Validation and Regression Testing for a Cross-linguisic Grammar Resource

Emily M. Bender, Laurie Poulson, Scott F. Drellishak, Christie Evans
University of Washington
Department of Linguistics
Seattle WA 98195-4340 USA
`{ebender,lpoulson,sfd,chrisev@u.washington.edu}`

Abstract

In this paper we present a methodology for validation and regression testing of the Grammar Matrix core grammar and customization system. The space of possible language types covered by the Grammar Matrix is large and growing. It is not possible to develop a test suite for all of the language types. Instead, we have created a test suite generation system which reads in the same language-type description file as the grammar customization system and creates a gold standard test suite for comparison. The gold standard test suite includes well-formed strings paired with all of their valid semantic representations (for the language type) as well as strings which are ill-formed for the language type.

The string-semantics pairs for a test suite are selected from a set of candidates by a system of regular-expression based filters. The filters amount to an alternative grammar building system, whose generative capacity is limited compared to the actual grammars. We perform error analysis of the discrepancies between the test suites and grammars for a range of language types, and update both systems appropriately. The resulting resource serves as a point of comparison for regression testing in future development of the Matrix.

1 Introduction

The development and maintenance of test suites is integral to the process of writing deep linguistic grammars (Oepen and Flickinger, 1998; **?**). Such test suites typically contain hand-constructed examples illustrating the grammatical phenomena which have been considered to date as well as representative examples taken from texts from the target domain. They are used for both validation (does this analysis do what we intend it to do?) and regression testing of precision grammars. With test suite management software such as [incr tsdb()] (Oepen, 2001)), the grammar engineer can compare the linguistic 'competence' of the current grammar to that of previous stages of the grammar. In such comparisons, one can not only find out if the overall coverage, overgeneration and degree of ambiguity have increased (or decreased), but also explore which test suite items have different numbers of analyses in the different runs, and even which have different analyses (in terms of syntax or semantics). As recently as 10 years ago, running the test suite was an overnight affair, and so could only simultaneously test all of the days' changes. Recent advances in efficient parsing technology (Oepen et al., 2002b) and the ability to parallelize test runs have made it possible use test suite runs to explore the consequences of any given change the grammar engineer is considering (Oepen et al., 2002a). This has only increased the importance of the test suite in the grammar development process.

In this paper, we consider what happens when the precision grammar resource being developed isn't a

grammar of a particular language, but rather a cross-linguistic grammar resource. In particular, we consider the LinGO Grammar Matrix (Bender et al., 2002; Bender and Flickinger, 2005; Drellishak and Bender, 2005). There are several (related) obstacles to making effective use of test suites in this scenario:

1. The cross-linguistic resource isn't itself a grammar, and therefore can't parse any strings.

2. There is no single language modeled by the cross-linguistic resource from which to draw test strings.

3. The space of possible grammars (alternatively, language types) modeled by the resource is enormous, well beyond the scope of what can be thoroughly explored.

We present a methodology for the validation and regression testing of the Grammar Matrix which addresses these obstacles. In addressing obstacle 1, it relies on the fact that recent work on the Grammar Matrix (Bender and Flickinger, 2005; Drellishak and Bender, 2005) has been augmented with libraries implementing analyses of a range of variations for a small set of linguistic phenomena (e.g., basic word-order). In addition, there is a 'customization system' which elicits typological information about a language from a web-based form and outputs a customized 'starter-grammar' as well as a 'language description' file encoding the the choices the user made on the form.

In its broad outlines, the strategy looks like this:

1. Define an abstract vocabulary to be used for test suite purposes.

2. Define an initial small set of string-semantics pairs.

3. Construct a large set of variations on the string-semantics pairs.

4. Define a set of filters which can select the legitimate string-semantics pairs for a particular language type from the overall set

The filters in effect constitute a parallel grammar definition system, albeit one which creates 'grammars' of very limited generative capacity. As such,

the output of the filters cannot be taken as ground truth. Rather, they serve as a point of comparison that allows us to find discrepancies between the filters and the Grammar Matrix (including the libraries and customization system). The discrepancies might be due to an error on either side. Nonetheless, we believe that maintaining the filter systems is a worthwhile investment, as we see no other systematic way to turn up errors in the Grammar Matrix short of using it to build grammars for all known languages.

