Distilling Information from Text: The EDS *TemplateFiller* System

H. Kelly Shuldberg,* Melissa Macpherson, Pete Humphrey, and Jamii Corley
EDS Research, 5951 Jefferson Street NE, Albuquerque, NM 87109-3432

A system is described which digests large volumes of text, filtering out irrelevant articles and distilling the remainder into templates that represent information from the articles in simple slot/filler pairs. The system is highly modular in that it consists of a series of programs, each of which contributes information to the text to help in the final analysis of determining which strings constitute valid values for the slots in the template. This modular design has the dual advantage of allowing relatively easy debugging and of permitting many of the component programs to participate in other projects. The system is customized to specific domains, taking advantage of simple string matching techniques to improve the effectiveness of more complex sentence-level semantic processes. The extension to new domains has been facilitated by dividing system data files into generic vs. specific categories; domain extension requires the creation of only the domain-specific files.

Introduction

Among the most important challenges in information science is the efficient management of machine-readable text data. This basic problem breaks down into at least two major divisions: (1) providing ways to find specific information in very large but more or less static repositories of texts; and (2) effectively managing the enormous volume of new text that becomes available every day. In the first case, effective querying for retrieval of whole texts is typically the goal, and since repositories of text may have been assembled for diverse reasons and under different conditions, flexibility is one of the most important requirements. In the second case, it is often possible to assume that machine-readable text is being collected for a specific reason. The main goal then is just to reduce the volume sufficiently that an information consumer can get what he or she requires without having to wade through a huge amount of redundant or irrelevant text along with it.

The problem of the sheer volume of new information is especially acute in the computer technology domain, and here it is compounded by the additional factor of rapid obsolescence. The computer industry is one of the fastest-changing industries the world has ever witnessed. Rapid advances and innovation in computer technology mean that new computer products are constantly being introduced into the marketplace. Keeping abreast of new products is critical both to customers, who naturally want the most for their money, and vendors, who hope to outdistance the competition. Falling behind in knowledge of the most current technology available could mean a tremendous competitive disadvantage for any business that relies on computers for sales or support. Because of this, literally dozens of periodicals chronicle computer product innovations and releases, and the amount of new text generated in this area each week is significant. It is clear that an automatic method of distilling large quantities of this text into small amounts of information would be extremely valuable.

This paper describes the *TemplateFiller* system developed at EDS Research in Albuquerque, New Mexico. This system reads raw text input from computer industry journals and uses it to automatically produce bulletins on new computer products; these bulletins are then distributed to interested parties within EDS.

The remainder of the article is divided into four sections. "Why Templates?" examines the concept of text distillation, along with the conditions under which template filling is appropriate as a method. "Other Research in Template Filling: MUC," gives a comparative overview of the most significant research effort which is similar to what we are doing here, the Message Understanding (MUC) evaluations sponsored by DARPA. "The EDS *TemplateFiller*" discusses the various components of the *TemplateFiller,* with examples of input and output for each module, and describes the process of extending the system into a new domain. The final section presents *TemplateFiller* results, addresses some issues which remain difficult under this text distillation methodology, and suggests further work in this area.

Why Templates?

Here at EDS Research we have developed various effective retrieval, categorization, and summarization algo-

*e–mail: hkelly@edsr.eds.com

Received December 1, 1992; revised March 17, 1993; accepted April 13, 1993.

rithms. However, the EDS *TemplateFiller* goes beyond text retrieval and categorization and simple methods of summarization to the actual extraction of information from text. It uses text-filtering technology just to reduce the workload for the more computationally intensive processes of language analysis which are necessary for data extraction. It is this linguistic analysis which allows automatic text distillation, so that the large quantities of text which are available in the area of computer technology can be reduced to a manageable volume of pure information.

Given the general goal of automatic text distillation, the system could produce a number of different types of actual output, ranging from automatically generated sets of index terms to natural language abstracts of varying degrees of sophistication. In the *TemplateFiller* system, the output of the distillation process is extracted data presented in the form of a simple table, or template, with each row consisting of a slot name and a slot value. Each product description in an article results in the construction of one template. We have chosen this particular form of output because of the nature of the texts we are processing and the nature of the information needs we are addressing.

First, we find that product announcement texts, while they are unconstrained in format and overall writing style, are still somewhat regular in other ways, just by the nature of what the articles are talking about: new products with sets of standard features, including name, vendor, price, physical characteristics, and performance characteristics. Because the content of the articles shows some predictability, it is possible to represent it in a template with very little loss of information.

Second, because these articles are product announcements, both what readers want out of the text and how they will use what they get out of it are also predictable. What they want to retain from a product announcement text is a concise description of the product(s) discussed. What they want to do with such descriptions is to compare them. Representing product descriptions in the form of templates, which of course can also be straightforwardly converted to database records, provides the best vehicle for comparing products. Retrieving whole texts or even fragments of texts would be less satisfactory for at least three reasons: (1) new incoming texts often repeat information available earlier in other articles or elsewhere in other journals; (2) information in a single text may be economically conveyed in brief bulletin style, or it may be dispersed through a very long and otherwise factually uninteresting exposition; and (3) since we are already narrowing the subject area considerably by drawing from computer industry journals only, categorization must be very fine-grained and therefore sometimes goes wrong. All of these factors would make a system which returned texts rather than pure information more annoying than useful.

What we have constructed, therefore, is a system which can read the raw text of articles such as that shown in (1) and automatically produce from it filled templates such as those shown in (2). Both examples are actual *TemplateFiller* input and output.[1]

(1)

> ***DG Rolls Out Notebook 486DX-Based System***
> Data General Corp. last week released the 5.5-pound WalkAbout/386SL notebook and the Dasher II-486/50TE2 deskside PC.
>
> The 25MHz 386SL-based notebook starts at $2,445 with a 4-hour NiCad battery, 2M bytes of RAM and a 60M-byte hard drive; the 50MHz 486DX-based PC, which starts at $8,395, includes 8M bytes of RAM. Both units come with DOS 5.0 and Windows 3.1.

(2)

File:	PC_week.920904.20.clauses2
Vendor Name:	Data General Corp
Product Name:	Dasher_II-486_50TE2
Weight:	5.5 pound
CPU Type	80486DX
Ram (base):	8M
CPU Clock Speed:	50MHz
Footprint:	deskside

File:	PC_week.920904.20.clauses2
Vendor Name:	Data General Corp
Product Name:	WalkAbout_386SL
Weight:	5.5 pound
Price:	2445
CPU Type:	80386SL
Ram (base):	2M
CPU Clock Speed:	25MHz
Footprint:	notebook
Hard Disk Capacity:	60M

Other Research in Template Filling: MUC

Research which most closely resembles the work behind the EDS *TemplateFiller* is that associated with the ongoing Message Understanding evaluations sponsored by DARPA (Sundheim, 1991, 1992). These yearly efforts by a number of research sites have provided a forum for a standardized evaluation of the state of the art in information extraction. Our work, like the work associated with the MUC tasks, involves extracting information from text and using it to fill slots in a predetermined template. However, there are numerous differences between the MUC and EDS projects which make direct comparison inappropriate. The tasks MUC participants have been asked to perform differ from those undertaken by the EDS project in at least the following ways.

[1] There are two things to note about these examples. First, the text we have chosen to show here is very short (62 words) for example purposes. The actual length of the articles we process is closer to 300 words, with some articles longer than 700 words. Second, the templates in (2) are obviously not perfect. The one missed slot and one inaccurate slot are discussed in the final section.

