Natural Language Engineering **6** (1): 99–108. Printed in the United Kingdom

PET — *A platform for experimentation with efficient* HPSG *processing techniques*

ULRICH CALLMEIER
Computational Linguistics, Saarbrücken University,
Im Stadtwald, 66123 Saarbrücken, Germany
e-mail: uc@coli.uni-sb.de

(*Received 1 November 1999; revised 16 February 2000*)

Abstract

The PET platform for experimentation with HPSG processing and implementation techniques is introduced. PET provides an extendible set of building blocks for the implementation of efficient processors. This enables straightforward comparison of different approaches, rapid development of new techniques and easy synthesis of known techniques. An overview of the basic design is given and key aspects of the implementation are discussed. Early empirical results on the three standard test sets for the LinGO grammar are given. Two improved expansion strategies are evaluated empirically. The results of fine tuning the quick check filtering method through empirical study are discussed as an exemplar of the proposed experimental approach to the development and optimization of processing techniques. An overview of the progress in processing the LinGO grammar made over a development period of three years is given.

1 Introduction

This volume presents a range of new techniques in HPSG processing. The PET platform aims to integrate these techniques in a modular fashion, allowing the empirical study of practical performance and a contrastive comparison of different approaches and their interaction. This article discusses several engineering aspects of the PET implementation and reports on practical performance achievements.

Empirical study is indispensable for the evaluation and optimization of the practical performance of constraint-based processing systems. As Carroll (1994) argues, we do not yet have the analytic tools that would allow us to predict how the properties of individual unification-based grammars will interact with particular parsing algorithms.

The PET platform was developed with two main goals: (1) synthesizing the best current practice from the collaborative body of research reported in this volume, and (2) providing an extendible basis for the empirical evaluation of new approaches and experimentation with processing techniques. At the same time, PET demonstrates what practical performance can currently be achieved by synthesizing the results from several development streams.

2 The PET platform

PET is a platform to build processing systems based on the descriptive formalism presented in the Appendix (Copestake, this volume) and represented by the LinGO grammar. It aims to make answering questions about all aspects of processing easy, including comparison of existing techniques and evaluating new ideas. Thus, flexibility and extendibility were the main design objectives. This is achieved by a tool box approach—PET provides an extendible set of configurable building blocks that can be combined and configured in different ways to instantiate a concrete processing system. The set of building blocks includes objects like *chart*, *agenda*, *grammar*, *type hierarchy* and *typed feature structure*. For instance, a simple bottom-up chart parser can be implemented using the available objects in a few lines of code.

Alternative implementations of a certain object may be available to allow comparison of different approaches to one aspect of processing in a common context. For instance, there are currently three implementations of the *typed feature structure* object, one based on Wroblewski (1987), one based on 'into'-unification as applied in CHIC (Ciortuz, 2000), where only one of the input structures is destructively modified, and one based on Tomabechi (1991) with (optional) subgraph sharing improvements by Malouf, Carroll, and Copestake (this volume). In this setup properties of various graph unification algorithms and feature structure representations can be compared among each other and in interaction with various processing regimes. Experimentation with different feature structure representations, especially exploring WAM-inspired fixed arity encodings, is currently under way.

PET implements (and employs in the cheap parser) all relevant techniques from Kiefer, Krieger, Carroll, & Malouf (1999) (viz. conjunctive-only unification, rule filters, quick check, restrictors), as well as techniques developed in other systems, e.g. key-driven parsing from PAGE, caching type unification and hyper-active parsing (Oepen & Carroll, this volume) from the LKB and partial expansion from CHIC. Re-implementing techniques developed in other systems allowed for improved implementations, because previous implementation experience was available, and specific requirements could be accounted for in the design phase.

Efficient memory management and minimizing memory consumption was another important consideration in the development of the system. Experience with Lisp-based systems has shown that memory management is one of the main bottlenecks when processing large grammars. In fact, one observes a close correlation between the amount of dynamically allocated memory and processing time, indicating much time is spent moving data, rather than in actual computation.

Studying performance profiles of an early version of cheap that used the built-in C++ memory management supported this. Allocation and release of feature structure nodes was accounting for almost forty per cent of the total run time. However, like in the WAM (Aït-Kaci, 1991), a general memory allocation scheme allowing arbitrary order of allocation and release of structures is not necessary in this context. When parsing we typically continue to build up structures. Memory is only released in the case of a top-level unification failure when all partial structures built during this unification are released.