The remainder of this paper is structured as follows: §2 briefly describes the relevant properties of the Grammar Matrix. §3 discusses related work on the testing of complex software systems and explores how the issues that arise in multilingual grammar engineering differ from those usually considered in software engineering. §4 presents the methodology for creating test suites in more detail. §5 illustrates the kind of errors which can be isolated with this technique. §**??** concludes the paper with a discussion of how the test suite resource should be maintained and extended over time and potential uses of the test suite resource outside the Grammar Matrix project.

2 Background

2.1 Grammar Matrix Overview

The Grammar Matrix is an open-source starter kit designed to jump-start the development of broad-coverage precision grammars, capable of both parsing and generation and suitable for use in a variety of NLP applications. The Grammar Matrix is written within the HPSG framework (Pollard and Sag, 1994; Sag et al., 2003), using Minimal Recursion Semantics (MRS; (Copestake et al., 2005)) for the semantic representations. The particular formalism we use is TDL (type description language) as interpreted by the LKB (Copestake, 2002) grammar development environment. Initial work on the Matrix (Bender et al., 2002; **?**) focused on the development of a cross-linguistic core grammar. The core grammar provides a solid foundation for sustained development of linguistically-motivated yet computationally tractable grammars (e.g., (Hellan and Haugereid, 2003; Kordoni and Neu, 2005)).

However, the core grammar alone cannot parse and generate sentences: it needs to be specialized

with language-specific information such as the order of daughters in its rules (e.g., head-subject or subject-head), and it needs a lexicon. Although word order and many other phenomena vary across languages, there are still recurring patterns. To allow reuse of grammar code across languages and to increase the size of the jump-start provided by the Matrix, in more recent work Bender and Flickinger (2005) and Drellishak and Bender (2005) have been developing 'libraries' implementing realizations of various linguistic phenomena. Through a web interface, grammar developers can configure an initial starter grammar by filling out a typological questionnaire about their language, which in turn calls a CGI script to 'compile' a grammar (including language-specific rule types, lexical entry types, rule entries, and lexical entries) by making appropriate selections from the libraries. These little grammars describe very small fragments of the languages they model, but they are not toys. Their purpose is to be good starting points for further development.

The initial set of libraries includes: basic word order of major constituents in matrix clauses (SOV et al), optionality/obligatoriness of determiners, noun-determiner order, NP v. PP arguments of intransitive and transitive verbs, strategies for expressing sentential negation and yes-no questions, and strategies for constituent coordination. Even with this small set of phenomena covered (and limiting ourselves arbitrarily for testing purposes to a maximum of two co-ordination strategies per language), we have already defined a space of hundreds of thousands of possible grammars.[1]

2.2 The Non-modularity of Linguistic Phenomena

Usability considerations put two important constraints on the customization system:

1. The questions must be ones that are sensible to linguists, who tend to consider phenomena one at a time.

2. The output grammar code must be both readable and maintainable.

To achieve readable grammar code in the output TDL, among other things, we follow the guideline that any given constraint is stated only once. If multiple types require the same constraint, they should all inherit from some supertype which bears the constraint. In addition, all constraints pertaining to a particular type are stated in one place.

The Grammar Matrix customization system reads in the user's language specification and then outputs language-specific definitions of types (rule types, lexical entry types and ancillary structures) which inherit from types defined in the crosslinguistic core of the Matrix but add constraints appropriate for the language at hand. The customization system is implemented as a Python script which builds TDL descriptions, prints them to the appropriate files, includes the cross-linguistic shared files, and presents the user with an archive for downloading.

