriate. For short elements like headlines and decks, the leading should generally be one point, no matter the type size, to keep the elements together.

Special characters are difficult because most can't be imported from word processors (page-layout programs don't use IBM's extended ASCII set). You could insert special codes in your text as place-holders, but because there's no search and replace, you have to go through the entire document, find them, and change them to the Pagemaker codes.

Pagemaker comes equipped with the nine fonts and two symbol sets built into the Apple Laserwriter Plus. However, if your printer does not support those fonts, you won't get a message telling you so until you try to print. At that point, you will be prompted to let Pagemaker recompose your document with the available fonts and sizes.

The two symbol sets are confusing because what you see is not what you get. You may select a symbol font, but you'll see normal alphanumeric characters on the screen. The printout will have the symbols.

Interface. The MS-DOS version's user interface wasn't bad and I found no significant differences from the Macintosh version. The screen resolution seemed to be as good as the Mac as well.

But the program is slow, probably because it displays in painstaking detail what you'll see on your printout. Because of Pagemaker's snail pace, I had many long waits with my finger depressed on the mouse button. More memory may have helped, but an option to use generic letters rather than the detailed font reproductions would give even more speedup.

Pagemaker is slow, probably because it displays in painstaking detail what you'll see on your printout. I had many long waits with my finger depressed on the mouse button.

It would have helped a lot to use wait icons more. They appeared rarely, and if it weren't for the flashing disk-drive light, I wouldn't have known whether I was getting no response because the program was working on something else or because my mouse was disconnected, or because of something else altogether.

Moving large text (bigger than 12 points) often left pieces of the text in its original position. I had to zoom out and back in so the screen would repaint correctly.

After the slowness, the greatest annoyance was the availability of menu options. In this Mac-like interface, active options are supposed to be in black, while inactive ones are supposed to be in gray. So, while in layout mode, I selected a block of text to change its type specs and selected the type option, made my changes, and clicked OK. Nothing happened. I finally realized I had to switch modes (to text mode). The program should not have displayed in black (meaning available) the type options when they weren't available (when I wasn't in text mode).

I was not pleased that some of the fonts that came with Pagemaker could *not* be displayed on the screen, forcing Windows to substitute its generic font (a nice Windows feature).

There is no back-annotation feature to write text changes in Pagemaker back to disk in your original word-processor format, and that's a void that should be filled.

The documentation is superficial. For example, I was surprised to find that something as elementary as "symbol" was not indexed in the manual, yet it was in the on-line help file. The technical-support people were helpful.

The manual also fails to mention anything about connecting to laser printers, information that is sorely needed when hooking an MS-DOS computer to an Apple printer. Another product's manual revealed that you can use the same serial cable you'd use to connect to an HP Laserjet, which makes sense because both printers use Canon engines.

Amid these deficiencies was one pleasant extra: Pagemaker used the left button as the command button and the right button as a toggle between zoom modes — a nice feature Ventura would do well to copy.

Ventura

Xerox's Ventura is fast, and its interface (based on Digital Research's Gem environment) is a bit easier to use than Pagemaker's. The speed seems to have come at the expense of resolution, font emulation, and other options like the number of available fonts and sizes.

It is also more capable in many areas. It will layout an entire document for you at once, automatically numbering pages and placing headers and footers (grunt work!).

I was pleased at how easy it is to insert and remove pages. That means if a layout changes because of a late ad that goes where the first page is, I don't have to move each page. All I have to do is insert the page for the ad. Of course, I'll have to reposition graphics, but I'd have to do that if I was working on paper.

Ventura uses the concept of frames for its layout, meaning that you define the borders for elements such as headlines, decks, body copy, and byline. You can move and scale these frames, but moving text among them can be difficult. When you import text, make sure the frame's margins are zeroed out, or

you'll have unwanted margins within your frame.

Another drawback is that you can't define your own page size. I like using paper larger than the final printed version so I can put in trim marks and notes to the printer. (Layout boards are oversized for this reason.) Ventura only allows standard photocopy sizes (like 8½"×11").

And while you can display pages side by side (a spread), you can't work with both pages at once like you do with art boards. That can be a problem when a graphic (like a bar or box) runs across a spread.

Typesetting. Again, I found the default spacing to be too airy, and I couldn't adjust it as well as I could Pagemaker's. And again, the default point size and font (12-point Courier) is inappropriate.

The font selection at first appears to be abysmal. Only the default Postscript symbol set and fonts are available: Times, Helvetica, and Courier. But the font file that comes with Ventura has the other six fonts and the symbol set that come with the Laserwriter Plus — they're just commented out. You can uncomment them and add any point size you want, but only if the total number of fonts is eight or less and the total number of sizes selected is 100 or less.

The separation of point size and leading specifications also confused me. The two are traditionally thought of as one parameter. When you spec type, you write something like "9/10 ET × 13ΔRR" to mean "nine points with 10 points leading in 13-pica, ragged right columns."

While I found Ventura easier to use overall than Pagemaker, its structure took some time to get used to, so my first sample layout actually took longer on Ventura than on Pagemaker. But once I got the hang of it, I could work faster.

