The PUNDIT
Natural-Language Processing System

Lynette Hirschman, Martha Palmer, John Dowding,
Deborah Dahl, Marcia Linebarger, Rebecca Passonneau,
François-Michel Lang, Catherine Ball, Carl Weir

Paoli Research Center, Unisys
P.O. Box 517, Paoli, PA 19301

ABSTRACT
In this paper we describe the PUNDIT\textsuperscript{1} text-understanding system, which is designed to analyze and construct representations of paragraph-length texts.\textsuperscript{2} PUNDIT is implemented in Quintus Prolog, and consists of distinct lexical, syntactic, semantic, and pragmatic components. Each component draws on one or more sets of data, including a lexicon, a broad-coverage grammar of English, semantic verb decompositions, rules mapping between syntactic and semantic constituents, and a domain model. Modularity, careful separation of declarative and procedural information, and separation of domain-specific and domain-independent information all contribute to a system which is flexible, extensible and portable: Versions of PUNDIT are now running in five domains, (four military and one medical).

1 Introduction

PUNDIT is a large, modular natural-language text-understanding system implemented in Prolog. This paper discusses the design and implementation of PUNDIT, focusing on several issues which have been crucial to the development of the system:

- Portability, supported by clear factoring of domain-independent and domain-specific information, and by a collection of tools to support creation of the various modules;
- Modularity, supporting incremental development of a large-scale system including lexical, syntactic, semantic, and pragmatic components;
- Integration of multiple sources of information, to provide search focus during parsing and convergence on a correct interpretation of individual sentences (and ultimately of the entire discourse);
- Robustness, provided by a broad-coverage grammar, integration of multiple knowledge sources to detect inconsistent information, and feedback to the user to provide help in diagnosing missing or incorrect information;
- A development environment for the construction of a large-scale natural-language processing system, including tools for debugging, testing, updating, and tailoring the system to different types of development.

We begin by describing some applications of PUNDIT and the domains of discourse in which PUNDIT has been used. Next, we present the three principal modules of the PUNDIT system: Syntax, semantics, and pragmatics. Section 6 deals with the problem of portability, and we conclude with a brief discussion of implementation and system issues. Figure 1 describes the overall architecture of the system.

2 Applications and Domains

The core facilities of PUNDIT, which are represented by rectangular boxes in Figure 1, are domain- and application-independent: They provide the basic text-understanding capabilities of the system, accepting natural-language text
Figure 1: Organization of PUNDIT
as input and producing a representation of the meaning of the text as output. (See Figures 2 and 3 for sample output.) For a specific application within a given domain, the core system is augmented by domain-specific data files (the knowledge base, the lexicon, etc.) and by application modules for special front-end and back-end functions. These provide the user interface to PUNDIT and may perform application-specific tasks based on PUNDIT’s output. Some of these tasks include summarizing the input, interfacing to an expert system, and updating or querying a database. The two major application areas to date have been message processing and database queries. Future plans include integration of a speech-recognition capability.

2.1 Message Processing

Applications have been developed to process four types of messages or reports: maintenance reports on Burroughs equipment, messages reporting equipment failures on Navy ships (CASREPS), Navy RAINFORM tactical messages, and Trouble and failure reports (TFRS) from Trident submarines. We are now also porting the system to a domain of medical abstracts.

The CASREPS application was developed in the context of a Navy battle management domain focused on force readiness. For this application, the system processes the remarks field of messages describing failures of starting air compressors (sacs), and generates a tabular summary of the major problems and findings. A sample CASREP summary is shown in Figure 2.

In the RAINFORM message application we also process message “header” information (originator, report date, etc.), and use this to create a context for the interpretation of the message.

The Trident TFR application, which is currently under development, allows a message to be collected interactively through a prompt-response dialogue. The system asks the user a number of questions designed to elicit key facts about the problem (what went wrong, what was the cause, what action was taken, etc.); the user’s answers are analyzed and validated by PUNDIT, and the results of analysis are used to update a historical database of equipment problems. This database can then be queried using a database query language or an English-language front end (another application of PUNDIT—see subsection 2.2). The three Navy applications have been developed as part of the DARPA Strategic Computing program.

2.2 Natural Language Query Processing

The core components of PUNDIT analyze an English query and produce a set of meaning representations, which are then mapped to a set of database relations. For queries to a foreign database (either remote or local), the database relations are translated into QUEL, and the resulting QUEL query is used to access an INGRES database. The results are displayed to the user. Alternatively, the database relations may be used to access an internal Prolog database. These approaches have been used to support English queries on an INGRES database of ship movements, and on Prolog databases of CASREPS, RAINFORMS, and computer failure reports.