In light of the two basic constraints on the customization system, we have found that it is not possible to treat the libraries as black-box modules with respect to each other. The libraries are interdependent, and the portions of the script which interpret one part of the input questionnaire frequently need to make reference to information elicited by other parts of the questionnaire. For example, the customization system implements major constituent word order by specializing the head-complement and head-subject rule types provided in the core grammar. In an SOV language, these would both be cross-classified with the type head-final, and the head-subject rule would further be constrained to take only complement-saturated phrases as its head daughter. The TDL encoding of these constraints is shown in Figure 1.

Following standard practice in HPSG, we use the head-complement phrase not only for combining verbs with their complements to make VPs, but also for all other head complement structures, notably PPs, CPs, and VPs headed by auxiliaries. These three are notable because they are all implemented in the Grammar Matrix customization system and because the order of head and complement can differ among them. Consider Polish, a free word order language that nonetheless has prepositions. The

---

[1] If all of the choices in the customization system were independent, we would have more than 2 x 10^{27} grammars. In actuality, constraints on possible combinations of choices (e.g., if a grammar has two case-marking adpositions, they must both be prepositions or both be postpositions) limit this space considerably.

```
comp-head-phrase := basic-head-1st-comp-phrase & head-final.
subj-head-phrase := basic-head-subj-phrase & head-final &
  [ HEAD-DTR.SYNSEM.LOCAL.CAT.VAL.COMPS < > ].
```

Figure 1: Specialized phrase structure rule types for SOV language

order of a verb with respect to its complements is free, so we instantiate both head-comp and comp-head rules, which inherit from head-initial and head-final respectively. Yet the prepositions must be barred from the head-final version lest the grammar license *post*positional phrases by mistake. We do this by constraining the HEAD value of the comp-head phrase. Similarly, question particles (such as *est-ce que* in French or *ma* in Mandarin) are treated as complementizers: heads which select for an S complement. Since these, too, may differ in their word order properties from verbs (and prepositions), we need information about the question particles (elicited with the rest of the information about yes-no questions) before we have complete information about the head-complement rule. Furthermore, it is not simply a question of adding constraints to existing types: Consider the case of an SOV language with prepositions and sentence-initial question particles. This language would need a head-initial head-comp rule that can take only prepositions and complementizers as its head. To express the disjunction, we must use the supertype to *prep* and *comp*. This, in turn, means that we can't decide what constraint to put on the head value of the head-comp rule until we've considered questions as well as the basic word order facts.

We expect to study the issue of (non-)modularity as we add additional libraries to the resource and to investigate whether the grammar code can be refactored in such a way as to make the libraries into true modules. We suspect at this point that while it might be possible to reduce the degree of interdependence, it will not be possible to achieve completely independent libraries, because syntactic phenomena are inherently interdependent. Consider the case of agreement in NP coordination. In English and many other languages, coordinated NPs are always plural, regardless of the number value of the coordinands. Furthermore, the person of the coordinated NP is the minimal person value of the coordinands.

(1) a. A cat and a dog are/*is chasing a mouse.

b. Kim and I should handle this ourselves.

c. You and Kim should handle this yourselves.

In languages with gender systems, there is often a similar hierarchy of gender values, e.g., in French coordianted NPs the whole NP is feminine iff all coordinands are feminine and masculine otherwise. Thus it appears that it is not possible to define all of the necessary constraints on the coordination rules without having access to information about the agreement system.

Even if the libraries could be made completely independent at the customization level, however, the various parts of the grammar need to be able to interact properly in the analysis of individual sentences. Any sentence which illustrates sentential negation, a matrix yes-no question, or coordination also necessarily illustrates at least some aspects of word order, the presence v. absence of determiners and case-marking adpositions, and the subcategorization of the verb that heads the sentence. Furthermore, broad-coverage grammars need to allow negation, questions, coordination etc. all to appear in the same sentence.