Ventura back-annotates files, so changes in the text made while you do layout are stored in the original word processor file. The back-annotation is automatic, even if you don't want all changes saved and, at least for Microsoft Word, it replaced font and point-size attributes with codes (<P9M>). Unfortunately, it removed the formatting from the Word file so when I subsequently printed from Word to a laser printer, I got the default — the unwanted 12-point Courier.

Also, I wrongly assumed that it would back-annotate to a new file name. Each time you load the layout, Ventura reloads the file that reflects the latest changes to that file — whether you want it to or not. Knowing this, you can simply copy the file and retain an original version — gaining the benefits of back annotation without the side effects that happened to my Word file.

Overall, I'd choose Ventura over the other programs because its speed and relative ease help counter its limitations.

You can create several combinations of these available-font files so you can change your mind while laying out a document. However, you have to leave Ventura and run a utility to create the files.

Interface. The few toolbox (icon menu) choices made using Ventura easier, but the organization of some functions threw me.

For example, Ventura uses easy-to-create tags to label text types. Any change in point size or justification to text will be applied to all text with the same tag. But it took me some time to figure out such a global change was handled in paragraph mode, not in page or text mode where I expected. I should have looked in the manual.

These tags and the ability to store style sheets make layout of similar documents much easier.

Ventura does lack some niceties: There are no guides on the rulers that follow your cursor so you can more accurately place text and graphics. (This is promised for a future version.) And the option to change from inches to picas or points only affects the ruler: All the fields still use inches, an unnatural measurement for most typesetting functions.

The manual had some bright spots, like cabling instructions for most popular printers and detailed information on downloading fonts.

Overall, I'd choose this program over the other page-layout programs because its speed and relative ease help counter its limitations. I would use it to do simple jobs that I might now do with Microsoft Word's formatting abilities.

Harvard

Software Publishing's Harvard Professional Publisher has some really neat features, but it requires a lot of brute force and patience to make the most of them.

This is especially unfortunate because it has some capabilities not found in the powerhouse Pagemaker and Ventura programs. This is also surprising because Harvard was cowritten by Bestinfo, creators of the $7000, PC-based Superpage multitypesetter front end and layout system. Indeed, Harvard is a low-end front end to Superpage.

Layout. Because it was designed with a single-page orientation and tagged-text structure similar to Ventura, I thought Harvard would be easy to learn. It's not. While I was able to learn Pagemaker and Ventura without referring to the manuals often, I decided to use the Harvard manual after a day knocking my head against the wall. It's a must. (I did not try the tutorial because I prefer the random access and quick jumps possible in reading manuals.)

Once I followed the layout setup procedure in the manual, I started getting somewhere. But not too far. The program works like a series of linked programs, so when you switch applications, you don't always return to where you left off. For example, if I left the third page of a layout to change my font selection, I would not return to p. 3 in my layout — I'd end up in the main menu and have to return to layout mode and then go to p. 3.

Unlike the other programs, Harvard has no default template, so you can't just start experimenting. You have to know what you want beforehand (whether you really do know or not). Also, you can't work with spreads — you can't even display pages side by side.

But once you begin layout, you discover two very neat features: polygonal wraps and keepout areas (masks) are supported without breaking your text into several pieces. Unlike Ventura and Pagemaker, which require you to define where text exists, Harvard lets you define the text areas and then put exceptions (the keepouts and wraps) in those areas. Rather than cut and paste text areas to wrap around pictures or to insert decks, you just define a keepout and the text flows around it. You can even put special text in the keepout.

However, its column alignment feature assumes that every page, including the last one, must align at the top and bottom. You cannot select a range of pages (the last page often should end naturally, something the program

should be able to deal with. But Harvard was the only program to even address the issue — Pagemaker and Ventura don't.

Typesetting. Harvard supports only ASCII and DCA files (produced by low-end word processors like IBM's Writing Assistant). I think any page-layout program that doesn't support Microsoft Word, Word Perfect, and Wordstar (and probably XyWrite) has missed the boat. (Luckily, my Microsoft Word can create ASCII and DCA files.)

And while Harvard can handle the nine Apple Laserwriter Plus fonts and two symbol sets, it can handle only two fonts per document! (The two fonts can have two faces — roman and italic — and two weights — medium and bold.) That means even a font-restrained magazine like *IEEE Software* can't be completely done on the program. (We use three fonts.) Anything done on it would have to be very conservative (like the gray Harvard manual, produced on the Harvard system).

Unlike in Ventura, you can't expand the few point-size choices or adjust linespacing, wordspacing, and letterspacing.

Despite these omissions, Harvard's typesetting approach was a pleasure. For example, the default leading for all point sizes is two points — a traditional default in typesetting, and far better than the leadings assigned by Pagemaker and Ventura.

Also, Harvard defines paragraph indents by number of ens (a proportional measure) and defaults to two ens (an em, the standard paragraph indent). Pagemaker and Ventura force you to use fixed-width indents. The program recognizes the difference between indent-hang paragraphs and drop indents (used in bulleted lists and reference lists). Pagemaker and Ventura don't.