Therefore, PET offers an efficient and simple stack-based memory management strategy tailored to the specific needs of processing with large feature structures. In this allocation scheme, memory is acquired from the operating system in large chunks and then sub-allocated. There is no way to release individual objects; instead a *mark-release*-mechanism allows saving the current allocation state (the current stack position) and returning to that saved state at a later point. Thus, releasing a chunk of objects amounts to a single pointer assignment. Switching to this memory management implementation resulted in a significant overall speedup (a little less than a factor of 1.6) for cheap.

The implementation of Tomabechi's unification algorithm uses a very compact[1] representation of nodes. In combination with unfilling (section 3.2), subgraph sharing improvements (Malouf et al., this volume) and hyper-active parsing this results in very attractive memory consumption characteristics for the cheap parser.

PET provides an interface to the [incr tsdb()] system (Oepen & Flickinger, 1998). The common set of metrics defined by [incr tsdb()] greatly enhances comparability among different PET configurations as well as other systems.

Since testing hypotheses can require a large number of test runs on large sets of data, special attention was paid to efficiency and compactness when developing PET. Critical objects are carefully optimized. PET is implemented in ANSI C++, but uses traditional C representations (rather than C++ objects) in cases where minimal overhead is required, e.g. for the basic elements of feature structures.

2.1 The pre-processor

Short startup time is desirable for rapid experimentation. This is achieved in PET by pre-processing the source form of the grammar into a compact binary representation that can be loaded efficiently by the runtime system.

The pre-processor reads a grammar in $\mathcal{TDL}$ syntax (Krieger & Schäfer, 1994), expands $\mathcal{TDL}$ templates, constructs a lower semi-lattice from the type hierarchy[2] (using an efficient implementation of the theoretical construction from Aït-Kaci et al. (1989, section 3)), infers appropriateness conditions[3] and performs configurable expansion of type definitions (Krieger & Schäfer, 1995).

The LinGO grammar source, about 100,000 lines of $\mathcal{TDL}$, is pre-processed on a 500 megahertz Pentium III in 18.4 s, resulting in a 3,192 kb dump file that the runtime system loads in 0.7 s. In contrast, pre-processing and loading the grammar cannot be seperated in PAGE and LKB, where the whole process of reading the grammar takes 96.5 s and 48.8 s, respectively.

[1] The size of one node is only 24 bytes (compared to e.g. 48 bytes for the LKB system).

[2] Types are encoded using the transitive reflexive closure encoding from Aït-Kaci, Boyer, Lincoln, & Nasr (1989, section 4). At run time the computation of type intersection is cached in a hash table. Empirical evaluation showed this is as efficient as a full pre-computed table of greatest lower bounds as suggested in (Kiefer et al., 1999).

[3] This is using an algorithm previously implemented for the CHIC system by Liviu Ciortuz.

Test Set	Parser Tasks			CPU Time		Memory Usage		
	filter ϕ (%)	etasks ϕ	stasks ϕ	first ϕ (s)	tcpu ϕ (s)	space ϕ (Kb)	fssize[a] ϕ	process[b] (Mb)
‘csli’	95·0	287	160	0·01	0·03	416	117	24
‘aged’	95·2	756	428	0·02	0·08	1,162	133	24
‘blend’	95·7	2,956	1646	0·08	0·34	5,589	146	98

[a] Average number of nodes in feature structures of passive edges
[b] Maximum size of the Unix process when parsing the test set

Table 1. *Results for* cheap *on the standard test sets. All numbers are obtained on a 500 megahertz Pentium III with the reference version of the LinGO grammar. A set of quick check paths optimized for the ‘blend’ test set was used. The limit for the number of passive edges was set to 20,000. For a detailed explanation of the column headings refer to Oepen & Carroll (this volume).*

3 Current development status

3.1 Results for the standard test sets

Table 1 shows results obtained with the cheap parser on the three standard test sets (see the introduction to this volume) for the LinGO grammar. The cheap parser is a hyper-active, bidirectional key-driven, bottom-up chart-parser using PET’s Tomabechi unifier with subgraph sharing[4]. Using the [incr tsdb()] machinery all results (number of readings and passive edges; all derivation trees) have been compared with reference results obtained on the LKB system and yielded an exact match.

The increasing complexity of the test sets is demonstrated by the rise in number of executed (*etasks*) and successful (*stasks*) parser tasks from the *‘csli’* to the *‘blend’* test sets, and the correlating increase in average cpu time to parse each item (*tcpu*). There is a moderate increase in the filter rate (*filter*), since the set of quick check paths used was sampled on the *‘blend’* test set. The number of successful unifications (*stasks*) per second of parse time decreases from about 3,200 for the *‘csli’* test set down to about 2,500 for the *‘blend’* test set. This can be explained by the increasing average size of processed feature structures *fssize*. The table illustrates the close correlation between parsing time (*tcpu*) and memory consumption (*space*) quite well.