The MUC-3 and MUC-4 tasks involved extracting information about terrorist activities from a wide variety of text sources, while the EDS system pulls information about new computer products from product announcement articles taken from a small number of computer industry journals. The nature of the domains, as well as the heterogeneity of the input, could have significant effects on the ease of information extraction.

The input text for MUC-3 and MUC-4 has been all upper case, while input text to the EDS system more naturally contains both upper and lower case. While this may seem an insignificant difference, our system depends heavily on case variation in recognizing novel strings as instances of such categories as *product name* or *company*, and this capability lessens the load on the preconstructed lexicon.

The MUC-3 and MUC-4 systems were required to find information about terrorist events, with each separate event producing a new template. A finite number of event types was assumed for each stage of the MUC project. For the EDS project, there is only one type of event—a product announcement—but there are several types of products that the system identifies, with each new product producing a separate template.

A single master template was used in each stage of the MUC research projects, varying in size and complexity from one year to the next; the MUC-4 template contained 24-slots. The EDS system in its present state uses two template types: one with 31 slots for personal computer systems, and another with 45 slots for printers. The single MUC template was mandated before the test, whereas we have had the freedom to design a template in accordance with the kinds of information that we find in the training texts as we create a domain.

Text for the MUC-3 and MUC-4 evaluations consisted of several hundred messages, some of which were used for training, and a random sample of which were reserved for a test set. Since the *TemplateFiller* was put into automatic operation last September, we have been processing text downloaded from Dialog^{TM} on a weekly basis without further work on the system. Each week's run of an average of about 65 articles is another test of the *TemplateFiller.* Since technology, and therefore terminology, change very rapidly in our domain area, we are leaving our training set further behind with each passing week. Some of our future research will be devoted to tracking the effect of this "domain drift" on system performance.

The MUC-3/MUC-4 tasks focused on a single domain: terrorist activities. We speak in terms of two "domains" for the existing *TemplateFiller.* While these two domains are very closely related, we have built into the system everything necessary to facilitate extension to any new domain, regardless of its distance from our present area of operation; two extensions from the original domain of personal computer systems have tested these domain extension utilities.

Finally, methods of evaluation differ somewhat between the EDS system and the MUC project. Evaluation will be discussed in more detail in the Results section.

The EDS *TemplateFiller*

The EDS *TemplateFiller* is designed as a series of processing modules. The transformation from raw text in machine-readable form to what we want as final output—very specific information represented in template or tabular form—cannot be effected in a single step. Rather, each module performs a different text-processing operation on the input and then sends its results to the next process, creating the effect of a series of text filters. The chained-together series of processes incrementally reduces the volume of downloaded text until what we have left is what we want and nothing else.

Because what we return is not a text or text fragment but distilled *information* which we have extracted from the raw text, our system must be able to discover not only entities but also the relationships that hold between entities which are encoded in the text. This requires more than simple word-based search or categorization techniques; it depends upon actual analysis of the language used in the text. Since this analysis is computationally intensive, we want to do this type of processing on as little text as possible. Therefore, the analysis stage is preceded by a number of simpler processes, some of which filter out articles and sections of articles which are uninteresting and some of which add information, in the form of entity labels of various types, which serves to streamline the final, most computationally intensive, processing stage.

Our philosophy has been to use a generic linguistic processing approach, applicable across any subject domain, in combination with programs which have access to repositories of domain-specific knowledge where appropriate. All of our natural language parsing technology, for instance, is domain independent; that is, neither the analytical methods employed by the parser nor the representation used for its output are customized to any particular subject area. The input that the parser provides to the domain-specific, goal-directed, template-building mechanism in the final stage of analysis is itself completely generic. However, both the parsing and the final data-extraction stages in the *TemplateFiller* benefit from the labeled bracketing introduced in an earlier stage by domain-customized preprocessing. We have found this combination of generic and domain-dependent components to provide a maximum of portability and power.

Two more benefits of this approach should be mentioned. First, since it is possible to examine the output of each processing stage separately, it is possible to identify the source of problems in the development phase by comparing input with output for any component module. Second, the ability of each of the *TemplateFiller* modules to stand on its own allows us to use each of them alone or in other combinations for other text-processing applications. For example, the parser and lexicon (see "Parsing and Logical Form" subsection) have been used in diverse projects with very little modification from one to the next. The same is true for the preprocessor (see "Format Regularization"). Various incarnations of the sentence selector (See "Sentence

Selection") exist for purposes of summarization (Shuldberg, 1991) and categorization. Modularizing allows reuse of applicable programs for many tasks.

The Template

The first step in creating a *TemplateFiller* system for a given domain or product type is designing the template. A template consists of a set of fields, or slots, each of which captures information about a particular feature of a target product. Since the template is domain-specific, it contains slots pertaining specifically to the type of product under consideration. In our case, we have been interested primarily in computer product announcements. We have designed templates for personal computer systems, for printers, and for monitors and displays.

The template design process requires reading sample articles from the domain and determining the most important characteristics of the product type. Some characteristics are tied to a particular product type, such as print engine speed for laser printers, or refresh rate for monitors. Others are generic (at least across computer product announcements), such as vendor name, product name, and price. Once the appropriate characteristics have been identified, they are represented as slots in the template. The template definition in (3) is the one we created for personal computer systems. The words *single* and *multiple* refer to the possibility of more than one value for a given slot.

(3)

Template:		computer_systems	
File:		single	
Vendor Name:		single	
Product Name:		single	
Product Model:		single	
Weight:		single	
Height:		single	
Depth:		single	
Width:		single	
Price:		single	
Number Parallel Ports:		single	
Number Serial Ports:		single	
Baud Rate:		single	
Warranty Period:		single	
CPU Type:		single	
Ram (base):		single	
Ram (maximum):		single	
CPU Clock Speed:		single	
Ram Wait States:		single	
Cache Size:		single	
Footprint:		single	
FloppyDiskAttrs		multiple	
	FloppyDiskSize		single
	FloppyDiskCapacity		single
	NumberDrives		single
SlotAttrs		multiple	
	SlotSize		single
	SlotBits		single
	NumberSlots		single
	BusType		single
DisplayAttrs		multiple	
	DisplayResolution		single
	DisplayType		single
	VideoStandard		single
HardDiskAttrs		multiple	
	HardDiskSize		single
	HardDiskCapacity		single
	HardDiskAccessTime		single

Note that a slot for product release date has not been included in the template, even though such information is likely to have high interest value. The slot has been omitted because, generally speaking, release date information in product announcement articles is vague; phrases like "next month" or "sometime early next year" are common. Although it may be possible to arrive at a more specific value for release date using the publication date for the article and then using information gained from the text to establish a distance from that date, such inferencing goes well beyond the scope of the current *TemplateFiller* system.

Note also that the template is not a completely flat structure, because some product characteristics cannot be adequately represented with such an object. Consider the following sentence.

(4)
In addition, the ME 486-EISA offers one parallel and two serial ports, seven 32-bit EISA slots and one 8-bit ISA slot.

To identify dependencies between slots, we define functionally dependent slots, where connected values form parts of a substructure. The template in (5) below was produced from the article which contained the sentence in (4). Note that the slot information has been collected into two functionally dependent structures labeled "Slot Info."

(5)

File:		pc_week1991-07-10-08:34-58.clauses2
Vendor Name:		Micro Express Inc
Product Name:		486-EISA_33
Price:		4999
CPU Type:		80486
Ram (base):		4M
Parallel Ports:		one
Serial Ports:		two
CPU Clock Speed:		33MHz
Slot Info:		
	Number:	one
	Bits:	8
	Bus Type:	ISA
Slot Info:		
	Number:	seven
	Bits:	32
	Bus Type:	EISA
Hard Disk Info:		
	Capacity:	150M

Such dependency relationships should be detected in the initial examination of the domain and encoded as part of the template construction process.[2] The use of functional dependencies addresses the same concerns that have prompted the MUC test designers to move to an object-oriented template (Krupka & Rau, 1992).