Finally, Harvard lets you condense text to one of two percentages. While a digital typesetter can condense and expand (and slant) type at almost any increment, Pagemaker and Ventura don't support this feature (called scaling) at all, so even two settings is nice.

Interface. Harvard requires a lot of preparation before you can get to the layout, but you can store the formats to reduce the preparation time for future documents. Harvard also requires a lot of coding in your word processor. Like Ventura, it automatically back-annotates the text files but, unless your original is an ASCII file, the back-annotation is made to your translated import file, leaving the original untouched.

PC-based typesetting

One option to reduce typesetting costs is to do as much of the typesetting coding before the text ever gets to a typesetter. *IEEE Software* has cut typesetting bills in half by coding articles on our word processor and running them through a preprocessor to create codes for our typesetter's Compugraphic machine.

G.O. Graphics offers a program that supports true typesetting on a PC XT or clone with at least 512K bytes of RAM and a graphics card. The Deskset program emulates Compugraphic typesetters and outputs in Postscript format (for laser printers and Linotype typesetters). A promised new version will also write to floppies in Compugraphic format. A preview mode lets you see on the screen what your codes will do.

This program is *not* for people who are unfamiliar with typesetting. It is almost indistinguishable from a real typesetter. The only difference I found is the hardware: The commands have been moved around to adjust to the PC keyboard.

Its commands match the Compugraphic MCS commands on a digital typesetter. The composing functions and display are the same. The intolerance for coding errors is the same. But it was a joy to work with a program that delivered what it promises, true PC-based typesetting.

Deskset gives you full control over kerning, point size (from four points to 127.5 points in half-point increments), leading, multicolumn tabs, number of fonts, font expansion and compression, font slanting (to the left and to the right), justification, hyphenation rules, width tables, and indents. You see hyphenation and line breaks.

A document can have as many as 16 active fonts, selected from a library of 256. The nine standard Apple Laserwriter Plus fonts and two symbol sets come with the program. If you don't like your current font selection, you can add and switch fonts without leaving your document or losing your current font selection.

You can get a length count for the text. And you can import graphics from Dr. Halo, Lotus 1-2-3, and other graphics and graphics-producing programs.

The people who will get the most from this program are those who already have a Linotype or Compugraphic typesetter. They can transfer to a PC the work now causing bottlenecks at the typesetting terminal (which cost about $40,000).

They know how typesetting works, so they won't have the adaptation problems a typical PC editor or designer would have. People very familiar with Troff and other code-intensive output systems might also adapt easily.

You might also define a simplified, macro-like coding system to use while editing and run the files through a preprocessor of your own before calling the files into the program for cleanup and preview. The files must be in ASCII, but you can use the extended ASCII set to put in Deskset's command codes (Alt-221 and Alt-222 in a file will be read as the command-on and command-off angle brackets seen in Deskset).

Deskset is not cheap — $995 — and it uses a hardware lock at the LPT1 parallel port to prevent the nonprotected program from being used at more than one station. And the price you pay for its capability is that you really must become a typesetter. **Reader Service Number 14**

Figure A. This multicolumn output on a 300-dpi laser printer shows Deskset's range. The Helvetica test is four-point type and the large Times *g* is 127-point type. The text at right shows various character attributes.

This coding-intensive approach lets you handle typesetting features like long dashes and quotes. But it either forces editors to do a lot of formatting or designers to do a lot of editing, inviting undesirable job crossover.

You can edit the format codes in Harvard, but the editor is slow and the display difficult to read. Other layout programs use a background on their pages, so rules and text appear in black. Not Harvard. Its rules are in the screen color — and barely visible. The black background is not as easy on the eyes as white ones.

Harvard doesn't use an operating environment like Gem or Windows. Communication with the mouse is slow, forcing you to drag the mouse slowly and to press the mouse buttons for several seconds. Except for moving around the image, I found it a lot faster and more exact to use function keys and the cursor.

The manual is insufficient in several areas. For example, after spending hours trying to figure why the ASCII files I imported wouldn't show up in the document, I called customer service. It turns out that there's a difference between *importing* and *loading* files that was not mentioned. You import only those ASCII files you don't want any formatting done to (like tables). You load all other ASCII files in layout mode by entering the file name (which must have the extension .TXT).

Another example is the font-installation procedure. The default is to install only Times and Helvetica, yet the Harvard menu lists all 11 Laserwriter Plus fonts. When you try to select fonts from the Harvard menu, it can find only Times and Helvetica.

I thought I had made a mistake when installing the program, so I reinstalled it, paying attention to the fonts loaded. Still only Times and Helvetica. I listed an install-disk directory and saw the other nine fonts (like ZAPF.BAT and PALATINO.BAT), but the manual should have told me what the default was, and the menu shouldn't list non-loaded fonts. When I printed these Laserwriter Plus fonts on my Laserwriter, I got good old Courier instead of a message saying the fonts I selected were unavailable for my printer.