Comparison with LKB results from (Oepen & Carroll, this volume) indicates that cheap is about a factor of five[5] faster than the hyper-active parser in the LKB

[4] Please refer to Oepen & Carroll (this volume) and Malouf et al. (this volume) for a detailed discussion of these concepts.
[5] The numbers presented in (Oepen & Carroll, this volume) are sampled on a 300 megahertz UltraSparc; the factor of five was empirically confirmed by both obtaining cheap performance data on the same machine, and by running the LKB on the 500 megahertz Pentium.

while using the same basic algorithms. The process size is more than an order of magnitude smaller than that of comparable LKB or PAGE processes.

3.2 Partial expansion and unfilling

This section presents the evaluation of two simple, yet effective improvements over making all feature structures well-formed (also called 'expansion') prior to processing (as it is done in PAGE and LKB).

The first technique, known as *partial expansion*[6], was first explored and found beneficial for the LinGO grammar in the CHIC system (Ciortuz, 2000). Leaf nodes[7] in feature structures are only made well-formed when necessary at run time, that is when a leaf node is unified with a non-leaf node. This technique alone significantly reduces the size of the expanded grammar. As table 2 shows, the added cost for the delayed unification of constraints at run time is compensated by the reduced size of structures that the system manipulates, resulting in an overall performance improvement of about ten per cent. The increase in executed tasks is due to decreased quick check efficiency, because in the partially expanded structure quick check paths may not always be available (when their expansion has been delayed).

Unfilling (Götz, 1993; Gerdemann, 1995) goes a step further. After performing (partial) expansion, structures are shrunk again, by recursively removing leaf nodes from the structures. A leaf node under a feature f is removed if its type is the maximal appropriate type of f, and if this node does not introduce structure sharing. It is not removed on root level of the type introducing f. Table 2 shows that unfilling removed almost half the nodes after partial expansion for the LinGO grammar. Again the benefits of smaller structures outweigh the additional cost of expansion at run time significantly, resulting in a performance improvement of about twenty six per cent.[8] However, the LinGO grammar already employs a technique, namely the introduction of super-types with a minimal set of features (Flickinger, this volume), with effects similar to partial expansion and unfilling, to make processing in LKB and PAGE more efficient. This reduces the potential benefit of partial expansion and unfilling for LinGO.

The practical benefit one can expect from applying the partial expansion and unfilling techniques will highly depend on the architecture of the particular grammar and can only be determined empirically. Depending on the amount of partial expansion and unfilling that the grammar permits, the increased cost in run time expansion might even outweigh the benefits of smaller structures for some grammars. This, once again, emphasizes the importance of empirical study.

[6] Closely related lazy evaluation techniques are also discussed in Götz (1993), Carpenter & Qu (1995), Wintner (1997).

[7] Leaf nodes are nodes without any δ-descendants, see Copestake (this volume).

[8] A nice side-effect is that shrunk structures are also more humanly readable than fully expanded structures, because they are smaller, and the crucial pieces of information become more obvious.

	etasks ϕ	**tcpu** ϕ (s)	**space** ϕ (Kb)	**grammar size**[a] (nodes)	**process size** (Mb)
full expansion	623	0·12	2,031	611,920	42
partial expansion	756	0·11	1,662	442,608	34
unfilling	756	0·08	1,162	238,366	24

[a] total number of feature structure nodes for the expanded grammar, not including lexicon entries

Table 2. *Evaluating expansion strategies. Results for the 'aged' test set on a 500 megahertz Pentium III with the reference version of the LinGO grammar.*

3.3 A case study: fine tuning the quick check

PET provides an implementation of filtering by quick check (Malouf et al., this volume). At run time the set of quick check paths is represented in an annotated feature structure (as opposed to a list of paths in other implementations). This makes extraction of the quick check vectors computationally cheaper, as many of the paths have common prefixes.

The cheap parser can be configured to collect a set of quick check paths for a given test set. In general, this is done by recording all failure paths when parsing the test set using a modified unification algorithm that continues even after a failure is encountered. Then these paths are sorted by their respective effectiveness, and the best n paths are chosen. The next two sections will discuss finding a good measure for effectiveness of a path, and determining the number of paths to use.

3.3.1 Quick check path ordering

The most obvious measure for effectiveness of a failure path is its frequency of occurrence in parsing the test set. This corresponds to assigning a weight of 1 to each occurrence. We can successively improve on that measure.