3 The Syntax Component

In PUNDIT, syntactic analysis yields two syntactic descriptions of the sentence. One is a detailed surface parse tree, and the other is a regularized structure called the Intermediate Syntactic Representation (ISR). The surface structure parse tree is produced during syntactic analysis, and is used to check syntactic constraints. The ISR is constructed incrementally from the surface parse tree and reduces surface structure to a canonical predicate-argument form appropriate for input to semantics (see Section 4) and selection (see Section 6.1).

3.1 The Grammar

The syntactic component of PUNDIT is based on the logic-grammar formalism of Restriction Grammar [8] (a descendent of Sager’s string grammar [16]), and includes facilities for writing and maintaining large natural-language grammars. The grammar consists of a set of context-free BNF definitions (currently numbering ap-
message:
LOSS OF LUBE OIL PRESSURE DURING OPERATION.
INVESTIGATION REVEALED ADEQUATE LUBE OIL
SATURATED WITH METALLIC PARTICLES.
REPLACED SAC.

Finding:
Part: oil pressure  State: lowered

Finding:
Part: oil  State: saturated with particles

Status of Sac:
Part: old sac  State: removed from starting_air_system

Status of Sac:
Part: new sac  State: included in starting_air_system

Figure 2: CASREP summary

approximately 80) augmented by restrictions (approximately 35), which enforce context-sensitive well-formedness constraints and, in some cases, apply optimization strategies to prevent unnecessary structure-building. The grammar can either be interpreted or translated (or a mixture of both), and uses a top-down left-to-right parsing strategy augmented by dynamic rule pruning for efficient parsing [4]. In addition, a meta-grammatical component generates definitions for a full range of co-ordinate conjunction structures [6] and wh-constructions [7]. Syntactic phenomena treated by the current grammar include declarative sentences, imperatives, questions, sentence fragments, sentence adjuncts, conjunction, relative clauses, complex complement structures, wh-constructions, and a wide variety of nominal structures, including compound nouns, nominalized verbs and embedded clauses.

3.2 Intermediate Syntactic Representation

In parsing a sentence, the syntax processor uses the rules of the grammar to produce two structures: a detailed surface-structure parse, and an Intermediate Syntactic Representation (ISR), which is a regularization of the syntactic parse into a canonical predicate-argument form. Regularization is accomplished by annotating each rule in the grammar with an expression in the lambda-calculus-based translation rule language, which provides directions for combining the ISRs from the children of a node into the ISR of their parent.

An important step in the regularization process involves mapping fragment structures onto canonical verb-subject-object patterns, and flagging missing elements. For example, a two fragment consists of a tensed verb + object, such as Replaced sac. Regularization of this fragment maps the two syntactic structure into the verb + subject + object structure replaced + pro(elided) + sac. The semantic and pragmatic components provide a semantic filler for the missing subject (represented as pro(elided) in the ISR) using general pragmatic principles and specific domain knowledge [12]. The ISR for the sentence Re-

3The rules for fragments enable the grammar to parse the "telegraphic" style characteristic of message traffic, such as Disk drive down, and Has select lock.
placed sac is the following:

\[
\begin{align*}
\text{[past, replace,} & \\
\text{subj([pro([elided,singular,ID1]))],} & \\
\text{obj([det([]),} & \\
\text{[noun([sac,singular,ID2])])])}
\end{align*}
\]

The variables ID1 and ID2 in the ISR are instantiated by the semantics and reference resolution components to distinct referential indices.

4 The Semantics Component

4.1 Clausal Decompositions

Semantic representations of sentences are derived from the corresponding ISRs. The main verb of each clause is decomposed into a predicate/argument structure that makes explicit the relations comprising the verb's meaning. The arguments of each predicate in the verb decomposition are either further semantic predicates or thematic roles [11]. For example, the verb replace decomposes into the following representation:

\[
\text{causeP(agent(AGENT),} \\
\text{becomeP(exchangedP(} \\
\text{object1(OBJ1),} \\
\text{object2(OBJ2))))}
\]