Overall impressions

Promises. The promotional material promises reduced costs in typesetting and the elimination of pasteup artists and designers. "Where you once needed a typesetter, designer, and pasteup artist, you can do it all yourself," reads the promotional material for Pagemaker.

Yes, you can do it all yourself — sort of. You can transfer the burden of three jobs to your desktop, but you'll not only lose your sanity, you'll sacrifice quality.

None of these programs beats a good typesetter, designer, and pasteup artist — in quality, time, or cost. Doing it all yourself invites bad design if you're not a designer, bad typography if you're not a typesetter, and bad execution if you're not a pasteup artist.

Integrating graphics with type is another selling point for these programs. I only cursorily checked these functions for two reasons. First, I do not believe the output resolution meets magazine standards. For example, the gray bars I printed had a resolution of about 55 dots per inch, while newspaper standard is 75 dpi and the magazine standard 133 dpi. The laser printers can output 300 dpi, but the page-layout screens don't take advantage of it. Second, text is (to me) what counts most. Being able to put shapes in my layout representing my graphics is sufficient, and the programs did this easily.

You can transfer the burden of three jobs to your desktop, but you'll not only lose your sanity, you'll sacrifice quality.

After reading all the hype, I was disappointed with all three programs. Unfortunately, I spent most of my time battling limitations and figuring out how to make the pieces work. I felt I was debugging, not using, the software.

I think these programs make the same mistake integrated software like Framework did: They try to put together a "total" package, a package composed of a couple outstanding elements and several average or substandard elements. Each program had major strengths, but the strengths were always undone by the many weaknesses.

Layout. Figure 1 shows the first page of the Seviora article as produced by each program. I was very disappointed that these page-layout programs laid out pages so poorly. Specifically, they make a natural design approach, experimentation, difficult: It is very difficult to move elements around and you are penalized if you change your mind. Only flowing text was automatic and simple.

The manuals recognize this lack of flexibility, advising that you delete text and load it into a new position rather than moving it. And the programs' menu structures reinforce this.

The programs don't simplify and automate the grunt work (like balancing columns, adding space between paragraphs, and handling folios), they merely transfer the burden from paper to screen. While some programs handle some of these needs, overall the tasks remain essentially the same.

For example, to make sure tops and bottoms of columns align, we add space between paragraphs and before headlines (our pasteup artist does it with a knife). I wanted the page-layout program to do this, not me. What I got was "vertical justification," which changes the spacing between lines within each column, making the leading inconsistent. Only Harvard handles this problem for you.

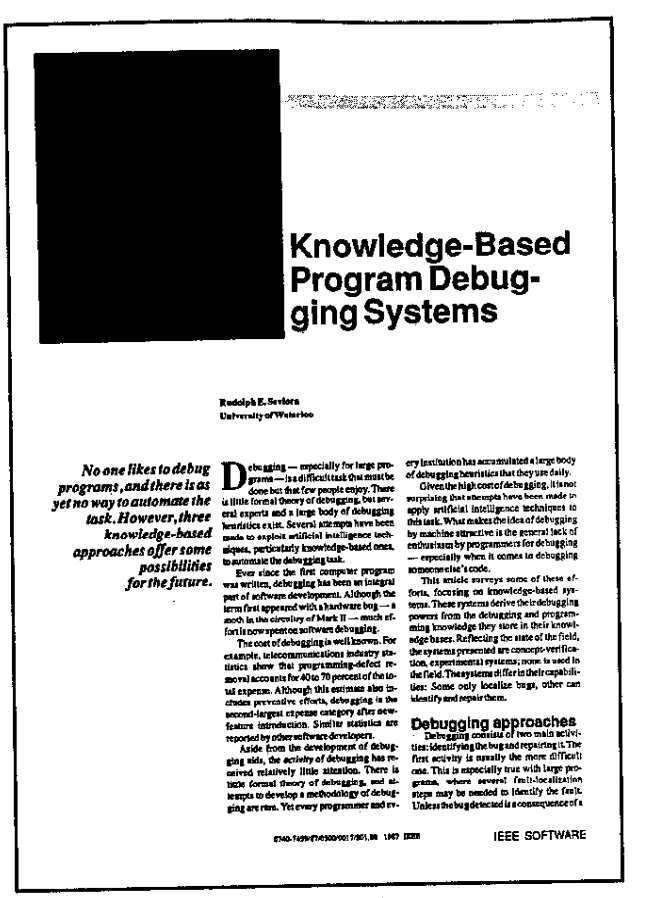
Adding and removing text can also be very difficult. The programs recommend you store the elements of an article (the body, headline, and decks) in separate files and load them individually. That makes for a management headache (and unnatural work flow) at the editor's end.

Typesetting. The typesetting features were better, but for the most part are only equal to the analog phototypsetting systems replaced a decade ago by chemical burn-in digital systems. The font selection is limited, the characters ill-formed (they don't always align along the baseline), and the 300-dpi resolution of laser printers is unacceptable for professional publications.