Given this non-modularity, we would ideally like to be able to validate (and do regression testing on) the full set of grammars generable by the customization system, representing every combination of every answer to every question. To approximate such a thorough testing procedure, we can instead sample from the grammar space.

3 Related Work: Validation of complex software systems

4 Methodology

This section describes in detail our methodology for creating test suites on the basis of language-type descriptions. We begin by defining some terms: We use *string* to refer to a sequence of words to be input to a grammar and *result* as the (expected) semantic representation. An *item* is a particular pair of string

and result. Among strings, we have *seed strings* provided by the Matrix developers to seed the test suite, and *constructed strings* derived from those seed strings. The *constructor function* is the algorithm for deriving new strings from the seed strings. Seed strings are arranged into semantic equivalence classes, from which one representative is designated the *harvester string*. We parse the harvester string with some appropriate grammar (derived from the Matrix customization system) to extract the semantic representation to be paired with each member of the equivalence class.

A *language type* is a collection of feature-value pairs representing a possible set of answers to the Matrix customization questionnaire. We refer to these as language types rather than languages, because the grammars produced by the customization system are underspecified with respect to actual languages. For example, they currently lack any analysis of case (outside the option of case-marking adpositions) or agreement. In fact, they will always be underspecified, no matter how large the system gets, because it is not possible to describe all of the details of all of the world's languages in a system like this—just getting the 'core' grammar down will be challenge enough. Thus one and the same starter grammar might be extended into multiple models corresponding to multiple actual human languages. Accordingly, when we talk about the predicted (un)grammaticality of a candidate string, we are referring to its predicted well- or ill-formedness given the information contained in the language type definition, not its grammaticality in any particular (human) language.

4.1 Implementation: Python and MySQL

The test suite generation system includes a MySQL database, a collection of Python scripts which interact with the database, and some stored SQL queries. As the list of items we are manipulating is quite large (and will grow as new items are added to test additional libraries), using a database is essential for rapid retrieval of relevant items. Furthermore, the database facilitates the separation of procedural and declarative knowledge in the definition of the filters.

4.2 Abstract vocabulary for abstract strings

In order to parse and generate, a grammar needs not just syntactic constructions and lexical types, but also an actual lexicon. Since we are working at the level of language types (and not actual languages), we are free to define this lexicon in whatever way is most convenient. Much of the idiosyncrasy in language resides in the lexicon, both in the form of morphemes and in the particular grammatical and collocational constraints associated with them. Of these three, only the grammatical constraints are manipulated in any interesting way within the Grammar Matrix customization system. As such, we define all of the language types to draw the *forms* of their lexical items from a shared, standardized vocabulary, illustrated in Table 1. Table 1 also lists the options which are currently available for varying the grammatical constraints on the lexical entries. Using the same word forms for each grammar contributes substantially to building a single resource which can be adapted for the testing of each language type.

4.3 Constructing master item set

For each library to be tested, we require that the grammar engineer provide a list of seed strings, organized into semantic equivalence classes with designated harvester strings, and a specification of one or more grammars which can parse the harvester strings.

The seed strings, when looked at as bags of words, should cover all possible realizations of the phenomenon treated by the library. For example, the negation library allows both inflectional and adverbial negation, as well as negation expressed through both inflection and an adverb together. To illustrate negation of transitive sentences (allowing for languages with and without determiners[2]), we require the seed strings in (2):

(2)

Semtag: neg1	Semtag: neg2
n1 n2 neg tv	det n1 det n2 neg tv
n1 n2 neg-tv	det n1 det n2 neg-tv
n1 n2 tv-neg	det n1 det n2 tv-neg
n1 n2 neg neg-tv	det n1 det n2 neg neg-tv
n1 n2 neg tv-neg	det n1 det n2 neg tv-neg

---

[2]We require additional seed strings to account for languages with and without case-marking adpositions