Employing a preset template has advantages and disadvantages. It restricts the kind of information that the system has to seek to a finite, manageable set of fields, but it also means that new developments in a domain will not be caught until the template is suitably modified, which, of course, does not happen automatically. On the positive side, limiting search to tightly focused topic areas means that when the system is presented with articles that have been miscategorized and have little or nothing to do with the topic of interest, the system often simply produces templates which are either empty or so sparsely populated that they are not preserved. This result ensures that miscategorized articles typically cost us only time and not accuracy.

TemplateFiller *Data Flow*

The actual work of text distillation by the *TemplateFiller* includes the following stages. Input to the system is raw, machine-readable text, which can come from any machine-readable source, but we are currently processing *PC Week* articles which we obtain from the Dialog™ service. First, downloaded articles are automatically categorized on the basis of their probable utility for a domain, and only those deemed relevant are subjected to further processing. The next stages are dedicated to preprocessing: cleaning up format problems and recognizing and labeling domain-specific semantic objects—potential slot fillers—in the raw text. Next, a keyword-matching routine is used to discard individual sentences which do not contain enough of these semantic objects to warrant further processing. The remaining text is analyzed linguistically, transforming the input into a representation which shows the underlying semantic relations between objects, including those recognized by the preprocessor as potential slot values. The final component, the template builder, uses that representation to select values for slots in the template.

Figure 1 illustrates the data flow through the individual *TemplateFiller* modules. Figure 2 gives examples of the output of the system at significant points in the flow. A more detailed discussion of each of the processing stages follows.

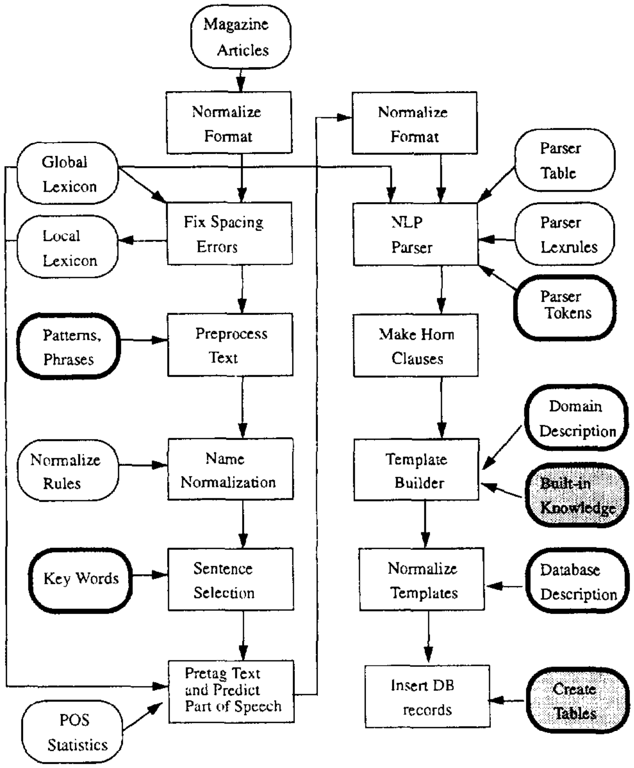

FIG. 1. *TemplateFiller* data flow.

The first step in template filling is article collection and categorization. A Bayesian categorization system (Hill & Schnedar, 1992; Mosteller & Wallace, 1964) is used to automatically choose articles that are likely to be relevant to the domain. The Bayesian categorizer was trained about two years ago on a set of 857 hand-sorted product release articles from computer industry journals. The classification which the system achieves is not perfect (see "Template-Filler Results"), but it does serve to reduce the workload for subsequent processes, and we have evidence that effective text categorization up front can contribute significantly to better end results (see "TemplateFiller Results"). In the meantime, miscategorized articles can be filtered out by later processes, so that in many cases we lose more time than accuracy. As for the occasional pertinent article which the system fails to retrieve, we rely on the repetition which is characteristic of the body of texts we receive to give us second chances at information missed in one pass.

Format Regularization, Preprocessing, and Part-of-Speech Tagging. The preprocessing stage includes a number of separate components, which accomplish the following operations in sequence: format normalization, repair of separated words, recognition of domain-specific semantic objects in the text, and part-of-speech prediction. All of these separate processes are similar in that they involve the addition of structure or information to the original text.

Articles that we download often contain formatting information that must be stripped out before the articles can be processed further. Header information, including bylines and datelines, one-line article summaries, and nonprinting characters, is removed. What remains is just the title and text of the article.

We have also found that it is common for downloaded articles from at least one source to contain words with randomly placed extra spaces in them. Since word boundaries

[2] Note that the functional dependency Hard Disk Info is also present, although only one of its subslots has been filled.

Original Text

Data General Corp. last week released the 5.5-pound WalkAbout/386SL notebook and the Dasher II-486/50TE2 deskside PC.

Preprocessed Text

```
((:PARAGRAPH 2) (:SENTENCE 1) (:COMPANY_SG "Data General" "Data General Corp.")
 (:TEMPORAL "last week" ) ((:VERB-PAST :VERB-PSP ) "released") (:DET "the")
 (:WEIGHT "5.5 pound" "5.5-pound" ) (:HAS_A_NUMBER "WalkAbout/386SL" )
 (:FOOTPRINT "notebook" "notebook" ) (:CONJ "and") (:DET "the")
 (:NAME_AND_NUMBER_SG "Dasher II-486/50TE2" )
 (:FOOTPRINT "deskside" "deskside PC" ) :PERIOD )
```

Logical Form

```
ctype          act
name           release
actor          ctype          company
               plu            -
               name           "Data General"
theme          list l(  reference      ref      definite
                                       name     the
                        attr l(  name     "5.5 pound"
                                 ctype    weight

                                 name     "WalkAbout_386SL"
                                 ctype    has_a_number )l
                        name     "notebook"
                        ctype    footprint

                        reference      ref      definite
                                       name     the
                        attr l(  name     "Dasher_II-486_50TE2"
                                 ctype    name_and_number_sg )l
                        name     "deskside"
                        ctype    footprint )l
               ctype    conj
               conjname and
adjunct        l(       name     "last week"
                        ctype    temporal)l
```

Template (1 of 2)

```
vendorname          "Data General Corp."
productname         "Dasher II-486/50TE2"
physicalweight      "5.5 pound"
cputype             "80486DX"
ramsizebase         "8M"          (supplied by another sentence)
cpuclockspeed       "50MHz"       (supplied by another sentence)
csfootprint         "deskside"
```

FIG. 2. Output at selected stages of processing.

are defined in our system by spaces, and since a number of subsequent processes depend on reliable word boundary identification, it is important that extra spaces be removed. Relying on our lexicon to identify known words, the space-fixer looks for unknown words in the text and attempts to combine them with adjacent words. If the combination produces a known word, the combination is used in place of the separated parts. Note that our modular arrangement allowed us to add this extra processing stage with no disruption to the rest of the system.

The next stage is string and regular expression matching on the text. Part of the *TemplateFiller* system's success is due to the fact that domain knowledge can be used in the early stages of processing to help identify key text strings that are likely to provide potential values for template slots. The preprocessor, with the help of domain-specific control files, recognizes such strings and labels them with appropriate token labels.

