Natural Language Engineering **1** (1): 1–17. Printed in the United Kingdom

Parser Engineering and Performance Profiling

STEPHAN OEPEN
Computational Linguistics, Saarland University,
Im Stadtwald, 66123 Saarbrücken, Germany
e-mail: oe@coli.uni-sb.de

JOHN CARROLL
Cognitive and Computing Sciences, University of Sussex,
Brighton BN1 9QH, UK
e-mail: johnca@cogs.susx.ac.uk

(*Received 3 November 1999; revised 16 February 2000*)

Abstract

We describe and argue for a strategy of performance profiling and comparison in the engineering of parsing systems for wide-coverage linguistic grammars. A performance profile is a precise, rich, and structured snapshot of system (and grammar) behaviour at a given development point. The aim is to characterize system performance at a very detailed technical level, but at the same time to abstract away from idiosyncracies of particular processors.

Profiles are obtained with minimal effort by applying a specialized profiling tool to a set of structured reference data (taken from both existing test suites and corpora), in conjunction with a uniform format for test data and processing results. The resulting profiles can be analyzed and visualized at various levels of granularity in order to highlight different aspects of system performance, thus providing a solid empirical basis for system refinement and optimization. Since profiles are stored in a database, comparison with earlier versions, different parameter settings, or other processing systems is straightforward.

We apply several salient performance metrics in a contrastive discussion of various (one-pass, bottom-up, chart-based) parsing strategies (viz. passive vs. active and uni- vs. bidirectional approaches). Based on insights gained from detailed performance profiles, we outline and evaluate a novel 'hyper-active' parsing strategy. We also present preliminary profiles for techniques for 'packing' of local ambiguities with respect to (partial) subsumption of feature structures.

1 Background

> [...] *we view the discovery of parsing strategies as a largely experimental process of incremental optimization.* (Erbach, 1991a)

Until recently, the primary test corpus used in the development of the LinGO HPSG grammar (Flickinger & Sag, 1998) was the '*csli*' test suite (see the introduction to this volume). This test suite was designed to contain as little ambiguity as possible so that each sentence would exercise the grammar with respect to a single linguistic phenomenon or type of interaction between phenomena. The test suite therefore necessarily contains only short sentences. Extensions to grammatical coverage have been driven in part by the requirements of the VerbMobil project (Wahlster, 1997), which has provided word-level transcriptions of human-human dialogs as a development corpus. The majority of sentences in this corpus are also relatively short.

Carroll (1994) argues that parsers developed in tandem with large unification-based grammars often become tuned to the grammar concerned, in that the processing algorithms adopted are optimised for the particular grammar. It is likely that the LinGO grammar has had this effect on the LKB development environment (Copestake, 1992) parser—and also to some extent on the other parsers described in this volume. In addition, the tuning has extended to the type of input expected since the test suites and development corpora typically contain only short sentences.

Although the worst-case time complexity of parsing with current linguistically-motivated grammar formalisms such as HPSG and LFG is intractable, some progress has been made in improving average case complexity by devising parsing strategies which factor the search space intelligently (e.g. the combination of top-down and bottom-up control in van Noord (1997), and the contexted representation of disjunctive information in Maxwell and Kaplan (1995)). Such strategies have only just started to be investigated for parsing with the LinGO grammar (and HPSG implementations in general), since:

- the large size of grammars, and particularly of feature structures, means that primitive operations such as unification are potentially very expensive; this has led to the development of techniques for compilation of feature structure operations into abstract machine instructions (Miyao, Makino, Torisawa, and Tsujii, this volume), and pre-unification filtering and minimisation of feature structure copying (Malouf, Carroll, and Copestake, this volume); and
- the concentration of grammatical information in the lexicon and the lack of a context-free phrase structure backbone weakens the power of top-down prediction; thus most work in this area on parsing strategies has been concerned with refining purely bottom-up techniques;[1] an advantage of this, in the context of VerbMobil for example, is that in the absence of top-down constraints all complete constituents are available for robust partial parsing (van Noord (1997) achieves a similar effect through underspecification of the initial parse goal).

Parsing to support grammar development needs to be fast only for short strings, but real-world applications may require fast processing of longer utterances, such as occur in newspaper text, and more ambiguous input, such as word lattices produced by a speech recogniser. Unfortunately, no analytical technique exists that can adequately characterize grammar complexity in a practical setting or can predict

[1] An exception is the work reported by Torisawa, Nishida, Miyao, and Tsujii (this volume) on parsing with an automatically extracted context-free backbone, which then supplies predictive information on feature-based constraint application in a second phase.

from a given grammar what the best parsing strategy for it might be. We therefore need to use an empirical approach to identifying good parsing strategies. In the rest of this article we describe such an approach (Section 2), profile the performance of a number of parsers using the LinGO grammar (Section 3), use evidence from the profiling to synthesise a novel 'hyper-active' parsing strategy (Section 3.3), and conclude with preliminary experimental results for a parsing regime that incorporates local ambiguity packing based on (partial) subsumption of feature structures (Section 4).

2 Performance Profiling

In system development and optimization, subtle algorithmic and implementational decisions often have a significant impact on system performance, so the ability to monitor system evolution very closely is crucial. Developers should be enabled to obtain a precise record of the status of the system at any point; also, comparison with earlier results and between various parameter settings should be automated and convenient. System performance, however, cannot be adequately characterized merely by measurements of overall processing time (and perhaps memory usage). Properties of (i) individual modules (in a classical setup, especially the unifier, type system, and parser), (ii) the grammar being used, and even of (iii) the input presented to the system all interact in complex ways. In order to obtain a good understanding of the strengths and weaknesses of a particular configuration, finer-grained records are required. By the same token, developer intuition and isolated case studies are often insufficient, since in practise, people who have worked on a particular system or grammar for years still find that an intuitive prediction of system behaviour can be incomplete or plainly wrong.