The causeP predicate denotes a relation between an individual (the agent) and another predicate representing the type of relation that the agent is responsible for bringing about in a replace event. AGENT, OBJ1, and OBJ2 are Prolog variables that are eventually bound to discourse referents through a process of semantic interpretation briefly described below. The first argument of causeP, the agent thematic role, thus evaluates to a discourse referent representing the individual doing the replacing. The second argument of causeP, namely

\[
\text{becomeP(exchangedP(object1(OBJ1),} \\
\text{object2(OBJ2)))}
\]

is a predicate representing the type of situation that is brought about by the agent discourse entity. It consists of two nested predicates: the aspectual operator becomeP, which provides information about the temporal structure of a replace event (cf. Section 5.3 and Figure 3), and the predicate exchangedP, which specifies the result of a replace event as a relation among the discourse entities that are eventually bound to OBJ1 and OBJ2.

4.2 Thematic Roles

In the domains that PUNDIT currently supports, the thematic roles used are: actor, agent, experiencer, goal, instigator, instrument, location, object1, object2, patient, referencePt, source and theme. A dramatically different domain such as children's birthday parties might require additional thematic roles. Our goal is to establish a collection appropriate for a wide range of domains, since the algorithm that assigns referents to roles is general across domains.

For any specified context, a thematic role may be asserted to be obligatory or essential [12]. If a role is obligatory in a specified context, then it is required that it be filled by a referent introduced overtly by some expression in the current utterance. If a role is essential, it must be filled either by a referent introduced by an expression in the current utterance, or by some referent in the discourse context. Roles that are not declared to be obligatory or essential with respect to some predicate environment need not be filled at all in that environment.

The algorithm used to fill thematic roles makes use of syntactic mapping rules to associate types of syntactic constituents with thematic roles. For example, one mapping rule indicates that a subject can provide a filler for the agent role. These mapping rules can be tailored to account for idiosyncrasies of particular verbs, but for the most part they are fairly general and port readily to new domains. Selectional rules express restrictions on fillers of thematic roles for a given class of verbs. These restrictions are more domain specific, since they are heavily dependent on the co-occurrence patterns and semantic classes characteristic of the verb in the given domain.

4.3 Nominalizations

The interpretation of nominalizations is similar to the interpretation of clauses, but differs in...
what linguistic information is accessed and how thematic roles are assigned [3]. In the most straightforward cases, the decomposition of a nominalization is the same as that of its corresponding verb, but with different syntactic mapping rules. For example, comparing *Replaced sac* with *Replacement of sac*, a likely filler for the *object1* role of *replace* is the referent introduced by the syntactic object, but a likely filler for the *object1* of *replacement* is the referent introduced by an *of*-prepositional phrase. In addition to having different mapping rules, nominalizations do not have any obligatory roles. A role that would be obligatory for the verb, such as the *object1* role for *replace*, is considered to be essential for nominalizations.

5 The Pragmatics Components

5.1 Reference Resolution

Reference resolution is called by semantics when it is ready to instantiate a thematic role with a specific referent. Reference resolution finds the referent of noun phrases, creates unique identifiers for entities (see the *id* relations in Figure 3), and also establishes other semantic relationships among entities mentioned in the text. The reference resolution component handles the following phenomena:

- pronouns (including zeroes, such as the unexpressed subject in *Replaced sac*) and *one-anaphora*, using a syntax-based focusing algorithm [2];
- definite and indefinite noun phrases, as well as noun phrases without determiners found in telegraphic-style messages;
- *implicit associates*, [2,1] such *sac* and *pressure* in *Sac failure due to loss of oil pressure*, where it is important to express the fact that the oil under consideration is the oil in the sac, not just any oil;
- conjoined noun phrases;
- nominal references to events and situations first mentioned in clauses, such as *failure in Sac failed. Failure occurred during engine start [3]*;
- referents not mentioned explicitly, such as the investigated item in *Investigation revealed adequate lube oil [12]*;
- Processing for pronominals (i.e., pronouns, *one-anaphora*, and zeroes) selects a referent from the *Focus List* (see Section 5.2 and Figure 3). If no previously mentioned entity is appropriate, the system uses a default. For example, in analyzing *Replaced sac*, the default *ship’s force* is taken as the agent (see Section 4.1 for the decomposition of *replace*). The referents for full noun phrases and implicit referents are selected from entities mentioned in the discourse. If there is no explicit referent for a definite noun phrase or for a noun phrase without a determiner, the system associates the entity with one in focus via an implicit associate relationship. Indefinite noun phrases are assumed to introduce new referents. While this is is not strictly correct [1], it seems to be sufficient for the CASREP domain. After a referent is found, control returns to semantics.