String constants that the system must recognize are words and phrases which either require a consistent labeling to aid in further processing or which have a particular meaning in the domain under consideration. Many product characteristics can be picked up directly in the form of string constants: known CPU types, known vendors, or particular kinds of ports or interfaces, for instance. Other characteristics are not completely described but can be located with the aid of words such as *RAM* or *ROM*, and strings such as *characters-per-minute* or *pages-per-minute*. Not only can these strings be labeled by the preprocessor, but also multiple distinct input strings can be mapped to the same output string. For instance, *pages-per-minute* and *ppm* both become *PPM* for the sake of subsequent processing.

Using regular expression definitions, we can identify a potentially open-ended set of strings as vendor names, product names or dollar amounts. For example, any string of capitalized words followed by *Co., Inc., Ltd.,* etc., is recognized and tagged as a company name. This allows many company names that have not been seeded into the string constant file to be recognized. Similarly, a variety of combinations of upper-case letters and numbers are tagged and in subsequent processes considered as possible product names. Previously recognized and labeled string constants can also serve as input to regular expression matching. For example, either *pages-per-minute* or *ppm*, recognized and tagged as *PPM*, can participate in a regular expression that looks for a number preceding the tagged *PPM* string. The number and the *PPM* string can then be combined and labeled as *PRINT_ENGINE_SPEED*, a printer characteristic that could be used to fill a template slot.

The tokenizer, a separate component which works on the output of the preprocessor, normalizes references in cases where one reference to a company or person is a shortened or elliptical version of another reference in the text. The tokenizer examines the set of tokens labeled by the preprocessor in a single text, and where it finds an acceptable partial match, it normalizes both strings into a single token. For example, if the article contains an early reference to *Consolidated Electronic Technologies, Inc.* and a subsequent reference to *Consolidated, Inc.*, the tokenizer will recognize that the two tokens refer to the same entity and return the same output string (the longer of the two) for both. While it is capable of normalizing any specified set of labeled tokens, we currently use the tokenizer just for company and personal names, since both are cases where the patterns of elliptical reference are fairly regular.

Recognizing and labeling strings at this early stage of processing provides two assets to subsequent processes. First, it catches words and phrases which may be proper values for slots in the template. Second, it collects multiword strings into single units that can be treated as such for syntactic processing.

The first sentence from the article in (1) is shown in (6) below, as it stands after format normalization, preprocessing, and tokenizing.

(6)

((:PARAGRAPH 2) (:SENTENCE 1) (:COMPANY "Data General" "Data General Corp.") (:TEMPORAL "last week") released the (:WEIGHT "5.5 pound" "5.5-pound") (:HAS_A_NUMBER "WalkAbout/386SL") (:FOOTPRINT "notebook" "notebook") and the (:NAME_AND_NUMBER "Dasher II-486/50TE2") (:FOOTPRINT "deskside" "deskside PC") :PERIOD)

The next stage is part-of-speech tagging for the remaining unbracketed words in the text. A major contributor to complexity in natural language analysis is grammatical category ambiguity. For example, the word *released* in (6) can be an adjective, a past-tense verb, or a past-participle verb. The syntactic parser has to know which interpretation is correct in order to parse the sentence successfully, but considering all the possibilities supplied by the lexicon increases parsing complexity and processing time. To help reduce this complexity, and therefore also processing time, we have implemented a program which statistically predicts part of speech on the basis of context (Church, 1988; deMarcken, 1989; DeRose, 1988). Our part-of-speech tagger narrows the choices for each word to one or two possibilities, with the result that the parser can now successfully handle much longer sentences than it could without tagging, and parsing is now twice as fast.[3]

Sentence Selection. In order to further reduce the amount of text that must be processed, the system discards some sentences from the original text. Sentences are preserved if they contain one or more of a predetermined set of keywords or, in some cases, only if they contain certain combinations of those keywords. Typically, keywords are preprocessor labels, since it is assumed that the strings of most interest have been recognized and tagged in the preprocessor stage. However, any word in the text can serve as a keyword. The result is a file that contains only the sentences which are most likely to contribute information to a template. The amount of reduction in text volume depends on a number of factors, including the number and nature of the keywords in the controlling file and the number of keyword occurrences in the input file; but typically, it is between 50% and 75%.

Parsing and Logical Form. Even though the preprocessor has identified and labeled potential strings for slot values in the template, linguistic analysis of the language used in the description is necessary to decide which values are the correct ones for a given template. Parsing is the most CPU-intensive part of the template-filling process; however, it is necessary for accurate information extraction. The grammar we have developed is a GPSG-HPSG hybrid (Gazdar, Klein, Pullum, & Sag, 1985; Pollard & Sag, 1987), and our parser is based on the LR parsing algorithm (Tomita, 1986 and 1991). Our syntactic lexicon of about 45,000 entries was created using information extracted from a machine-readable version of the *Oxford Advanced Learner's Dictionary of Current English,* which we obtained from the Oxford Text Archive.

Using just the syntactic lexicon and grammar, the parser is able to output syntactic structure trees for input sentences. Syntactic structure representations have been found to be sufficient for some information extraction applications, because they do capture many of the important relationships encoded in the text (Metzler et al., 1989). However, we have also augmented our lexicon with a set of lexical types which, when expanded, allow the construction of thematic structure representations (Jackendoff, 1972) as well as parse trees for input sentences. These thematic structure representations, or "logical forms," identify the underlying relationships encoded in a sentence.

A slightly simplified example of how our thematic structure represents the logical form of a sentence is shown below in (7). Note that preprocessor token labels are now called *ctype; has_a_number* is a particular preprocessor label for certain strings which have a high likelihood of

[3] The part of speech predictor can narrow the choice to a single possibility; however, in that case when it is wrong, it causes parse failures. When possibilities are statistically close in probability, we allow more than one category to be proposed.

being product names in this domain. Indentation represents embedding of one subgraph within another.

(7)
Sentence: DataGeneral last week released the 5.5 pound WalkAbout_386SL notebook.

Logical form:

```
ctype   act
name    release
actor   ctype   company
        name    "Data General"
theme reference      ref      definite
                     name     the
        name    "notebook"
        ctype   footprint
        attr|(  name    "5.5 pound"
                ctype   weight

                name    "WalkAbout_386SL"
                ctype   has_a_number )|
adjunct   name "last week"
          ctype temporal
```

Logical form representation exposes linguistic relationships in a more uniform way than surface syntactic structure representations can, and therefore, provides a better basis for the actual extraction of information. For instance, standard surface syntactic representations for active and passive variants of the same message differ. A search for underlying relationships which used surface syntactic structure as input would have to allow for both of the surface possibilities. However, in our thematic structure representations, such surface variation is neutralized; active and passive variants of the same message have functionally identical "logical forms."[4]

Our logical forms also identify other relationships which are implicit but invisible in surface structure. For example, in a sentence like the one in (8), English speakers know that *SuperCo* will be the vendor of *the computers,* even though *selling* has no explicit direct object. Our representation of this sentence uses numerical indices to make the implicit linking explicit; note that *SuperCo* is correctly identified by means of these indices (gap_106) as the actor of both *agree* and *sell,* while *computers* is linked to *sell* as its theme despite its separation from the verb in surface structure.

(8)
Sentence: The computers which SuperCo has agreed to start selling are expensive.

Logical form:

```
attr name expensive
ctype state
name be
topic reference ref definite
                    name the
name "computers"
ctype device_word
varname gap_106
attr aux have
    ctype act
    name agree
    actor [#1]
         name "SuperCo"
         ctype company
    theme modal to
          ctype act
          context start
          situation ctype act
                    name sell
                    actor →1
                    theme ctype ref
                          refname gap_106
```

Note that because of this ability to neutralize surface variation, all of the variations on the example sentence in (7) [shown in (9)a–c, below] would result in thematic structure representations which are identical in their critical elements; they would share the substructure shown in (10).