Most grammar development environments supply facilities to batch-process a test corpus and record the results produced by the system. However, these systems are typically restricted to processing a flat, unstructured input file (listing test sentences, one per line) and outputing a small number of processing results into a log file. (Meta-)Systems like PLEUK (Calder, 1993) and HDRUG (van Noord & Bouma, 1997) that facilitate the exploration of multiple descriptive formalisms and processing strategies come with slightly more sophisticated benchmarking facilities and visualization tools. However, they still largely operate on monolithic, unannotated input data sets, restrict accounting of system results to a small number of parameters (e.g. number of analyses, overall processing time, memory consumption, possibly the total number of chart edges), and only offer a limited, predefined choice of analysis views.

Oepen and Flickinger (1998) propose a methodology which they term *grammar profiling* that builds on structured and annotated collections of test and reference data (traditionally known as *test suites*) and an adaptation to grammar engineering of the profiling metaphor familiar in software development. The *performance profiling* approach we elaborate in this article can be viewed as a generalization of this methodology to the development, refinement, and empirical evaluation of constraint-based processing systems—in line with the experimental paradigm sug-

i-length	length of test item in words; see Oepen, Netter, and Klein (1997)
readings	number of complete analyses obtained; when applicable, after unpacking
ftasks	number of argument instantiations filtered prior to execution
etasks	number of attempts to instantiate an argument position in a rule
filter[3]	percentage of parser actions predicted to fail: *ftasks* / (*etasks* + *ftasks*)
stasks	number of successful instantiations of argument positions in rules
pedges	number of passive edges built by the parser (typically in all-paths search)
unifications	number of top-level calls into the feature structure unification routine
copies	number of top-level feature structure copies made
tcpu	amount of cpu time (in milliseconds) spent in processing
space	amount of dynamic memory allocated during processing (in bytes)

Table 1. *Some of the parameters making up a competence & performance profile.*

gested by, among others, Erbach (1991a) and Carroll (1994). We define a *competence & performance profile* as a rich, precise, and structured snapshot of system behaviour at a given development point. The production, maintenance, and inspection of profiles is supported by a specialized software package (called [incr tsdb()][2]) that supplies a uniform data model, application program interface to the grammar-based processing components, and facilities for profile analysis and comparison. Profiles are stored in a relational database which accumulates a precise record of system evolution, and which serves as the basis for flexible report generation, visualization, and data analysis via basic descriptive statistics. All tables and figures used in this article, as well as most of the data collection for other contributions to this volume, were generated using [incr tsdb()].

We have defined a common set of descriptive metrics which aim both for in-depth precision and also for sufficient generality across a variety of processing systems. Most parameters are optional, though analysis potential may be restricted for partial profiles. Roughly, profile contents can be classified into information on (i) the processing *environment* (grammar, platform, versions, parameter settings and others), (ii) grammatical *coverage* (number of analyses, derivation and parse trees per reading, corresponding semantic formulae), (iii) *ambiguity* measures (lexical items retrieved, number of active and passive edges, where applicable, both globally and per result), (iv) *resource consumption* (various timings, memory allocation), and indicators of (v) *parser* and *unifier* throughput. Excluding relations and attributes that encode annotations on the input data, the current *competence & performance* database schema includes some one hundred attributes in five relations. Table 1 summarizes some of the parameters relevant to the discussion of parsing strategies in this article and to other contributions in this volume; individual parameters will be discussed in more detail in the sections to come.

[2] See '`http://www.coli.uni-sb.de/itsdb/`' for the (draft) [incr tsdb()] user manual, pronunciation rules, and instructions on obtaining and installing the package.

[3] Note that our definition of the pre-unification *filter* rate differs from that used by Kiefer, Krieger, Carroll, and Malouf (1999): in the denominator they subtract the number of

While the current [incr tsdb()] data model has already been successfully adapted to six different parsing systems (with another two pending; see Section 5), it remains to be seen how well it scales to the description of a larger variety of processing regimes. And although absolute numbers must be interpreted with care, the common metric has already increased comparability and data exchange among several processing platforms and sites, and has in some cases also helped to identify unexpected sources of performance variation. For example, we have found that two Sun UltraSparc servers with identical hardware configuration (down to the level of cpu revision) and OS release reproducably exhibit a performance difference of around ten per cent. This appears to be caused by different installed sets of vendor-supplied operating system patches. Also, average cpu load and availability of main memory have been observed to have a noticeable effect on cpu time measurements; therefore, the data reported in this article, was collected in an (artificial, in some sense) environment in which sufficient cpu and memory resources were guaranteed throughout each complete test run. Finally, in the Lisp-based LKB and PAGE environments garbage collection (gc) creates an additional element of noise. Firstly, gc frequency and efficiency are highly dependent on how the basic Lisp system is configured, and especially on the overall process size and tuning of the garbage collection strategy. Secondly, it seems there is a penalty on non-gc time after a garbage collection has finished; this may be due to rehashing after object relocation or other system-internal bookkeeping, and requires further investigation.

3 Analyzing and Comparing Parser Strategies

In the following sections we apply the performance profiling methodology to a variety of parsing strategies and show how the analytical parameters sketched earlier can be used in the contrastive study of different algorithms. Sections 3.1 to 3.3 review a development cycle that resulted in a refinement of traditional parsing algorithms (the 'hyper-active' parser) that was found beneficial in both all-paths (typical in grammar development) and best-first modes (in time-critical applications like VerbMobil). Hyper-active parsing is now the standard algorithm for the majority of processing environments represented in this volume.