1. When a failure occurs under n paths, assign each of them only a corresponding fraction of the weight—that is $1/n$.
2. Do not take into account failures that the quick check could not detect, by checking in the original structures if the information leading to the failure is already there. This is not always the case, as constraints may be unified in during unification, partially expanded paths might be expanded etc.
3. Make the weight dependent on the cost of finding that failure by full unification. We use the number of nodes visited (recursive calls to the unification function) as a measure for cost of unification. The idea is that some quick check paths only filter unifications that fail very soon, and paths which filter more expensive unifications should be favored. An obvious example is the empty path.

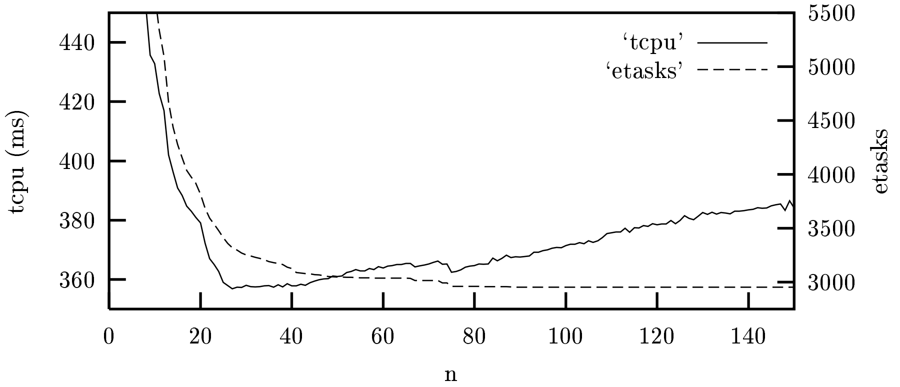

Fig. 1. Determining the number of quick check paths. The graph shows *tcpu* and *etasks* average values with the number of quick check paths ranging from zero to one hundred and fifty. The results are obtained with cheap on the '*blend*' test set running on a 500 megahertz Pentium III.

Evaluation of these measures on the '*blend*' test set demonstrates their effectiveness. The reduction from the base line (using paths computed with the naive measure) to the first measure is 8.5 per cent in parser tasks, and 2.1 per cent in parsing time. The second measure reduces parser tasks by another 7.6 per cent, and parsing time by another 3.8 per cent. The third measure does not improve much upon the previous ones: the reduction in parser tasks is another 0.8 per cent, the reduction in parsing time another 0.5 per cent.

3.3.2 Determining the number of paths to use

Malouf et al. (this volume) discuss the tradeoff in choosing the optimal number n of quick check paths. They conclude that n can not be determined analytically, and report about an experiment to determine n in the LKB for the LinGO grammar. However, for practical reasons, only a subset of the '*blend*' test set is used, and the variation of n is restricted to a number of support points for the graph.

Using PET we can run the experiment on the full '*blend*' test set trying all n in the range from zero to one hundred and fifty in reasonable time. Figure 1 shows the result of this experiment. The results for zero to ten paths, where parsing time quickly drops from 965 ms down to 432 ms are outside the visible part of the graph, since we focus on the minimum of *tcpu*.

The minimum cpu time is at twenty seven paths, but choosing any number between twenty five and forty eight paths is no more than one per cent worse than the optimum. Even choosing one hundred paths results in a performance degraded by only four per cent. This means the number of paths can be chosen from a relatively wide range without a significant loss of performance. This was confirmed in experiments on the other test sets. The outcome reflects the findings of Malouf et al. (this volume), suggesting the relative speed of type and feature structure unification is comparable between PET and the LKB.

Grammar	Platform	readings ϕ	filter %	etasks ϕ	pedges ϕ	tcpu ϕ (s)	space ϕ (kb)
October 1996	PAGE	2·55	51·3	1763	97	36·69	79093
August 1999	PET	7·00	95·2	756	292	0·15	1162

Table 3. *Progress made in processing the LinGO grammar over three years. Numbers obtained on a 300 megahertz UltraSparc using the 'aged' test set.*

3.4 Quantifying progress

In this section we take a wider perspective and give an impression of the overall progress made in processing the LinGO grammar over a period of three years. The oldest available profiles (for the '*aged*' test set) were obtained with PAGE (version 2.0 released in May 1997) using the October 1996 version of LinGO. The current best parsing performance, to our best knowledge, is achieved in the cheap parser of PET. All data was sampled on the same 300 megahertz UltraSparc server.