(9)

a. The WalkAbout_386SL notebook was released last week by Data General
b. Data General, which just released the WalkAbout_386SL notebook, has also announced other new products.
c. Data General has released a new 5.5-pound notebook, the WalkAbout_386SL.

(10)
Logical form:

```
ctype act
name release
actor name "Data General"
     ctype company
theme reference ref definite
                    name the
      name "notebook"
      ctype footprint
      attr |( name "WalkAbout_386SL"
              ctype has_a_number )|
```

In all these cases, the parser identifies the uniform relationship holding between the entity which has ctype *company,* which is a candidate for the slot *vendorname,* and the one with ctype *has_a_number,* which is one of several possibilities for the slot *productname.* When these representations are handed on to the template builder, that component need not know anything about the variation present in the original text. Instead, it can use a single thematic structure pattern to extract the correct data relationships from each of these sentences.

For a variety of reasons, such as memory limitations, ungrammatical or incomplete text, or syntactic structures that fall outside our grammar's coverage, parsing sometimes fails. Still, even when the parser has failed to resolve the structure of the sentence as a whole, in many cases it has succeeded in creating valid structures for some of

[4] Note that for purposes of tracking discourse topic we do retain a marker specifying whether a passive construction was the source of the thematic structure.

the sentence's constituents. In these cases, the successfully resolved structures, such as they are, are collected into a partial thematic structure called a Recovery Logical Form. Frequently, a Recovery Logical Form will be complete enough that it can be used by the template builder to fill template slots. A Recovery Logical Form is "complete enough" if it contains enough information to satisfy all the goal statements in a relevant PROLOG predicate in the template builder. This is discussed in more detail in the next section.

The Template Builder. The template builder is the component of the *TemplateFiller* system that makes assertions about specific values for template slots given thematic structure input. Our approach to constructing templates combines the general-purpose linguistic information produced by the parsing component with very specific information about what sort of templates we need to generate. Instead of trying to produce a complete knowledge representation of the information in each sentence, we try to prove simple relationships between information in the sentence and information we want. The process is more goal-directed and, we believe, more efficient than a more general knowledge-based approach. It is implemented in PROLOG as a simple pattern-matching and inferencing system.

The basic approach of the template builder is quite simple and relies on two kinds of information. At the top level, template definitions provide the goal for the inferencing process. At the bottom level, patterns of thematic structures define how individual pieces of linguistic information within a sentence can be combined to satisfy the goals established by the template definitions. The algorithm operates bottom-up, first identifying thematic patterns and then combining them into template structures.

We refer to patterns of thematic information as *configurations* in order to distinguish them from the regular expression patterns of the preprocessing phase of the system. Each configuration looks for a set of thematic elements that stand in a particular relationship to one another. For example, we have a series of configurations that look for verbal structures expressing relations between companies and products. One such configuration, shown here in (11), can be paraphrased as shown below in (12).

(11)

```
configuration(vendorl, vp, assert,
     and(match(company_act,
          actor(match(company, name(V))),
          theme(match(product_name, name(P)))),
     This),
     [vendorname(This, V), productname(This, P)]).
```

(12)

A vendorl pattern, composed of vendor V, and product P, is one in which:

a) the verb is a 'company_act,' encoding one of a predefined set of actions (e.g., "announce," "roll out," "upgrade," etc.),

b) the actor of the verb can be interpreted as a company name and has the surface form V, and

c) the theme of the verb can be interpreted as a product name and has the surface form P.

Note that the actor and theme portions of the configuration allow for items that may not have been explicitly marked as companies or products in the preprocessing stage.

When a configuration is matched, it produces a number of assertions. In the vendorl configuration above, two assertions are produced:

```
vendorname(This, V).
productname(This, P).
```

These are treated as a single ANDed assertion. Here, the variable 'This' will be bound to the thematic structure of the verb as required to maintain the association between the two pieces of information V and P and to provide sequencing information for the subsequent template-filling process. The variables V and P will be bound to actual text strings from the text, for example:

```
vendorname(lf1, 'Data General Corp.'),
productname(lf1, 'WalkAbout 386SL').
```

The second step in the template-building algorithm processes the assertions made in the configuration-matching step in sequential order, looking for the minimum set of templates that satisfy all the assertions. In other words, new assertions are combined with existing templates unless a conflict forces the creation of a new template. Each template is also treated as a set of related assertions, with the constraint that all assertions relating to single-valued slots are ANDed, and those relating to multivalued slots are ORed. Thus, a template corresponding to the example assertions shown above would have the following form.

```
template(t1),
vendorname(t1, 'Data General Corp.'),
productname(t1, 'WalkAbout 386SL').
```

The template-building process is simply one of finding an optimal set of template assertions consistent with the individual assertions produced by the configuration-matching phase. "Optimal" in our system means finding the set of templates that has the fewest number of filled slots. For example, given the template t1 above, the new assertions

```
productname(lf2, 'WalkAbout 386SL'),
csfootprint(lf2, 'notebook').
```

could be interpreted as a new template or as an extension of template t1. The first interpretation will result in two new slots in a new template, while the second results in only one new slot (csfootprint) in the existing template. Under our rule of finding the minimal set of templates, these assertions will cause template t1 to be extended.

```
template(t1),
vendorname(t1, 'Data General Corp.'),
productname(t1, 'WalkAbout 386SL'),
csfootprint(t1, 'notebook').
```

The template-building process operates across sentence boundaries and in a sense provides a crude process for resolving anaphoric reference. In the article (1) from which the preceding example was taken, the vendorname, productname, and footprint information come from one sentence. In a subsequent sentence, the cputype is mentioned in conjunction with the footprint, generating two new assertions.

```
csfootprint(lf3, 'notebook'),
cputype(lf3, '80386SL').
```

Although multiple candidate templates exist from the first sentence into which the new assertions might fit, the presence of the shared footprint information allows us to associate the new information with the correct template. This technique mimics the effect of definite NP anaphora, where a reference is made to a previously mentioned noun phrase by using a definite article (e.g., *the* as opposed to *a*). In the above example, the definite NP is *the 25MHz 386SL-based notebook,* and it adds two new pieces of information to the notebook template.

```
template(t1),
vendorname(t1, 'Data General Corp.'),
productname(t1, 'WalkAbout 386SL'),
csfootprint(t1, 'notebook'),
cpuclockspeed(t1, '25MHz'),
cputype(t1, '80386SL').
```

Enforcing the minimal set of templates helps to avoid the creation of spurious templates, but it can also have the effect of illegitimately combining slot values for separate products into a single template, which occurs most frequently when multiple products are described in terms of different attributes, or when products are described in terms of attributes that can fill multivalued slots. Enforcing the minimal rule typically produces the correct results, but obviously has the price of sometimes combining information into a single template that should appear in multiple templates; however, all things considered, the tradeoff appears to work to our advantage.

During the template-building process, a primitive approximation of the effects of discourse focus is applied which allows ordering of templates by last reference. If a set of assertions could apply equally well to more than one template, the most recent is selected. Pronouns are allowed to match in certain positions during the configuration matching process, causing the information associated with the pronoun to be attached to the most recently mentioned product consistent with the rest of the assertions.

Note that configuration definitions are separate from the actual pattern-matching algorithm that processes them. They are defined using a simple declarative syntax which facilitates maintenance. While developing the template-building system and extending it to new domains, we found that many of the configuration patterns could be expressed in terms of classes of "attributes" and mappings from these attributes to slot names. As a result, most of the patterns are defined in a domain-independent way and need only be augmented by a small number of domain-specific configurations, attribute definitions, and attribute-to-slot mappings.