3.1 Starting Point: LKB *vs.* PAGE

We start with an empirical comparison of the performance of a passive and an active chart parser, the former implemented in the LKB (by Ann Copestake and the second author) and the second in PAGE (by Bernd Kiefer). While collaboration

successful unifications (*stasks*), giving a measure relating filter successes to the number of unifications that ultimately fail. They therefore obtain seemingly higher filter rates since their baseline is what a filter could maximally achieve, i.e. one hundred per cent if all failures could be predicted. In contrast, our definition of *filter* gives the percentage of (potential) parser actions that were not executed, providing a slightly more direct handle on the share of the total search space that was not explored using (full) unification. Although both definitions can be justified, there is unfortunately potential for confusion.

and exchange between the LKB and PAGE developers have already resulted in homogenization of approaches and individual modules (the conjunctive PAGE unifier, for instance, was developed at CSLI Stanford), the parsing regimes deployed in the two systems differ significantly. Both parsers are implemented in Common Lisp, use the Tomabechi (1991) quasi-destructive unification algorithm with the copy operation structure sharing improvements of Malouf et al. (this volume), are purely bottom-up, compute all possible analyses, and perform no ambiguity packing (but see section 4 below). Before any unification is attempted, both parsers apply the same set of pre-unification filters, viz. a test against a static rule compatibility table (Kiefer et al., 1999), and the 'quick check' partial unification test (Malouf et al., this volume).

The LKB passive chart parser uses a breadth-first CKY-like algorithm; it processes the input string strictly from left to right, constructing all possible complete constituents whose right vertex is at the (current) right boundary of the chart before moving on to the next lexical item. All complete constituents found are entered in the chart. Attempts at rule application are made from right to left (i.e. rightmost daughter first). All daughter unifications must succeed before the copy operation on the rule mother is attempted. The basic strategy is similar to that used by ALE, version 3.2 (Penn & Carpenter, 1999).

The active chart parser implemented in PAGE uses a variant of the algorithm sketched by Erbach (1991b). It operates bidirectionally, both in processing the input string and instantiating rules; crucially, the *key* daughter (see Section 3.2 for details) of each rule is analyzed first, before the other daughter(s) are instantiated. As all active edges are added to the chart, the parser must perform a copy operation after each successful unification.

Both parsers have been used in large-scale grammar development for quite a while and have been engineered and optimized to a similar degree. However, while the LKB and the PAGE developers both assumed the strategy chosen in their own system was the best-suited for parsing with large feature structures (as exemplified by the LinGO grammar), the choices are motivated by conflicting desiderata. Not storing active edges (as in the passive LKB parser) reduces the amount of feature structure copying but requires frequent recomputation of partially instantiated rules, in that the unification of a daughter constituent with the rightmost argument position of a rule is performed as many times as the rule is applied to left-adjacent sequences of candidate chart edges. Creating active edges that store partial results to the chart, on the other hand, requires that more feature structure copies are made, which in turn avoids the necessity of redoing unifications. However, given the effectiveness of the pre-unification filters it is likely that for some active edges no attempts to extend them with adjacent inactive edges will ever be executed, so that the copy associated with the active edge was wasted effort.

A contrastive summary of performance profiles (using the LinGO grammar and the '*csli*' test suite) for the two parsers is presented in Table 2. For each of the systems the table is broken down into two aggregates (divisions of the test set): test items that are rejected by the grammar (the '*readings* = 0' row), and items

'*csli*'	Aggregate	items ♯	filter %	etasks ϕ	stasks ϕ	pedges ϕ	time ϕ (s)	space ϕ (kb)
LKB	*readings* $= 0$	476	94·1	477	401	75	0·26	1745
	readings ≥ 0	817	94·2	764	645	125	0·43	2669
PAGE	*readings* $= 0$	470	95·5	160	140	78	0·29	3836
	readings ≥ 1	818	95·6	284	238	144	0·56	3738

Table 2. *Contrasting parser performance profiles: traditional* LKB *vs.* PAGE.[5]

for which at least one analysis was derived ('*readings* $\geq$ 1').[4] The performance profiles confirm that the two parsers solve roughly the same problem, in that they reject and accept almost identical subsets of test items, achieve comparable filter efficiency, derive a similar number of passive edges, and exhibit the property that it is computationally cheaper to reject an ungrammatical sentence than to enumerate all analyses for well-formed input. Also, overall cpu time performance is similar for both parsers. At the same time, the passive LKB parser executes dramatically more parser actions (*etasks*, i.e. attempts to instantiate an argument position of a rule by unification) while the active parser in PAGE requires significantly larger amounts of memory (*space*).

Both observed differences can be explained given knowledge of the parsing regimes deployed in the two systems. While the need to recompute partial analyses adds to the number of executed parser tasks (in the LKB), the reduced copying of feature structures results in lower memory consumption.[6] Thus, we can conclude that the LKB successfully trades unifications for copies, and that the two distinctly different parsing approaches achieve broadly equivalent performance.

However, one aspect in Table 2 requires further discussion: since grammar rules in LinGO are at most binary branching, the LKB exhibits an imbalance between the average number of passive edges built (*pedges*) and the number of parser tasks

[4] While the LKB and PAGE behave very similarly in most respects, minor system idiosyncracies—mostly in the treatment of inflectional morphology and irregular spellings—result in small differences for the aggregate sizes in Table 2; a third class, viz. items that were not parsed because of some processing error, typically because of missing vocabulary, is omitted. Also, PAGE generates a few additional passive edges (the *pedges* column) because it does not perform run-time type inference (i.e. the application of constraints associated with a greatest lower bound that is computed at run-time) as is done in the LKB. However, with the LinGO grammar and the types of profiling we are reporting these minor differences can be ignored.