Form	Description	Options
det	determiner	
n1, n2	nouns	det is optional, obligatory, impossible
iv, tv	intransitive, transitive verb	subj, obj are NP or PP
p-nom, p-acc	case-marking adpositions	preposition or postposition
neg	negative element	adverb, prefix, suffix
co1, co2	coordination marks	word, prefix, suffix
qpart	question particle	

Table 1: Standardized lexicon

Sentential negation has the same semantic reflex across all of its realizations, but the presence v. absence of overt determiners does have a semantic effect. Accordingly, the seed strings shown in (2) can be grouped into two semantic equivalence classes, shown as the first and second columns in the table, and associated with the semantic tags 'neg1' and 'neg2', respectively. The two strings in the first row could be designated as the harvester strings, associated with a grammar for an SOV language with optional determiners preceding the noun and sentential negation expressed as pre-head modifier of V.

We then use the LKB to parse the harvester strings with the associated grammar through [incr tsdb()] and add the resulting linked string and result rows into our database. We further add the relations between the harvester strings, semantic tags, and parsing results into the database, and then use that information to create new rows in the item and result tables for the seed strings from each semantic equivalence class. We then use the constructor function to create new strings on the basis of the seed strings in the DB. This process is schematized in Figure 2.

Graphic showing how harvester & seed strings fan out into full test set.

Figure 2: Derivation of master item set from seed strings

Currently, we have only one constructor function ('permute') which takes in a seed string and returns all unique permutations of the morphemes in that seed string.[3] This constructor function is effective in producing test items which cover the range of word

order variation permitted by the Grammar Matrix customization system. As the range of phenomena handled by the customization system expands, we may need more sophisticated constructor functions, which could handle, for example, the addition of all possible case suffixes to each noun in the sentence.

4.4 Filters

The master item list provides us with an inventory from which we can find positive (grammatical) examples for any language type generated by the system as well as interesting negative examples for any language type. To do so, we filter the master item list, in two steps.

4.4.1 Universal Filters

The first step is the 'universal' filters, which mark any item known to be ungrammatical across all language types currently produced by the system. For example, the word order library does not currently provide an analysis of radically non-configurational languages with discontinuous NPs (e.g., Warlpiri (Hale, 1981)). Accordingly, the string in (3) will be ungrammatical across all language types:

(3) det det n1 n2 tv

The universal filter definitions (provided by the grammar engineers) each comprise one or more regular expressions, a filter type which specifies how the regular expressions are to be applied, and a list of semantic tags specifying which equivalence classes they apply to. For example, the filter which would catch example (3) above is defined as in (4):

---

[3] 'permute' strips off any affixes, permutes the stems, and then attaches the affixes to the stems in all possible ways.

(4) Type: reject-if-match
     Regexp: (det (n1|n2).*det (n1|n2))|
                 (det (n1|n2).*(n1|n2) det)|
                 ((n1|n2) det.*det (n1|n2))|
                 ((n1|n2) det.*(n1|n2) det)
     Sem-class: [semantic classes for all transitive
                 sentences with two determiners.]

We create a 'filters' table in the database with a row for each filter. We then select each item from the database and process it with all of the filters. For each filter whose semantic-class value includes the semantic class of the item at hand, we store the result (pass or fail) of the filter for the item in the item-filters table. When all of the items have been processed with all of the universal filters, we can query the database to produce a list of all of the potentially well-formed items.

4.4.2 Specific Filters

The next step is to run the filters which find the grammatical examples for a particular language type. In order to facilitate sampling of the entire language space, we define these filters to be sensitive not to complete language type definitions, but rather to particular features (or small sets of features) of a language type. Thus in addition to the filter type, regular expression, and semantic class fields, the language-specific filters also encode partial descriptions in the form of feature-value declarations of the language types they apply to. For example, the filter in (5) plays a role in selecting the correct form of negated sentences in a language has both inflectional negation and adverbial negation, but the two are in complementary distribution (like English *n't* and sentential *not*). The first regular expression checks for *neg* surrounded by white space (i.e., the negative adverb) and the second checks for the negative affixes.