As the preceding examples illustrate, the system does not really know anything about what it means to *announce* or *roll out* a new product. Instead it knows that a certain set of linguistic structures, such as those for the verbs *announce* and *roll out,* may contain information that we want. In addition, it knows where within these structures the pertinent information may be and how that information corresponds to slots in a template. Finally, it has a simple strategy for combining these small bits of information into coherent templates.

Domain Extension

The capacity to compress large quantities of incoming text down into a relatively small volume by the final stage of processing depends on the system's ability to take an extremely narrow view of what is likely to be important within a given domain. Ordinarily, while this kind of domain-targeting buys a great deal in terms of analysis of materials within the domain, it results in unacceptable fragility outside a domain. We have tried to prevent this in two ways: by compartmentalizing the system's knowledge, and by providing domain extension utilities that streamline the process of extending the system to a new domain.

Compartmentalization of knowledge itself takes two forms. First, the knowledge that the system uses at each of the processing stages is in all cases external to the programs that do the actual text processing. Therefore, no code, but only the knowledge files that the various programs consult, must be changed when we extend the system to a new domain. For example, the preprocessor consults domain-specific files of regular expression patterns and string constants for any given domain, but the program itself is domain-independent. The sentence selector and the template builder do likewise; on any given run they each consult files of target patterns pertinent to the domain in question, but they are themselves entirely generic and never need changing.

Second, we have divided the knowledge sources themselves into component files, depending on how generically useful the included knowledge is. These separate files are concatenated at run time into the single-file format that the various programs expect. For example, the preprocessor files are divided into generic, product-announcement, and domain-specific components. Patterns included in the first set would be useful across all text types; for example, they include patterns for recognizing personal names and titles, addresses, various ways of expressing numbers and measurements, etc. The second set includes patterns which are used in all categories of electronic technology product announcements; these files cover computer hardware "buzzwords" and phrases. The domain-specific set of files includes patterns which either are found only in discussions of a certain kind of hardware or have a unique meaning in such discussions. The target configurations used by the template builder have been divided in the same way, so that generic configurations for finding attributes of objects, for example, are maintained separately from configurations targeted at particular domain-dependent bits of information.

Because of this compartmentalization of the knowledge which the system requires, extending to a new but related domain is relatively simple and does not require the kind of backtracking that would be necessary without the system's generic core. In the case of moving from one computer hardware area to another, for instance, the new domain uses all the same code, and its knowledge files can be "seeded" with the generic and product-announcement knowledge files from the last domain; all that must be added is whatever completely domain-specific patterns and configurations the new domain may require. When we move beyond computer hardware product announcements to a completely different arena, we will take the generic patterns and configurations along as seed. Recently, this design allowed us to create a generic preprocessor for patent text in about 15 minutes.

To ensure consistency and generally to make the process easier, we have constructed a unified maintenance and testing environment for domain extension. Master knowledge files can be created or updated in their compartmentalized forms and can be concatenated and tested on any amount of text before being installed into the production system. The tool also includes a concordance utility, which facilitates the discovery of potential patterns within new batches of text files during the creation of a new domain. Using these tools, extending from the first domain we covered, personal computer systems, to the additional hardware domains of printers and monitors/displays took about two person weeks for each domain (Macpherson et al., 1992).

TemplateFiller Results

In this section, we describe the information extraction performance of the *TemplateFiller,* including overall statistics of precision and recall. As much as possible, we report our statistics in the terms used in the most recent MUC evaluations. This is not because the statistics should be compared (as discussed in "Other Research in Template Filling: MUC," the tasks and development constraints are too different in their details for comparison to be meaningful), but simply for the sake of following accepted practice.

System Status

We are now automatically building templates of new product information from *PC Week* articles and distributing them as weekly bulletins to interested parties within EDS. Our automatically generated bulletins typically require some editing, but the time we take doing that is much less than the time it would take to extract the information entirely by hand. The clerical errors that we sometimes make while filling templates by hand (a price of $22,500 became $225,000 in one article) continue to remind us of the utility of a completely automatic information extraction system. Hand-checking the generated templates also allows us to quantify success and identify problem areas with each weekly batch of templates. After hand-editing, each run is automatically scored for accuracy and completeness as part of the bulletin generation process.

System Performance

In Tables 1 through 5 we show performance statistics calculated over 12 weeks' worth of computer systems bulletins.

Table 1 shows the filtering effect of the Bayesian text classification program. Note that 658 articles downloaded over the 12 weeks are reduced by this method to 144 which must receive further processing, for a reduction of over 78%. Retrieval statistics show the classification system achieving 94% recall and 67% precision for the computer systems category, and 72% recall and 87% precision for printers.

Table 1 also shows two additional statistics, one a measure of system performance and the other a characteristic of the document collection. *Fallout,* which is a measure of the proportion of out-of-category documents which the system classifies as in-category, is satisfactorily low for both categories. *Generality* is the measure of how common articles of a particular category are within a collection of documents. All else being equal, low generality should impact precision; however, we see that text retrieval precision is better for the printer category than for computer systems, despite the relative scarcity of printer articles. What may explain this pattern is that the classification system is not just deciding *yes* or *no* for each category on its own, but is making a three-way decision among the two hardware domains and the negative category *neither.* Since the latest printers include memory, multiple ports, and even sometimes hard drives, articles about printers are difficult for the Bayesian classifier to distinguish from articles about computer systems. The collection covered by these statistics contains three instances in which printer articles were automatically categorized as computer system

TABLE 1. Categorizer performance.

c (compsys articles)	= 85
p (printer articles)	= 29
n (neither category)	= 544
t (total downloaded)	= 658
compsys generality = c/t	= 12.92%
printer generality = p/t	= 4.41%
cc (compsys articles correctly categorized)	= 80
yc (articles categorized as compsys)	= 120
pp (printer articles correctly categorized)	= 21
yp (articles categorized as printers)	= 24
compsys recall = cc/c	= 94.12%
compsys precision = cc/yc	= 66.67%
compsys fallout = (yc − cc)/(t − c)	= 6.98%
printer recall = pp/p	= 72.41%
printer precision = pp/yp	= 87.50%
printer fallout = (yp − pp)/(t − p)	= 0.48%

TABLE 2. Filtering effect of template filling.

(Records whether a template file is generated for a categorized article, not whether the templates in the file are right. Considers articles selected by Bayesian classifier as complete corpus.)

gtc (made template file for article RIGHTLY categorized as compsys)	= 72
tc (nonempty template files made for compsys)	= 94
empty compsys template files in category	= 8
empty compsys template files out of category	= 18
template filler compsys recall = gtc/cc	= 90.00%
template filler compsys precision = gtc/tc	= 76.60%
gtp (made template file for article RIGHTLY categorized as printer)	= 19
tp (nonempty template files made for printers)	= 21
empty printer template files in category	= 2
empty printer template files out of category	= 1
template filler printer recall = gtp/pp	= 90.48%
template filler printer precision = gtp/tp	= 90.48%

articles, and no instances of the opposite mistake. That is, the deficit to printer recall is at the same time a deficit to precision in the computer systems category.

Table 2 shows the additional filtering effect of the template filling process itself, given the output of the classification program. For these figures, just the articles automatically classified into a category are considered as the entire input collection for that domain. An article is considered to have been chosen as relevant in this stage if a nonempty template file was produced for it; no consideration is given to whether the information in the template file is completely correct or not. Here we see that the *TemplateFiller* weeds out about 10% of articles that it should have kept in both categories. It also does make templates for articles which were not placed in the correct category. Precision figures are somewhat higher than for the initial classification process, but since the generality of relevant articles is also much higher than in the initial collection (it is equal to the precision figures from the first stage), there is less likelihood of making a mistake at this stage.