You can output to a Linotype typesetter that has the $16,000 Postscript interface, but you won't have access to the Linotype's fonts and capabilities because these programs send a bitmap image of the pages they create (including the fonts, graphics, and spacing) — otherwise, they couldn't accurately show line breaks and handle hyphenation. However, you can get 1200-dpi output with a Linotype Postscript interface.

Postscript supports 35 fonts, including four typewriter fonts, two symbol sets, and one music set. Helvetica, Times, Courier, and one symbols set come in ROM with each Postscript output device. Some devices come with more fonts in ROM. Downloadable fonts generally cost $185 each.

I was impressed with the hyphenation, but not with the default parameters for spacing and leading. Also puzzling



Knowledge-Based Program Debugging Systems

Rudolph E. Seviora
University of Waterloo

No one likes to debug programs, and there is as yet no way to automate the task. However, three knowledge-based approaches offer some possibilities for the future.

Debugging — especially for large programs — is a difficult task that must be done but that few people enjoy. There is little formal theory of debugging, but several experts and a large body of debugging heuristics exist. Several attempts have been made to exploit artificial intelligence techniques, particularly knowledge-based ones, to automate the debugging task.

Ever since the first computer program was written, debugging has been an integral part of software development. Although the term first appeared with a hardware bug — a moth in the circuitry of Mark II — much effort is now spent on software debugging.

The cost of debugging is well known. For example, telecommunications industry statistics show that programming-defect removal accounts for 40 to 70 percent of the total expense. Although this estimate also includes preventive efforts, debugging is the second-largest expense category after new-feature introduction. Similar statistics are reported by other software developers.

Aside from the development of debugging aids, the activity of debugging has received relatively little attention. There is little formal theory of debugging, and attempts to develop a methodology of debugging are rare. Yet every programmer and every institution has accumulated a large body of debugging heuristics that they use daily.

Given the high cost of debugging, it is not surprising that attempts have been made to apply artificial intelligence techniques to this task. What makes the idea of debugging by machine attractive is the general lack of enthusiasm by programmers for debugging — especially when it comes to debugging someone else's code.

This article surveys some of these efforts, focusing on knowledge-based systems. These systems derive their debugging powers from the debugging and programming knowledge they store in their knowledge bases. Reflecting the state of the field, the systems presented are concept-verification, experimental systems; none is used in the field. The systems differ in their capabilities: Some only localize bugs, other can identify and repair them.

Debugging approaches

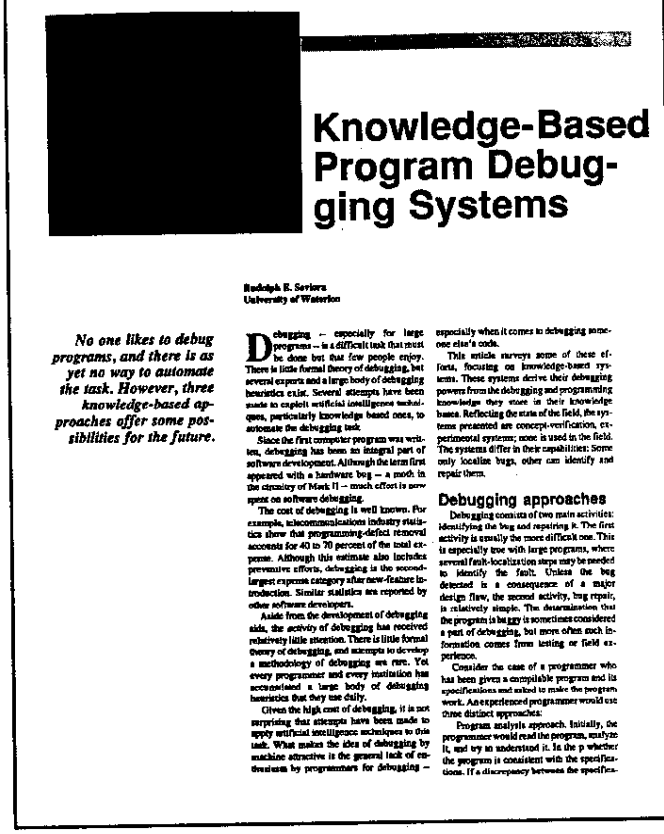
Debugging consists of two main activities: identifying the bug and repairing it. The first activity is usually the more difficult one. This is especially true with large programs, where several fault-localization steps may be needed to identify the fault. Unless the bug detected is a consequence of a major design flaw, the second activity, bug repair, is relatively simple. The determination that the program is buggy is sometimes considered a part of debugging, but more often such information comes from testing or field experience.

Consider the case of a programmer who has been given a compilable program and its specifications and asked to make the program work. An experienced programmer would use three distinct approaches:

Program analysis approach. Initially, the programmer would read the program, analyze it, and try to understand it. In the p whether the program is consistent with the specifica-

0740-7459/87/0500/0001 1987 IEEE

IEEE SOFTWARE



Knowledge-Based Program Debugging Systems

Rudolph E. Seviora
University of Waterloo

No one likes to debug programs, and there is as yet no way to automate the task. However, three knowledge-based approaches offer some possibilities for the future.