[5] Performance measures, here and in subsequent tables, were obtained on a 300 mhz UltraSparc with 1.2 gbyte main memory, running Solaris 2.6. Throughout the manuscript we use the symbol '♯' in table headings to indicate absolute numbers, while 'ϕ' denotes average values.

[6] See (Malouf et al., this volume) for details on why unification does not allocate larger amounts of memory by itself. As will be discussed below, the choice of parsing strategy crucially interacts with properties of other system components—like the unifier and type system—and cannot be studied in isolation.

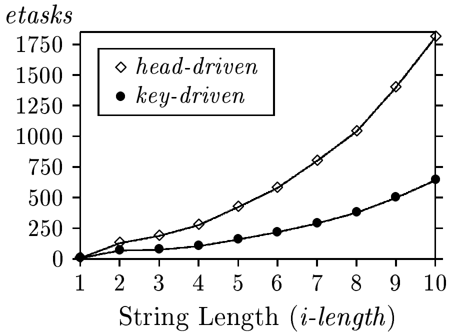
(i.e. unifications) that succeed (*stasks*); the average ratio of five successful unifications per edge clearly suggests that a large number of attempts to instantiate a grammar rule only fail when the second argument is instantiated. This is due to the unidirectional rule instantiation strategy used in the LKB parser. Whenever the input pointer is advanced to the right, the parser relies on the left corner of the chart being complete (i.e. it contains all passive edges that can be derived for the corresponding prefix of the input string). While applying rules to a new edge e, the parser can thus restrict rule postulation to edges left-adjacent to e, only using e in the rightmost daughter of each rule.[7] The following section elaborates the importance of bidirectionality and the right choice of *key* daughters that are instantiated first in active parsing (although it constitutes a slight digression from the active vs. passive comparison, it provides necessary background information for subsequent sections).

3.2 Bidirectional Active Parsing

Kay (1989) and Bouma and van Noord (1993) have argued for a (bidirectional) *head-driven* rule instantiation strategy, supplying supporting empirical evidence for a Categorial Unification Grammar (for English) and an HPSG-like DCG (for Dutch), respectively. Both grammars are strongly lexicalized. The approach, in general, builds on the observation that lexicalist grammars often employ rules containing more or less unspecific argument positions that are only constrained by coindexation with another argument in the rule; a generic head – complement schema, for example, imposes virtually no constraints on the complement daughter because the category of the daughter will be fully determined by selectional and governmental constraints imposed by the actual head feeding into the rule. Thus, instantiating a mostly unspecific argument position first will almost always succeed and in the case of the active parser create a large number of active edges that ultimately fail. The passive parser is less sensitive to this particular problem because it does not copy out partial analyses, and moreover because unification with an underspecified argument position (represented by a small feature structure) is likely to be computationally cheap.

Within the head-driven parsing paradigm, many authors implicitly (Kay, 1989) or explicitly (Bouma & van Noord, 1993) assume the linguistic head to be the argument position that the parser should instantiate first. However, no known techniques exist to analytically determine the right choice of the argument position in each rule such that it best constrains rule applicability (with respect to all categories derived by the grammar). Though that choice is likely to be related to the

[7] Without the left corner completeness condition on the chart, the parser would have to postulate rules with each argument position applied to a new edge; effectively, for a pair of adjacent edges, $\langle e_1 , e_2 \rangle$ say, all rules would be postulated twice, once when e_1 is derived and another time for e_2. The left corner completeness condition can only be maintained in breadth-first (or all paths) mode; therefore, the LKB parser when run in best-first mode has to instantiate rules bidirectionally which degrades performance significantly (see Figure 2 below).



	head-driven	key-driven	Δ
filter (%)	93·7	95·8	—
etasks	774	283	63·4 %
stasks	580	180	68·9 %
tcpu (s)	1·05	0·31	70·3 %
space (kb)	8932	2423	72·9%

Fig. 1. Effects of rule instantiation strategy (head- vs. key-driven) on parser work load.

amount and specificity of information encoded for each argument, for some rules a single feature value (e.g. the [WH +] constraint on the non-head daughter in one of the instantiations of the filler – head schema used in LinGO) can be very important. For terminological clarity, PAGE uses the term *key* daughter to refer to the argument position in each rule that is the best discriminator with respect to other categories that the grammar derives; thus, the notion of *key-driven* parsing emphasizes the observation that for individual rules in a particular grammar a non-(linguistic)head daughter may be a better candidate.

Figure 1 compares (active) parser performance for a rule instantiation strategy that always fills the (linguistic) head daughter first (labelled '*head-driven*' in both graphs) with a variant that uses an idiosyncratically chosen key daughter for each rule (termed '*key-driven*'; see below for how the keys are identified). The graphs show that the number of executed (*etasks*) as well as the number of successful (*stasks*) parser actions increase far more drastically with respect to input length in the head-driven setup (on the '*csli*' test suite, truncated above ten words due to sparse data). Since successful parser tasks are directly correlated to overall parser performance, the key-driven strategy on average reduces parsing time by more than a factor of three; clearly, for the LinGO grammar at least, linguistic headedness is not a good indicator for rule instantiation.