Table 3 combines the figures of Tables 1 and 2 to show overall text-filtering performance, as if Bayesian classification and text selection by the *TemplateFiller* were all one text-filtering operation. Note that for both categories recall is lower here than for the Bayesian classifier by itself, because the template filler does fail to select some in-category articles. Precision, however, is higher in both categories. If recall and precision are summed into a single score, the total is about 161 for computer systems and about 156 for printers at this stage. *F*-scores (Lewis & Tong, 1992) with $\beta = 1$ (weighting precision and recall equally) are 80.5 for computer systems and 76.0 for printers.

TABLE 3. Overall filtering performance through template filling.

(Records whether a template file is made for a categorized article, not whether the templates in the file are right. Considers all downloaded articles as the source corpus.)

compsys recall after TF = gtc/c	= 84.71%
compsys precision after TF = gtc/tc	= 76.60%
printer recall after TF = gtp/p	= 65.52%
printer precision after TF = gtp/tp	= 90.48%

TABLE 4. *TemplateFiller* information extraction performance, all articles.

Total possible correct templates:	257	
a. Valid generated templates:	180	
b. Missing templates:	77	
c. Spurious generated templates:	86	
d. Ignored templates:	120	
e. Possible slots:	2294	
f. Correct slots in generated templates:	774	
g. Missing slots in generated templates:	696	
h. Modified slots in generated templates:	36	
i. Wrong slots in generated templates:	103	
j. Slots in added templates:	685	
k. Spurious slots in valid generated templates:	30	
l. Total spurious slots:	311	
Template recall (a/(a + b)): 180 of 257		70.04%
Template precision (a/(a + c)): 180 of 266		67.67%
Template overgeneration (c/(a + c)): 86 of 266		32.33%
Slot recall:		
all templates, modified slots are correct ((f + h)/e): 810 of 2294		35.31%
all templates, modified slots are incorrect (f/e): 774 of 2294		33.74%
generated templates, modified slots are correct ((f + h)/(e − j)): 810 of 1609		50.34%
generated templates, modified slots are incorrect (f/(e − j)): 774 of 1609		48.10%
Slot precision:		
all templates, modified slots are correct ((f + h)/(f + h + i + l): 810 of 1224		66.18%
all templates, modified slots are incorrect (f/(f + h + i + l): 774 of 1224		63.24%
valid generated templates, modified slots are correct ((f + h)/(f + h + i + k)): 810 of 943		85.90%
valid generated templates, modified slots are incorrect (f/(f + h + i + k)): 774 of 943		82.08%
Slot overgeneration:		
all templates (l/(f + h + i + l)): 311 of 1224		25.41%
generated templates (k/(f + h + i + k)): 30 of 943		03.18%
F-score, all templates, modified are incorrect, β = 1:		44.00

Tables 4 and 5 illustrate the actual information extraction performance of the *TemplateFiller* for computer systems articles.[5] The statistics for this assessment were assembled by comparing automatically generated template output with templates prepared by human template fillers. Our procedure is for two people to edit the output of each weekly run, and when agreement is reached on the best template

[5] We produce bulletins from articles announcing both personal computer systems and printers, but since the number of printer articles tends to be quite low (three or less per week) we have provided template-filling statistics for computer systems articles only.

TABLE 5. *TemplateFiller* information extraction performance, in-category texts only.

Total possible correct templates:	244	
a. Valid generated templates:	170	
b. Missing templates:	74	
c. Spurious generated templates:	53	
d. Ignored templates:	58	
e. Possible slots:	2208	
f. Correct slots in generated templates:	734	
g. Missing slots in generated templates:	667	
h. Modified slots in generated templates:	35	
i. Wrong slots in generated templates:	102	
j. Slots in added templates:	670	
k. Spurious slots in valid generated templates:	30	
l. Total spurious slots:	217	
Template recall (a/(a + b)): 170 of 244		69.67%
Template precision (a/(a + c)): 170 of 223		76.23%
Template overgeneration (c/(a + c)): 53 of 223		23.77%
Slot recall:		
all templates, modified slots are correct ((f + h)/e): 769 of 2208		34.83%
all templates, modified slots are incorrect (f/e): 734 of 2208		33.24%
generated templates, modified slots are correct ((f + h)/(e − j)): 769 of 1538		50.00%
generated templates, modified slots are incorrect (f/(e − j)): 734 of 1538		47.72%
Slot precision:		
all templates, modified slots are correct ((f + h)/(f + h + i + l)): 769 of 1088		70.68%
all templates, modified slots are incorrect (f/(f + h + i + l)): 734 of 1088		67.46%
generated templates, modified slots are correct ((f + h)/(f + h + i + k)): 769 of 901		85.35%
generated templates, modified slots are incorrect (f/(f + h + i + k)): 734 of 901		81.47%
Slot overgeneration:		
all templates (l/(f + h + i + l)): 217 of 1088		19.94%
generated templates (k/(f + h + i + k): 30 of 901		3.33%
F-score, all templates, modified are incorrect, $\beta = 1$:		44.54

fills for the week's articles, then the generated templates are automatically scored as part of the bulletin generation process.

In Table 4, the input is the entire set of articles which were automatically classified as containing information about computer systems. In Table 5, the input is limited to just those articles which were predicted by human judges to really contain such information, that is, it is limited to correctly classified articles. The two tables together reveal two significant phenomena. First, the difference between the two tables in slot precision for the *all templates* condition suggests that template-filling performance is affected by the success of article categorization. Since the system overgenerates in the presence of miscategorized article input, precision suffers. On the other hand, note that the "Total possible correct templates" figure is not the same in the two tables, but is higher in Table 4, as is the figure for template recall. This indicates that the system is extracting useful and correct information even from articles which were not judged to have been correctly categorized. "Perfect" domain categorization would therefore negatively impact system recall in this particular case.

Lines a–l in Tables 4 and 5 are self-explanatory, with perhaps a couple of exceptions. Line d ("Ignored templates") records the number of generated templates which did not contain enough information to justify inclusion in the bulletin; a template must have at least a vendor name and a product name in order to escape this classification. Line h ("Modified slots in generated templates") counts slot fills which were judged to be "close enough" to the completely accurate slot that some credit should be given. The MUC-3 and MUC-4 evaluation scoring allows partial credit for "Partially correct" fills; we choose instead to show all performance figures two ways: counting the "modified" fills as correct, and counting them as incorrect.

Recall and precision figures are calculated for several different ways of looking at system performance. The *all templates* condition is always harsher, as it compares the total number of slot fills automatically generated to the total number which would have been correct for that collection of articles. The *generated templates* condition allows us to look just at the automatically produced templates and judge recall and precision for them; that is, in this condition, we judge the completeness and accuracy of the templates that the system did generate. The *valid generated templates* condition ignores completely spurious templates and judges slot precision just on the valid templates produced by the system. Our *all templates* condition corresponds to the scoring method of the same name in the MUC-4 evaluations (Chinchor, 1992).

Continuing Problems

In general, the system does best on articles which discuss only one product. Where an article contains multiple product descriptions, it is necessary to keep facts about one product from being attached to the templates for other products, and this requirement strains our rather simplistic methodology for combining information which comes from different sentences. An example of the difficulty involved in even a simple text can be seen in (1) and (2), repeated here as (13) and (14).

(13)

DG Rolls Out Notebook 486DX-Based System

Data General Corp. last week released the 5.5-pound WalkAbout/386SL notebook and the Dasher II-486/50TE2 deskside PC.