Debugging — especially for large programs — is a difficult task that must be done but that few people enjoy. There is little formal theory of debugging, but several experts and a large body of debugging heuristics exist. Several attempts have been made to exploit artificial intelligence techniques, particularly knowledge based ones, to automate the debugging task.

Since the first computer program was written, debugging has been an integral part of software development. Although the term first appeared with a hardware bug — a moth in the circuitry of Mark II — much effort is now spent on software debugging.

The cost of debugging is well known. For example, telecommunications industry statistics show that programming-defect removal accounts for 40 to 70 percent of the total expense. Although this estimate also includes preventive efforts, debugging is the second-largest expense category after new-feature introduction. Similar statistics are reported by other software developers.

Aside from the development of debugging aids, the activity of debugging has received relatively little attention. There is little formal theory of debugging, and attempts to develop a methodology of debugging are rare. Yet every programmer and every institution has accumulated a large body of debugging heuristics that they use daily.

Given the high cost of debugging, it is not surprising that attempts have been made to apply artificial intelligence techniques to this task. What makes the idea of debugging by machine attractive is the general lack of enthusiasm by programmers for debugging — especially when it comes to debugging someone else's code.

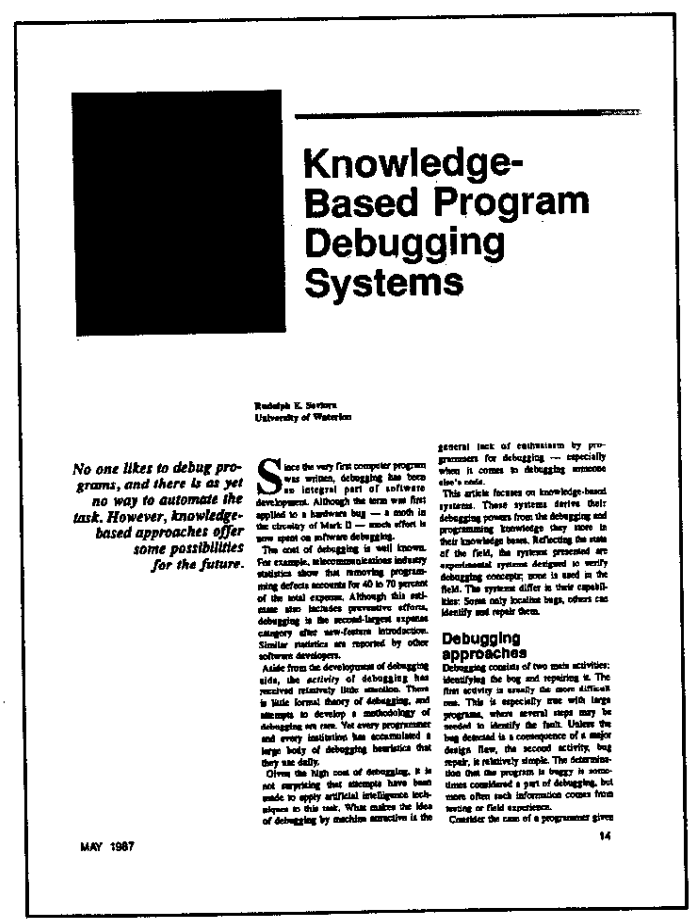
This article surveys some of these efforts, focusing on knowledge-based systems. These systems derive their debugging powers from the debugging and programming knowledge they store in their knowledge bases. Reflecting the state of the field, the systems presented are concept-verification, experimental systems; none is used in the field. The systems differ in their capabilities: Some only localize bugs, other can identify and repair them.

Debugging approaches

Debugging consists of two main activities: identifying the bug and repairing it. The first activity is usually the more difficult one. This is especially true with large programs, where several fault-localization steps may be needed to identify the fault. Unless the bug detected is a consequence of a major design flaw, the second activity, bug repair, is relatively simple. The determination that the program is buggy is sometimes considered a part of debugging, but more often such information comes from testing or field experience.

Consider the case of a programmer given a compilable program and its specifications and asked to make the program work. An experienced programmer would use three distinct approaches:

Program analysis approach. Initially, the programmer would read the program, analyze it, and try to understand it. In the p whether the program is consistent with the specifications. If a discrepancy between the specifica-



Knowledge-Based Program Debugging Systems

Rudolph E. Seviora
University of Waterloo

No one likes to debug programs, and there is as yet no way to automate the task. However, knowledge-based approaches offer some possibilities for the future.

Since the very first computer program was written, debugging has been an integral part of software development. Although the term was first applied to a hardware bug — a moth in the circuitry of Mark II — much effort is now spent on software debugging.

The cost of debugging is well known. For example, telecommunications industry statistics show that removing programming defects accounts for 40 to 70 percent of the total expense. Although this estimate also includes preventive efforts, debugging is the second-largest expense category after new-features introduction. Similar statistics are reported by other software developers.