5.2 Discourse Integration

Discourse integration occurs following the completion of semantic analysis of each sentence. This stage of processing updates the list of discourse entities with entities first mentioned in the current sentence, and maintains a list of entities that are available as referents of pronouns (the *Focus List*). The output of discourse integration is the *Integrated Discourse Representation* (IDR), which represents the entities and situations which have been mentioned in the discourse, and temporal relationships among the situations. Figure 3 shows a simplified version of the IDR produced by the analysis of the CASREP message shown with its summary in Figure 2.

5.3 Time Analysis

PUNDIT’s temporal analysis component distinguishes between potential and actual situations, and determines what kinds of intervals are associated with the latter and when they occur [14,13]. Three temporally distinct types of situations are distinguished: states, processes, and events. Each is represented with a distinct *temporal structure* indicating the time intervals over which the situation holds. The *temporal location* of a situation is represented in terms of

239
Focus List: [replace1, force4, [compressor8], sac, [reveal1],
[investigate1], [lose1], [operate2], [pressure3]]

Ids:
id(ships*force,[force4]) id(starting_air_system,[starting_air_system9])
id(sac,[compressor8]) id(sac,[compressor9])
id(state,[replace2]) id(event,[replace1])
id(1ube*oil,[oil3]) id(pressure,[pressure3])
id(state,[lose2]) id(event,[lose1])

Events and Processes:
event([lose1])
becomeP(loweredP(patient([pressure3])))
belowP(goal(.-22866),ref_pt(.-22868))
belowP(goal(.-22866),source(.-22877))
moment([lose1]))

event([replace1])
causeP(agent([force4]),
becomeP(exchangedP(object1([compressor8]),object2([compressor9])))
becomeP(exchangedP(object1([compressor8]),object2([compressor9])))
missingP(theme([compressor8]),source([starting_air_system9]))
includedP(theme([compressor9]),goal([starting_air_system9]))
moment([replace1]))

States:
state([lose2])
loweredP(patient([pressure3]))
belowP(goal(.-22866),ref_pt(.-22868))
belowP(goal(.-22866),source(.-22877))
period([lose2]))

state([replace2])
exchangedP(object1([compressor8]),object2([compressor9]))
missingP(theme([compressor8]),source([starting_air_system9]))
includedP(theme([compressor9]),goal([starting_air_system9]))
period([replace2]))

Complete Time Relations:
coincide(moment([lose1]),moment([operate2]))
precedes(moment([lose1]),moment(discourse_time))
precedes(moment([replace1]),moment(discourse_time))
precedes(moment([operate2]),moment(discourse_time))

Figure 3: IDR for CASREP message
three temporal indices [15]: a moment associated with the situation interval(s) (Reichenbach's event time); the discourse time, the time at which the text was produced (Reichenbach's speech time); and possibly a third time introduced by the past perfect tense (Reichenbach's reference time). PUNDIT currently processes certain relational adverbiai phrases and clauses [17], such as the prepositional phrase in Loss of lube oil pressure during operation. Determining when the loss and operation occurred involves deciding whether to interpret the fragment as past or present, which in turn depends on the lexical aspect of loss [10], and on the interpretation of the prepositional adverb during. The result of processing during is the relation

\[
\text{coincide(}\text{moment(}\{\text{lose1}\})\text{,}
\text{moment(}\{\text{operate2}\})\text{)}
\]

which means that some moment of the loss coincided with some moment of the operation. Since the lexical aspect of loss indicates past time, we also have the relation

\[
\text{precede(}\text{moment(}\{\text{lose1}\})\text{,}
\text{moment(}\text{discourse-time}\})\text{)}
\]

as shown in Figure 3.

In order to determine the temporal structure of a predication, PUNDIT needs two kinds of input: the predicate/argument structure produced by the semantic component, and the surface tense and aspect markings (e.g., perfect or progressive). The predicate/argument structure includes aspectual operators [5], which represent the inherent lexical aspect of the verb as stative (no aspectual operator), active (includes the operator doP) or eventive (includes the operator becomeP). These aspectual distinctions are relevant for verbs whose arguments are simple concrete entities (e.g., fail in the compressor failed where the verb and its argument refers to a single fail event). Another type of lexical semantic information used in temporal analysis pertains to the distinction between verbs in this class, and verbs whose arguments are themselves events (e.g., occur in a failure occurred). The temporal component recognizes three such verb classes, making it possible, among other things, to derive similar temporal information for the two different ways of referring to a failure event (e.g., Compressor failure occurred and The compressor failed) [13].