At this point, it should be clear that the choice of good parsing keys for a particular grammar is an entirely empirical issue. Key daughters, in the current setup, are stipulated by the grammar engineer(s) as annotations to grammar rules; in choosing the key positions, the grammarian builds on knowledge about the grammar—especially on what is known about the feeding relation between rules—and observations from parsing test data. The performance profiling tools can help in this choice since they allow the accounting of active and passive edges to be broken down by individual grammar rules (as they were instantiated in building edges). Inspecting the ratio of edges built per rule, for any given choice of parsing keys, can then help to identify rules that generate an unecessary number of active edges. Thus, in the experimental approach to grammar and system optimization the effects of

	Parser	filter %	etasks ϕ	stasks ϕ	unifs ϕ	copies ϕ	tcpu ϕ (s)	space ϕ (kb)
csli	*passive*	94·2	658	555	663	114	0·37	2329
	active	95·8	283	180	288	180	0·31	2432
	hyper-active	95·8	283	180	354	114	0·27	1686
aged	*passive*	94·2	1843	1604	1845	293	1·11	5692
	active	96·1	716	452	718	452	0·87	5449
	hyper-active	96·1	716	452	928	293	0·71	3830

Table 3. *Close-up of passive, active, and hyper-active parsing strategies in the* LKB.

different key selections can be analyzed precisely and compared to earlier results.[8] The remainder of this article assumes the key settings that are in effect in the July 1999 version of the LinGO grammar: exactly half of the twenty four binary rules are specified as key-first, the other twelve as key-final; out of the seventeen rules that actually assume a linguistic head daughter (the HPSG H-DTR feature), in only six does the key daughter correspond to the linguistic head.

3.3 A Synthesis: Hyper-Active Parsing

Returning to the LKB vs. PAGE comparison, we observed that the passive LKB parser appeared to successfully trade unifications for copies. To obtain fully comparable results, we imported the key-driven version of the PAGE active parser into the LKB, and use the LKB as the (single) experimentation environment for the remainder of this article. The direct comparison is shown in Table 3 for the '*csli*' and '*aged*' standard test sets. While the overall picture from the LKB vs. PAGE comparison is confirmed, the re-implementation of the active parser in the LKB in fact performs slightly better than the passive version and does not allocate very much more space. On the '*aged*' test set, the active parser even achieves a modest reduction in memory consumption. This is because for this test set the passive parser carries out a larger proportion of extra unifications compared to the savings in copies (columns five and six).

Having profiled the two traditional parsing strategies and dissected each empirically, one could ask whether it is possible to synthesize a new algorithm combining the good points of both strategies (i.e. reduced unification *and* reduced copying). The answer is yes, and we outline such a synthesis below which we term *hyper-active* parsing:

[8] For a given test corpus, the optimal set of key daughters could be determined (semi- or fully automatically) by comparing results for unidirectional left to right to pure right to left rule instantiation; the optimal key position for each rule is the one that generates the smallest number of active items. When this optimization was performed on the LinGO grammar for the first (and only) time, it confirmed the choices of the grammar writer in all but one case.

(1) use the bottom-up, bidirectional, key-driven control strategy of the active parser as sketched in Section 3.1;
(2) when an 'active' edge is derived (i.e. unification with an argument position in a rule has succeeded), store this partial analysis in the chart but do *not* copy the associated feature structure;[9]
(3) when an 'active' edge is extended (combined with a passive edge), recompute the intermediate feature structure from the original rule and already-instantiated daughter(s);
(4) only copy feature structures for complete passive edges; partial analyses are represented in the chart but the unification(s) that derived each partial analysis are redone on-demand.

Storing 'active' (or, in a sense, hyper-active) edges without creating expensive feature structure copies enables the parser to perform a key-driven search effectively, and at the same time avoid overcopying for partial analyses; additional unifications are traded for the copies that were avoided only where hyper-active edges are actually extended in later processing. For the medium-complexity '*aged*' test set, each hyper-active edge is extended (necessitating recomputation of the intermediate structure) on average less than twice. It would therefore be desirable to take advantage of the intermediate feature structure—even though it is not copied—at the point at which it is available. So when a hyper-active edge is derived, the parser deviates slightly (in the form of an excursion) from its normal agenda-driven control strategy: while the intermediate structure is still valid, one attempt to combine it with a suitable (adjacent) passive edge is made in the same unification generation.[10]

Table 3 confirms that hyper-active parsing combines the desirable properties of both basic algorithms: the number of copies made is exactly the same as for the passive parser, while the number of unifications is only a little higher than for the active parser (due to on-demand recomputation of intermediate structures). Accordingly, average parse times are reduced by twenty seven ('*csli*') and thirty six ('*aged*') per cent, while memory consumption drops by twenty eight and thirty three per cent, respectively. The benefits of hyper-active parsing are thus more apparent on the more complex data set. The observed differences in relative improvement for the two data sets confirm the claim made earlier that performance profiling on the basis of a small set of test data is limited in analytical potential.

To see how the hyper-active parsing regime scales up to the (currently) most challenging reference corpus available for the LinGO grammar (see the introduction to this volume), Table 4 compares parser performance on the '*blend*' test set

[9] Although the intermediate feature structure is not copied, it is used to compute the 'quick check' vector for the next argument position to be filled; as was seen already, this information is sufficient to filter the majority (in fact up to 95 per cent) of subsequent operations on the 'active' edge.

[10] Where the excursion is successful, one iteration in the main parser loop now results in the creation of two new edges: the hyper-active edge to start with, and a passive edge derived from the excursion. In any case, the one passive edge explored during the excursion is recorded with the hyper-active edge to avoid redoing the same combination later when regular processing of that edge is scheduled.