Aside from the development of debugging aids, the *activity* of debugging has received relatively little attention. There is little formal theory of debugging, and attempts to develop a methodology of debugging are rare. Yet every programmer and every institution has accumulated a large body of debugging heuristics that they use daily.

Given the high cost of debugging, it is not surprising that attempts have been made to apply artificial intelligence techniques to this task. What makes the idea of debugging by machine attractive is the general lack of enthusiasm by programmers for debugging — especially when it comes to debugging someone else's code.

This article focuses on knowledge-based systems. These systems derive their debugging powers from the debugging and programming knowledge they store in their knowledge bases. Reflecting the state of the field, the systems presented are experimental systems designed to verify debugging concepts; none is used in the field. The systems differ in their capabilities: Some only localize bugs, others can identify and repair them.

Debugging approaches

Debugging consists of two main activities: identifying the bug and repairing it. The first activity is usually the more difficult one. This is especially true with large programs, where several steps may be needed to identify the fault. Unless the bug detected is a consequence of a major design flaw, the second activity, bug repair, is relatively simple. The determination that the program is buggy is sometimes considered a part of debugging, but more often such information comes from testing or field experience.

Consider the case of a programmer given

MAY 1987

14

Figure 1. (a) Pagemaker layout, (b) Ventura layout, and (c) Harvard layout. Column alignment was one of the more difficult problems for all three programs; only Harvard tackled the problem. The Ventura headline type runs off the page because the black box to its left could not be moved, deleted, or resized to make room for the text (this was the only time I had difficulty with Ventura graphics).

Technical-document publishing

Denice Killian, American Honda

Every technical writer here has a PC, and we feel that desktop publishing can speed the work flow between the writing staff and the production staff. That faster turnaround is the key to technical publications: We want to get the information out to the field as quickly as possible.

Desktop publishing programs claim to offer faster turnaround on technical publications by letting us dump our technical writing (done on the XyWrite III word processor) into a predefined page make-up format. Such templates should save us a lot of time preparing the final publications.

The desktop publishing programs seem able to meet their time-savings claims for simple jobs. Work that uses standard templates appears to be done more easily in page-layout programs than on typesetting stations because the coding is predefined — we don't have to rekey the same codes for similar jobs. We could have the writers use formatting codes instead of us doing it, but that would require training and checking for errors.

Technical bulletins are usually good candidates for on-line layout and laser-printer output. They are usually only two pages long, and the 300-dpi resolution and the fonts available on laser printers are fine for this kind of internal documentation. (I bite my tongue when claiming this because my advertising background has me wanting everything from my department to have the best possible quality — and desktop publishing limits this quality.)

But desktop publishing is not applicable to all the work we do. Work that can't be handled well includes multiple-column parts lists with many horizontal and vertical rules, customer brochures, advertisements, newsletters, and publications with lots of graphics.

While we're still exploring our options, we seem to be headed toward desktop publishing on PCs for routine work (with output on both laser printers and Postscript-compatible typesetters) and toward continued use of traditional typesetting for complicated jobs.

Killian is production supervisor for service publications at Honda. She has 16 years of experience in design, typography, and production at newspapers, magazines, advertising agencies, and in-house publications.

was the default to a single-column, single-spaced, typewriter-font format. Desktop publishing is an expensive and time-consuming way to perform basic typing.

Pagemaker and Ventura could not handle drop indents, and both forced me to use a fixed indent for paragraphs (which I specified in picas or inches). Only Harvard handles drop indents and uses em spaces for paragraph indents.

Alternatives. I wouldn't recommend these systems to anyone planning to produce large jobs like magazines, books, or manuals. But what about internal documents — manuals, price sheets, newsletters — isn't that what these programs are aimed at?

Yes and no. The impressions of a technical-document production supervisor who is looking at desktop publishing are in the box above.

None of the programs was easier or faster than using my word processor to output the text to a laser printer and pasting up the columns manually. I was able to produce a 44-page novella — hyphenated, justified, with chapter headings and page numbering — with my word processor and a laser printer in about three hours.

Document processors like Microsoft Word, Lotus Manuscript, Word Perfect, and XyWrite III are all designed for editors who need to format documents easily. They're not WYSIWYG, but they produce what you need faster and more easily than the WYSIWYG page-layout programs I tested. Stand-alone systems like Interleaf also address the production of documentation and manuals.

Or you can do what we do: We edit with word processors, insert the typesetting commands, and transmit the file over phone lines to a typesetter that has a front-end translator. We've cut our typesetting costs in half simply by using this technology, which has been around for at least seven years. Many college newspapers use it routinely.

What we don't get now that desktop publishing offers is the chance to see the type before it is set in galleys. If we routinely edited at galley stage, the chance to preview type would be a big impetus to invest in these systems. But we don't edit on galleys, we do all editing and proofing on our PCs or on printouts. Layout and pasteup is where our magazine could use desktop help.

The bottom line. Of course, if you don't have a reliable, inexpensive typesetter (one you can transmit a file to), or production equipment, or money to hire a good designer, if your jobs are small (less than 16 pages) and the circulation small — then these programs might help. But they aren't ready to compete or even help with true publishing.