Table 3 shows that average parsing times[9] per test item have dropped by more than two orders of magnitude (a factor of two hundred and fifty on the '*aged*' data), while memory consumption was reduced by a factor of more than fifty. Because in the early PAGE data the quick check filter was not available, current filter rates are much better and result in a reduction of executed parser tasks. At the same time, comparing the number of passive edges licensed by the two versions of the grammar provides a good estimate on the search space explored by the two parsers. The '*aged*' data shows an increase by a factor of three. Assuming that the average number of passive edges is a direct measure for input complexity[10] (with respect to a particular grammar), we extrapolate the overall speed-up in processing the LinGO grammar as a factor of roughly seven hundred and fifty.

4 Conclusion

By synthesizing a range of techniques for efficient processing in an efficient implementation the cheap parser developed using the PET platform achieves very attractive practical performance. Both time and space requirements are significantly reduced compared to the PAGE and LKB systems, the process size is reduced by an order of magnitude. This clearly demonstrates that systematic experimentation, the precise and in-depth study of algorithms and encoding techniques used in various systems in conjunction with the synthesis of experience gained in several related development efforts are highly beneficial in building practical systems.

Using a collection of proven building blocks allows rapid implementation of new

[9] The *tcpu* values for PAGE include garbage collection time, which is eliminated in PET.

[10] This assumption is supported by very strong linear correlation between the number of passive edges and parsing time in both profiles ($r^2 = 0.92$ for the PAGE data; $r^2 = 0.99$ for the cheap data).

approaches and evaluation on realistic grammars. The integration to the [incr tsdb()] profiling package proved to be very useful for debugging (ensuring results were identical to a reference system) and identifying performance bottlenecks. A development mode of making small changes and analyzing the impact on performance after each change was very effective for optimizing the implementation.

Topics for further work include experimentation with different feature structure representations (exploiting strict appropriateness conditions), investigation of the interaction between parsing strategy and unifier and integration of subsumption-based local ambiguity packing (Oepen & Carroll, 2000).

4.1 Acknowledgments

I am grateful to Stephan Oepen, Ann Copestake and Dan Flickinger for many fruitful discussions, and answering numerous questions about [incr tsdb()], the LKB and the LinGO grammar, respectively. I would like to thank Liviu Ciortuz at DFKI Saarbrücken for the experience that I gained while working with him on the CHIC system. Finally, I am indebted to three anonymous reviewers for their valuable comments on this paper.

References

Aït-Kaci, H. (1991). *Warren's Abstract Machine: A tutorial reconstruction.* Cambridge, MA: MIT Press.

Aït-Kaci, H., Boyer, R., Lincoln, P., & Nasr, R. (1989). Efficient implementation of lattice operations. *ACM Transactions on Programming Languages and Systems, 11*(1), 115–146.

Carpenter, B., & Qu, Y. (1995). An abstract machine for attribute-value logics. In *Proceedings of the 4th International Workshop on Parsing Technologies.* Prague, Czech Republik.

Carroll, J. (1994). Relating complexity to practical performance in parsing with wide-coverage unification grammars. In *Proceedings of the 32nd Meeting of the Association for Computational Linguistics* (pp. 287–294). Las Cruces, NM.

Ciortuz, L. (2000). *Compiling HPSG into C* (DFKI Research Report). Saarbrücken, Germany: Deutsches Forschungszentrum für Künstliche Intelligenz GmbH.

Gerdemann, D. (1995). Term encoding of typed feature structures. In *Proceedings of the 4th International Workshop on Parsing Technologies* (pp. 89–97). Prague, Czech Republik.

Götz, T. (1993). *A normal form for typed feature structures.* Magisterarbeit, Universität Tübingen, Tübingen, Germany.

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). College Park, MD.

Krieger, H.-U., & Schäfer, U. (1994). $\mathcal{TDL}$ — A type description language for constraint-based grammars. In *Proceedings of the 15th International Conference on Computational Linguistics* (pp. 893–899). Kyoto, Japan.

Krieger, H.-U., & Schäfer, U. (1995). Efficient parameterizable type expansion for typed feature formalisms. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence* (pp. 1428 – 1434). San Francisco, CA.

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.* 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.

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

Wintner, S. (1997). *An abstract machine for unification grammars with applications to an HPSG grammar for Hebrew.* Unpublished doctoral dissertation, Technion, Israel Institute of Technology, Haifa, Israel.

Wroblewski, D. A. (1987). Non-destructive graph unification. In *Proceedings of the 6th National Conference on Artificial Intelligence* (pp. 582 – 587). Seattle, WA.