Aggregate	items	passive tcpu	passive space	hyper-active tcpu	hyper-active space	reduction tcpu	reduction space
	♯	ϕ (s)	ϕ (kb)	ϕ (s)	ϕ (kb)	%	%
$5000 \leq pedges \leq 19690$	112	50·59	213402	30·76	119453	39·2	44·0
$1000 \leq pedges < 5000$	337	8·55	32142	5·12	15198	40·1	52·7
$500 \leq pedges < 1000$	253	2·55	8017	1·60	3378	37·4	57·9
$250 \leq pedges < 500$	297	1·18	3923	0·80	1781	32·6	54·6
$0 \leq pedges < 250$	939	0·25	918	0·20	468	22·6	49·0
Total	**1938**	**5·05**	**20014**	**3·09**	**10487**	**38·7**	**47·6**

Table 4. *Comparison of passive vs. hyper-active strategies on 'blend' test set.*

for the original LKB passive parser and for the hyper-active parser. Here, the contrastive performance profile is aggregated by the number of passive edges derived (a parameter that, in our non-predictive bottom-up paradigm, does not vary with the parsing strategy), providing a very direct measure on input complexity. The results indicate that the effectiveness of hyper-active parsing actually increases with input complexity; for the top two aggregates, hyper-active parsing reduces average parsing times by around forty per cent. Taking into account the fact that the *tcpu* measurements exclude garbage collection time (which increases in direct proportion to memory consumption), a net speed-up of close to a factor of two is achieved in actual parser throughput. As for memory usage, Table 4 shows a bigger *space* reduction than was found in earlier experiments. When processing the '*blend*' corpus, another optimization (adapted from the PET system; see Callmeier, this volume) was enabled: the system maintains a statically allocated pool of *dag* objects that can be recycled cheaply after a parse has completed and that therefore do not show up in the dynamic *space* allocation figures. When dag creation is replaced by recycling, memory consumption primarily reflects structure allocated during unification (temporary arcs) and in the parser (edges); hence, *space* reduction in Table 4 corresponds closely to the overall decrease in the number of unifications.[11] In an application such as VerbMobil—containing several concurrent modules each of which may have a process size of a few hundred megabytes—savings in memory may actually be more important than improvements in throughput.

Finally, we noted in Section 3.1 that the passive parser is not well-suited for priority-driven best-first search (as is standard practise in applications like VerbMobil) since the left corner completeness condition on the chart cannot be maintained. This expectation is confirmed by the graphs in Figure 2, showing best-first parse

[11] The recycling pool used in the LKB has a fixed size, though. Once the pool is exhausted the system falls back into regular allocation mode (creating garbage); therefore, the reduction in memory usage reaches its maximum in the third class in Table 4 and then decreases for the top two aggregates where the dag pool is typically exhausted during the course of each parse.

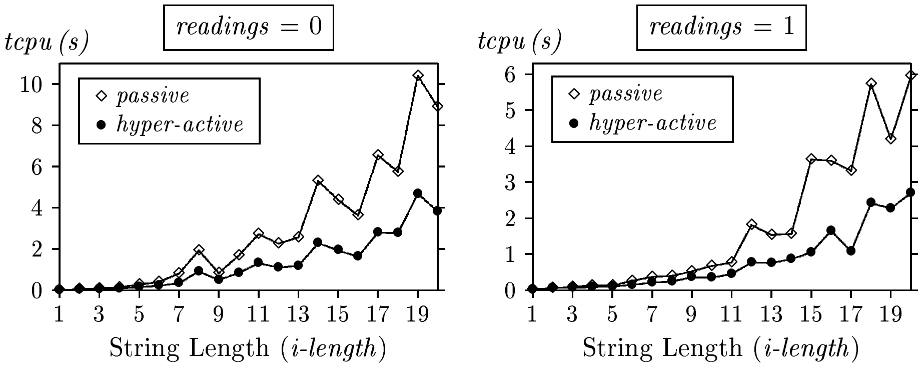


Fig. 2. Parsing times for passive vs. hyper-active strategies in first-only mode ('*blend*').

times for ungrammatical ('*readings* = 0') and grammatical ('*readings* = 1') items from the '*blend*' corpus. In contrast to exhaustive mode, finding just one analysis is significantly cheaper than rejecting ungrammatical input.[12] However, for both grammaticality classes parse times are reduced by more than a factor of two over passive mode, with the times increasing only slowly as sentences get longer. The hyper-active parser, although it started out as a vehicle for experimentation with parsing strategies, is now in regular use for grammar development and system optimization in the LKB, and has been ported to both the PAGE and the PET systems.

4 Packing Ambiguity: Preliminary Results

The passive, active, and hyper-active parsers add every completely recognised constituent to the chart as a separate edge. The parsers are therefore explicitly constructing a possibly (depending on the grammar and input string) exponential number of analyses. One approach to this problem is to remove from the grammar sources of exponential behaviour such as prepositional phrase attachment ambiguities, perhaps by allowing only a single attachment point in the phrase structure tree and creating an underspecified semantics which represents all of the alternatives. An orthogonal, computational approach is to 'pack' local ambiguities in the parser itself, such that if a constituent can be analysed in more than one way the different structures are packed into a single representation. Alshawi (1992) and Carroll (1993) have investigated this technique for unification grammars, using a feature structure subsumption test to check if a newly derived constituent can be packed.

If the newly derived constituent is equivalent to or subsumed by an existing

[12] As is demonstrated by Kiefer et al. (1999), it may be necessary to impose an upper limit—the average parse time for finding one reading as a function of input length, say—on the amount of time available to the parser. Also, the priority computation used in Figure 2 is naïve in that it only deals with simple and *local* scores attached to lexical entries and grammar rules; again, the VerbMobil experience suggests that more sophisticated scoring, incorporating various knowledge sources, is necessary in practical applications.