6 Portability

PUNDIT has been carefully designed with portability in mind: For example, the modular design of the system ensures that the domain-independent and domain-specific components of the system are kept completely separate. This modularity greatly facilitates porting PUNDIT to new domains, since such a port requires modification only of the domain-specific knowledge bases. In Figure 1, the rectangular boxes represent the domain-independent modules, and the diamond-shaped boxes represent the domain-specific knowledge bases. We have also implemented a number of modules and tools designed to further enhance PUNDIT's portability, which are described in the remainder of this section.

6.1 Automatic Generation of Selectional Restrictions

We have implemented the interactive module SPQR4 [9] which presents to the user syntactic co-occurrence patterns to be validated as they are generated during parsing. This interactive program greatly reduces the number of incorrect parses generated as well as the number of grammar rules tried, because any partially constructed parse containing an invalid syntactic pattern fails immediately. More importantly, however, the information about syntactic patterns can be stored, with patterns classified as good or bad, so that no pattern is ever presented to the user more than once. The system can thus interactively acquire domain-specific semantic information, and effectively bootstrap into a new domain.

SPQR operates by presenting to the user a syntactic pattern found in the ISR, such as subject-verb-object or noun-adjective, and querying him/her about the acceptability of that pattern. When presented with a syntactic pattern, the user can
respond to the query in one of two ways. If the pattern describes a relationship that can be said to hold among domain entities, the user accepts the pattern, thereby classifying it as good. An example of a good pattern is the n/pp pattern \[\text{[loss, of, sac]}\], generated by the fragment \text{Loss of second installed sac}. After encountering a good pattern, SPQR continues the analysis of the ISR, and the parsing of the sentence is allowed to continue. If, however, the pattern describes a relationship among domain entities that is not consistent with the user's domain knowledge or with his/her pragmatic knowledge, the user rejects it, thereby classifying it as bad, and signalling an incorrect parse. An example of a bad pattern is the subject-verb-object pattern \[\text{[loss, install, sac]}\], generated by the semantically anomalous analysis of the sentence \text{Loss of second installed sac} in which \text{installed} is taken as the main verb. Encountering a bad pattern causes the restriction which checks selection to fail, and as a result, the parse under construction is immediately failed, and the parser backtracks. As the user classifies these co-occurrence patterns into good patterns and bad patterns, they are stored in a pattern database which is consulted before any query to the user is made. Thus, once a pattern has been classified as good or bad, the user is not asked to classify it again.

6.2 The Development Environment

PUNDIT's development environment includes a set of tools which simplify writing, modifying and testing code, as well as porting PUNDIT to new domains. These include: a set of global switches to tailor the operation of PUNDIT and the generation of trace messages to the specific module of the system one is working on, a general Prolog structure editor that facilitates traversing and editing Prolog terms, a lexical entry procedure that assists in the task of creating new lexical entries, and a semantic rule editor that assists in the creation of consistent semantic rules.

6.3 Testing the System

We have implemented a set of procedures to automate the updating and testing of PUNDIT. In order to make revisions to PUNDIT, files containing updated code are deposited in a special directory, and the whole system is re-built. Next, PUNDIT processes the entire corpus of messages from each domain, and the resulting output is automatically verified to ensure consistency with the expected, stable output. Any discrepancies between the expected and actual output are flagged so that changes can be made to the update files. This testing procedure has proved invaluable for maintaining the coverage and integrity of the system, and for guarding against unforeseen interaction of multiple changes to the system.

7 Implementation

PUNDIT is currently implemented in Quintus Prolog 2.2, and has been ported to TI-Prolog and C-Prolog. The system can run on any hardware supporting one of these Prologs, and has been demonstrated on Vaxes, Suns, Symbolics, Xeroxes, and Explorers. In a typical domain (CASREPS), the PUNDIT system includes approximately 20,000 lines of code, which make up 2000 predicates, containing 6000 clauses distributed among 80 files. Typical sentence-processing times on a Sun 3/60 are in the range of 5–10 seconds for sentences of average length (approximately 10 words); discourse-processing times for entire messages vary from about 15 seconds for short messages to 45–60 seconds for longer and more complicated texts.

References