The 25MHz 386SL-based notebook starts at $2,445 with a 4-hour NiCad battery, 2M bytes of RAM and a 60M-byte hard drive; the 50MHz 486DX-based PC, which starts at $8,395, includes 8M bytes of RAM. Both units come with DOS 5.0 and Windows 3.1.

(14)

File:	PC_week.920904.20.clauses2
Vendor Name:	Data General Corp
Product Name:	Dasher_II-486_50TE2

```
Weight:              5.5 pound
CPU Type             80486DX
Ram (base):          8M
CPU Clock Speed:     50MHz
Footprint:           deskside

File:                PC_week.920904.20.clauses2
Vendor Name:         Data General Corp
Product Name:        WalkAbout_386SL
Weight:              5.5 pound
Price:               2445
CPU Type:            80386SL
Ram (base):          2M
CPU Clock Speed:     25MHz
Footprint:           notebook
Hard Disk Capacity:  60M
```

Note in (14) that we fail to pick up the price of the Dasher_II-486_50TE2 because, while the system is able to link the relative clause containing the price to the noun head PC, it does not know that PC can be an elliptical way to refer to a deskside PC. On the other hand, the system does manage to extract Hard Disk Capacity, CPU Clock Speed, CPU Type, and RAM (base) from the same environment, using a target configuration which works here but which could be too permissive in combining assertions in other cases. This same example, in fact, shows a case in which such overpermissiveness does go wrong: there is no linguistic basis for distributing the weight value "5.5 pound" across both templates, but the system does it on the basis of a configuration which is simply less restrictive than it should be. Just as in other information retrieval tasks, tradeoffs between recall and precision must be made here also.

An eye-opening lesson we have learned in the course of this research is the importance of certain seemingly low-level steps in the processing. Although we use linguistic processing to make the ultimate determination of relationships holding between entities in the text, the system is still very dependent on the early bracketing of semantic objects. Using the power of regular expression matching to accomplish this tokenizing obviously makes us much less dependent on literal patterns in the text than we otherwise would be, but we still often miss information because an object in the text which should be caught as an instance of a particular category falls slightly outside our regular-expression definition of that category. Again, this is a case where tradeoffs are necessary, because the broader the patterns are, the more likely they are to bracket elements inappropriately.

Likewise, enhancing the power of the up-front matching must be balanced against letting the linguistic analysis components do their job. The early bracketing speeds parsing enormously, because it reduces both the number of "words" (words and tokens) in a sentence and overall category ambiguity, thereby cutting complexity tremendously. On the other hand, the grammar now contains a few very counterinutitive rules, because it must accommodate the "help" that it is being given by the preprocessing components. For instance, because we capture as labeled tokens many strings which consist of a number followed by some quantifiable feature such as "Mbytes of memory" or "pages-per-minute," we end up with artificial "nouns" (e.g., "8 Mbytes of memory," "four pages per minute") which carry a feature that indicates that they contain a number. Since the number itself is not then a terminal element in the eyes of the parser, we must allow some of the modifiers which would ordinarily attach only to numbers (like "approximately"), to attach instead to these constructed "nouns."

Conclusion

The EDS *TemplateFiller* is an attempt to take information distillation and retrieval to one of many possible extremes. In this article, we have stressed some of its more important characteristics, such as its modularity and its use of natural language technology, and we have provided some typical results which show both its strengths and weaknesses.

We have described the *TemplateFiller* as an information distillation system. Yet one of its properties that we have discussed only indirectly is that, in its attempt to discover the information-intensive data core that is expressed in final template form, it actually adds a considerable amount of information to the original text. Each module makes use of a repository of knowledge to recognize and label text strings in ways that aid subsequent modules in their data recognition tasks, but which also amount to adding more information to the text. This is perhaps most obvious with the preprocessor and tagger, which recognize certain text strings and attach information-carrying labels to them, and the parser, which creates an entirely distinct and heavily annotated structure out of this labeled text. At each stage of the process, the output resembles the original raw text input less and less as more and more information is added. In fact, after the filtering effect of categorization, only two modules of the system actually reduce the amount of data from input to output: the sentence selector, which filters out sentences it does not see as relevant to the domain, and the template builder, which reduces the information it recognizes in the thematic structure to a simple set of assertions. It is perhaps ironic that the high ratio of text-to-data reduction achieved by the *TemplateFiller* is largely due to the addition of information at most stages; however, this methodology has been quite successful for our purposes.

We plan to further address issues of discourse and anaphora resolution in the future. Better methods in these areas could ultimately increase both recall and precision; the short-term necessity will be to make sure that increases in recall brought about through an enhanced ability to detect anaphoric reference do not result in the lowering of precision. We also hope to move to new domains and to broaden the raw text input to include multiple periodicals, so that the system will eventually be processing several hundred articles per week across a variety of computer products–related subject areas. What we have achieved so far clearly demonstrates the feasibility of fully automatic distillation of pure information from text.

References

Chinchor, N. (1992). MUC-4 Evaluation Metrics. *Proceedings of the Fourth Message Understanding Conference* (MUC-4), pp. 22–29.

Church, K. (1988). A stochastic parts program and noun phrase parser for unrestricted text. *Proceedings of the Second Conference on Applied Natural Language Processing,* 1988.

DeRose, S. (1988). Grammatical category disambiguation by statistical optimization. *Computational Linguistics, 14,* pp. 31–39.

de Marcken, C. (1990). Parsing the LOB corpus. *Proceedings of the 28th Annual Meeting of the Association for Computational Linguistics,* pp. 243–251.

Gazdar, G., Klein, E., Pullum, G. K., & Sag, I. A. (1985). *Generalized phrase structure grammar.* Cambridge, MA: Harvard University Press.

Hill, J., & Schnedar, M. (1992). *Bayesian procedures for automatically categorizing text documents* (Technical Paper). Albuquerque, NM: EDS Research.

Jackendoff, R. S. (1972). *Semantic interpretation in generative grammar.* Cambridge, MA: MIT Press.

Krupka, G., & Rau, L. (1992). GE adjunct test report: Object-oriented design and scoring for MUC-4. *Proceedings of the Fourth Message Understanding Conference (MUC-4),* pp. 78–84.

Lewis, D. D., & Tong, R. M. (1992). Text filtering in MUC-3 and MUC-4. *Proceedings of the Fourth Message Understanding Conference (MUC-4),* pp. 51–66.

Macpherson, M., Corley, J., & Shuldberg, K. (1992). *Template filling domain extension: Report of the first domain* (Technical Report). Albuquerque, NM: EDS Research.

Metzler, D. P., Haas, S. W., Cosic, C. L., & Wheeler, L. H. (1989). Constituent object parsing for information retrieval and similar text processing problems. *Journal of the American Society for Information Science, 40,* 398–423.

Mosteller, F., & Wallace, D. (1964). *Applied Bayesian and classical inference: The case of the Federalist Papers.* New York: Springer-Verlag.

Pollard, C., & Sag, I. (1987). *Information-based syntax and semantics.* CSLI lecture notes.

Shuldberg, H. K. (1991, October). The EDS summarizer: Automatic generation of article summaries. *ASIS Workshop on Language and Information Processing,* Washington, DC.

Sundheim, B. M. (1991). Overview of the third message understanding evaluation and conference. *Proceedings of the Third Message Understanding Conference (MUC-3),* pp. 3–16.

Sundheim, B. M. (1992). Overview of the fourth message understanding evaluation and conference. *Proceedings of the Fourth Message Understanding Conference (MUC-4),* pp. 3–21.

Tomita, M. (1986). *Efficient parsing for natural language.* Boston, MA: Kluwer.

Tomita, M. (1991). *Generalized LR parsing.* Boston, MA: Kluwer.