And that's the question to ask before you invest in a desktop publishing system. The software's price ($695 to $895) becomes a minor expense compared to the PC, the laser printer, the monitor and graphics card, plus the mouse.

Make sure you have a compatible word processor so your formatting is retained. These include Microsoft Word, Word Perfect, Multimate, and Wordstar on Pagemaker and Ventura, — plus Xywrite III and DCA-format editors (IBM Writing Assistant et al.) on Pagemaker. Otherwise, you'll have to create an ASCII-only file and do your formatting in the publishing program. Harvard accepts only ASCII and DCA files.

Another Computer Society magazine, *IEEE Expert*, produced 41 pages of its spring issue with Pagemaker on the Mac. While the typesetting costs decreased, the amount of overtime and the increased layout costs (because of the overtime) at least zeroed out that gain, Managing Editor Henry Ayling said. However, the magazine is still experimenting with the software, albeit on a much less ambitious scale.

Down the line. These programs may be like the early text editors. I suspect desktop publishing will evolve similarly over the next several years.

To do so, it must remove burdens, not duplicate them. It must be more flexible in layout, letting the designer experiment with placement easily. It should offer at least the same flexibility and range of modern typesetters, not duplicate decade-old technology. It must also decide who it's aiming for: Programs simultaneously aimed at four-page employee newsletters and 120-page magazines serve neither.

It is not enough to have the same graphics abilities on an MS-DOS computer that the Macintosh has always had. That's missing the point. Instead, these programs must meet the needs of the publishing application, regardless of the base machine.

For now, programs that address special needs, like typesetting front ends, document processors, and format translators, are better suited for publishing than the attempts to create a grand unified publisher. In five years, who knows?

No one likes to debug programs, and there is as yet no way to automate the task. However, three knowledge-based approaches offer some possibilities for the future.

Debugging — especially for large programs — is a difficult task that must be done but that few people enjoy. There is little formal theory of debugging, but several experts and a large body of debugging heuristics exist. Several attempts have been made to exploit artificial intelligence techniques, particularly knowledge-based ones, to automate the debugging task.

Ever since the first computer program was written, debugging has been an integral

ery institution has accumulated a large bc of debugging heuristics that they use dail

Given the high cost of debugging, it is surprising that attempts have been made apply artificial intelligence techniques this task. What makes the idea of debuggi by machine attractive is the general lack enthusiasm by programmers for debuggi — especially when it comes to debuggi someone else's code.

This article surveys some of these

No one likes to debug programs, and there is as yet no way to automate the task. However, three knowledge-based approaches offer some possibilities for the future.

Debugging -- especially for large programs -- is a difficult task that must be done but that few people enjoy. There is little formal theory of debugging, but several experts and a large body of debugging heuristics exist. Several attempts have been made to exploit artificial intelligence techniques, particularly knowledge based ones, to automate the debugging task

Since the first computer program was writ-

especially when it comes to debu one else's code.

This article surveys some forts, focusing on knowledge tems. These systems derive thei powers from the debugging and p knowledge they store in their bases. Reflecting the state of the f tems presented are concept-veri perimental systems; none is usec

No one likes to debug programs, and there is as yet no way to automate the task. However, knowledge-based approaches offer some possibilities for the future.

Since the very first computer program was written, debugging has been an integral part of software development. Although the term was first applied to a hardware bug — a moth in the circuitry of Mark II — much effort is now spent on software debugging.

The cost of debugging is well known. For example, telecommunications industry statistics show that removing program-

general lack of enthusiasm by grammers for debugging — espe when it comes to debugging son else's code.

This article focuses on knowledge-l systems. These systems derive debugging powers from the debuggin programming knowledge they sto their knowledge bases. Reflecting the of the field, the systems presente experimental systems designed to

No one likes to debug programs, and there is as yet no way to automate the task. However, knowledge-based approaches offer some possibilities for the future.

Since the very first computer program was written, debugging has been an integral part of software development. Although the term was first applied to a hardware bug — a moth in the circuitry of Mark II — much effort is now spent on software debugging.

The cost of debugging is well known. For example, telecommunications industry statistics show that removing programming defects accounts for 40 to 70 percent of the

attempts to develop a methodolo debugging are rare. Yet every progran and every institution has accumula large body of debugging heuristics they use daily.

Given the high cost of debugging not surprising that attempts have made to apply artificial intelligence niques to this task. What makes the id debugging by machine attractive i general lack of enthusiasm by prog

Figure 2. (a) Pagemaker type, (b) Ventura type, (c) Harvard type, and (d) Compugraphic typesetter type. Pagemaker's drop cap is badly positioned because I had to do it manually, but I was able to adjust the spacing to approximate a typesetter's. Ventura spacing was adjustable, but not to the same extent. The drop cap was positioned automatically. Harvard's spacing was not adjustable, and the drop cap positioning caused the line imbalance in the second column. The typeset copy's drop cap is placed with a macro.