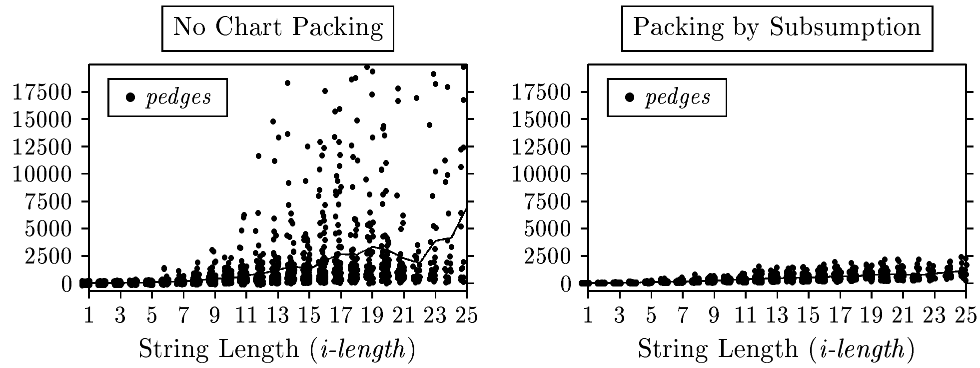


Fig. 3. Effects of maximal ambiguity packing based on (partial) subsumption ('*blend*').

constituent, then it can be packed into the existing one and will take no further part in processing. However, if the new constituent *subsumes* an existing one, the situation is not so straightforward: either (i) no packing takes place and the new constituent forms a separate edge, or (ii) previous processing involving the old constituent is undone or invalidated, and it is packed into the new one. In the former case the parse forest produced will not be optimally compact; in the latter it will be, but a potentially large amount of processing may end up being redone.

To explore the applicability of ambiguity packing to HPSG-type grammars, the hyper-active LKB parser was extended to incorporate the efficient subsumption test described by Malouf et al. (this volume). In order for the subsumption relation to apply meaningfully to HPSG signs, two conditions must be met. Firstly, parse tree construction cannot be duplicated in the feature structures (by means of the HPSG `DTRS` feature) but is left to the parser (i.e. recorded in the chart); this is standardly achieved by feature structure restriction (see Shieber, 1985, and Kiefer et al., 1999). Secondly, semantic composition (performed via the HPSG attribute `CONT`) must be delayed until a complete analysis is derived; in general, it seems desirable to postpone processing of constraints that build up new structure but do not restrict the search space.[13] Again, this requirement is fulfilled by restricting the structures, though in this case it is sufficient to remove all such attributes (temporarily) in lexical entries as they are input to the parser and in the rule set used during the first parsing phase.

Given a pair of edges e_1 and e_2, the subsumption test in a single pass determines whether e_1 subsumes e_2 or vice versa; trivially, mutual subsumption indicates equivalence of e_1 and e_2. When a new passive edge is derived, the parser tests subsumption against all chart entries that span the same portion of the input

[13] For the current reference version of the LinGO grammar, `CONT` unfortunately still embeds a small number of constraints that in rare cases can restrict rule applicability. This leads to overgeneration in the (first) recognition phase, but does not affect overall correctness of the parser because (currently) we validate all packed results in a second, unpacking phase by applying the complete constraint sets.

string. Depending on whether (i) a new edge e_1 is subsumed by an existing edge e_2 or (ii) the new edge e_1 subsumes an existing edge e_2, the parser distinguishes between (i) *proactive* (or forward) packing and (ii) *retroactive* (backward) packing. While proactive packing simply excludes e_1 from all further processing, retroactive packing needs to invalidate the existing chart edge e_2, everything that was derived from e_2, and all pending computation involving e_2 and its derivatives. However, chart entries that are invalidated in retroactive packing can in principle already host packed analyses; hence, while a host edge is invalidated, edges packed into it still represent valid analyses and need to be repacked into a new (more general) host edge derived from e_1 (once those new derivatives become available). In short, retroactive packing requires specialized and efficient accounting mechanisms in the chart *and* agenda. These are described in detail by Oepen and Carroll (2000).

Figure 3 presents first results achieved when both pro- and retroactive packing are enabled in the parser. Compared to regular parsing, the average number of passive edges is reduced by a good factor of three, while for a number of cases it drops by factors of thirty and more. Without unpacking the parse forest, a similar reduction in parse times is achieved, again not including reduced garbage collection time. Even with short sentences and little ambiguity no measurable penalty for the extra subsumption tests is observed. When the validation (unpacking) of the complete parse forest is included in the accounting, the packing parser still achieves a time *and* space reduction of fifty to seventy percent on medium-length input (ten to fifteen words). Finally, we note that nearly two thirds of the packings done by the parser involve equivalent feature structures; and using a simple breadth-first parsing strategy, the majority of non-equivalent packings require retroactive packing. Given these preliminary results, we conclude that subsumption-based local ambiguity packing can be successfully applied to parsing with HPSG grammars, and that we can gain dramatic reductions in chart size and achieve tractable average-case complexity.

5 Future Directions

We intend to develop the competence and performance profiling work described above in a number of different directions.

Firstly, by integrating [incr tsdb()] with further processing systems (connections to the Alvey Tools GDE and the Xerox XLE system are currently under consideration) we hope to gain a better understanding of the types of metrics that are useful for grammar and parser engineering. We also want to gauge the generality of the hyper-active parsing strategy by profiling with other systems and grammars; we have obtained some encouraging preliminary results for all three VerbMobil grammars (English, German, and Japanese) in the PAGE system.

Secondly, we will continue with the parser engineering line of research, informed by regular performance profiling and analysis. We will continue to develop the hyper-active parser and investigate techniques that reduce the cost of replaying unifications that are known to succeed. Representing unification results as environments (Pereira, 1985) that only record changes made to the original structures could be of benefit in this context.

Finally, we plan to run a series of experiments contrasting subsumption- vs. equivalence-based packing strategies, and explore techniques for prioritized computation to (i) try to minimize the amount of retroactive packing required and (ii) limit unpacking from the parse forest to incremental retrieval of preferred analyses. At the same time, better fine tuning to properties of the grammar may be required: if parts of the HPSG sign could be guaranteed to never restrict the search space, validation of substructures packed under equivalence would be unnecessary.

Acknowledgements

The authors are grateful to Ann Copestake and Bernd Kiefer (who implemented the original parsers in the LKB and PAGE, respectively) for many useful discussions about the algorithms used and the underlying reasoning. Also, Dan Flickinger and Ulrich Callmeier have been most helpful in discussing dependencies between various parsing strategies and properties of the LinGO grammar. Three anonymous reviewers gave detailed and constructive comments. The research reported in this article was supported by the *Deutsche Forschungsgemeinschaft* as part of the Collaborative Research Division *Resource-Adaptive Cognitive Processes* (SFB 378) project B4 (PERFORM), and by a UK EPSRC Advanced Fellowship to the second author.

References

Alshawi, H. (Ed.). (1992). *The Core Language Engine.* Cambridge, MA: MIT Press.

Bouma, G., & van Noord, G. (1993). Head-driven parsing for lexicalist grammars. Experimental results. In *Proceedings of the 6th Conference of the European Chapter of the ACL (EACL)* (pp. 71 – 80). Utrecht, The Netherlands.

Calder, J. (1993). Graphical interaction with constraint-based grammars. In *Proceedings of the 3rd Pacific Rim Conference on Computational Linguistics* (pp. 160 – 169). Vancouver, BC.

Carroll, J. (1993). *Practical unification-based parsing of natural language* (Tech. Rep. No. 314). Computer Laboratory, Cambridge University, UK.

Carroll, J. (1994). Relating complexity to practical performance in parsing with wide-coverage unification grammars. In *Proceedings of the 31st Meeting of the Association for Computational Linguistics* (pp. 287 – 294). Las Cruces, NM.

Copestake, A. (1992). The ACQUILEX LKB. Representation issues in semi-automatic acquisition of large lexicons. In *Proceedings of the 3rd ACL Conference on Applied Natural Language Processing* (pp. 88 – 96). Trento, Italy.

Erbach, G. (1991a). An environment for experimenting with parsing strategies. In J. Mylopoulos & R. Reiter (Eds.), *Proceedings of IJCAI 1991* (pp. 931 – 937). San Mateo, CA: Morgan Kaufmann Publishers.

Erbach, G. (1991b). A flexible parser for a linguistic development environment. In O. Herzog & C.-R. Rollinger (Eds.), *Text understanding in LILOG* (pp. 74 – 87). Berlin, Germany: Springer.

Flickinger, D. P., & Sag, I. A. (1998). Linguistic Grammars Online. A multi-purpose broad-coverage computational grammar of English. In *CSLI Bulletin 1999* (pp. 64–68). Stanford, CA: CSLI Publications.

Kay, M. (1989). Head-driven parsing. In *Proceedings of the 1st International Workshop on Parsing Technologies* (pp. 52–62). Pittsburgh, PA.

Kiefer, B., Krieger, H.-U., Carroll, J., & Malouf, R. (1999). A bag of useful techniques for efficient and robust parsing. In *Proceedings of the 37th Meeting of the Association for Computational Linguistics* (pp. 473–480). Baltimore, MD.

Maxwell III, J. T., & Kaplan, R. M. (1995). A method for disjunctive constraint satisfaction. In M. Dalrymple, R. M. Kaplan, J. T. Maxwell III, & A. Zaenen (Eds.), *Formal issues in Lexical-Functional Grammar* (pp. 381–401). Stanford, CA: Cambridge University Press.

van Noord, G. (1997). An efficient implementation of the head-corner parser. *Computational Linguistics, 23 (3)*, 425–456.

van Noord, G., & Bouma, G. (1997). Hdrug. A flexible and extendible development environment for natural language processing. In *Proceedings of the EACL/ACL workshop on Computational Environments for Grammar Development and Linguistic Engineering (ENVGRAM)* (pp. 91–98). Madrid, Spain.

Oepen, S., & Carroll, J. (2000). Ambiguity packing in constraint-based parsing. Practical results. In *Proceedings of the 1st Conference of the North American Chapter of the ACL (NAACL).* Seattle, WA.

Oepen, S., & Flickinger, D. P. (1998). Towards systematic grammar profiling. Test suite technology ten years after. *Journal of Computer Speech and Language, 12 (4) (Special Issue on Evaluation)*, 411–436.

Oepen, S., Netter, K., & Klein, J. (1997). TSNLP — Test Suites for Natural Language Processing. In J. Nerbonne (Ed.), *Linguistic Databases* (pp. 13–36). Stanford, CA: CSLI Publications.

Penn, G., & Carpenter, B. (1999). ALE for speech: a translation prototype. In *Proceedings of the 6th Conference on Speech Communication and Technology (EUROSPEECH).* Budapest, Hungary.

Pereira, F. C. N. (1985). A structure-sharing representation for unification-based grammar formalisms. In *Proceedings of the 23rd Meeting of the Association for Computational Linguistics* (pp. 137–144). Chicago, IL.

Shieber, S. M. (1985). Using restriction to extend parsing algorithms for complex feature-based formalisms. In *Proceedings of the 23rd Meeting of the Association for Computational Linguistics* (pp. 145–152). Chicago, IL.

Tomabechi, H. (1991). Quasi-destructive graph unification. In *Proceedings of the 29th Meeting of the Association for Computational Linguistics* (pp. 315–322). Berkeley, CA.

Wahlster, W. (1997). *VerbMobil — Erkennung, Analyse, Transfer, Generierung und Synthese von Spontansprache* (VerbMobil Report # 198). Saarbrücken: Deutsches Forschungszentrum für Künstliche Intelligenz GmbH.