(5) Type: reject-if-both-match
     Regexp1:   (\s|^)neg(\s|$)
     Regexp2:   -neg|neg-
     Sem-class: [sem. classes for all neg. sent.]
     Lg-feat: and(infl_neg:on,adv_neg:on,
                 multineg:comp)

This filter uses a conjunctive language feature specification (three feature-value pairs which must apply), but disjunctions are also possible (as well as conjunctions of disjunctions, etc.; these specifications are converted to disjunctive normal form before further processing).

As with the universal filters, the specific filters are stored in the database. We process each item which passed all of the universal filters with each specific filter. Whenever a filter's semantic-class value matches the semantic-class of the item at hand, we store the value assigned by the filter (pass or fail) in the item-filter table. We also store the feature-value pairs required by each filter, so that we can look up the relevant filters for a language-type definition.

The filtering process is illustrated in Figure 3.

Graphic showing how master item set is reduced by universal features and then indexed by specific features.

Figure 3: Derivation of master item set from seed strings

4.4.3 Recursive Linguistic Phenomena

Making the filters relative to particular semantic classes allows us to use information about the lexical items in the sentences in the definition of the filters. This makes it easier to write regular-expression based filters which can work across many different complete language types. Nonetheless, we have to be prepared to handle some recursive phenomena. A case in point is coordination, where an NP position, for example, is filled by a coordinated NP.

To handle such phenomena with our finite-state system, we do multiple with the filters. All items with coordination are first processed with the coordination filters, and then rewritten to replace any well-formed coordinations with single constituents. These rewritten strings are then processed with the rest of the filters, and we store the results as the results for those filters on the *original* strings.[4]

4.5 Language types

The final kind of information we store in the database is definitions of language types. Even though our system allows us to create test suites

---

[4] _FIX_ME_: We actually haven't built this part yet, and it's like to be a bit more complicated than this...

for new language types on demand, we still store the language-type definitions of language types we have tested, for future regression testing purposes. When a language type is read it, the list of feature-value pairs defining it is compared to the list of feature-groups declared by the filters. For each group present in the language-type definition, we find all of the filters which use that group. We then query the database for all items which pass the filters relevant to the language type. This list of items represents all those in the master item list predicted to be well-formed for this language type. From the complement of this set, we also take a random selection of items to test for overgeneration. We then export the test suite to [incr tsdb()] format.

4.6 Summary

The input required from the grammar engineer in order to test any new library is as follows:

1. Seed strings illustrating the range of expressions handled by the new library, organized into equivalence classes.

2. Designated harvester strings for each equivalence class and a grammar or grammars which can parse them and get the target semantic representation.

3. Universal filters specific to the phenomenon and seed strings.

4. Specific filters picking out the right items for each language type

This is a substantial investment on the part of the grammar engineer, but we believe it is worth it for the return of being able to validate the library which has been added.

[_FIX_ME_: Laurie, I think this might be a good place to mention the Arnold et al paper (1993?) which discusses an alternative way of constructing test suites. But, I don't have that paper, do you have it still?]

5 Error analysis

Emphasize that we can sample the language space at will, and what that buys us

6 Conclusion and Future Work

The methodology outlined in this paper addresses the three obstacles noted in the introduction: (1) Although the cross-linguistic resource itself isn't a grammar, we test it by deriving grammars from it. The Grammar Matrix customization system allows us to specify starter grammars for language types which can parse and generate. Since we are testing the derived grammars, we are simultaneously testing both the Matrix core grammar, the libraries, and the customization script. (2) Although there is no single language being modeled from which to draw strings, we can nonetheless find a relevant set of strings and associated them with annotations of expected well-formedness. The lexical formatives of the strings are drawn from a standardized set of lexical forms. The well-formedness predictions are made on the basis of the system of filters. The system of filters doesn't represent ground truth, but rather a second pathway to the judgments in addition to the direct use of the Matrix-derived starter grammars. These pathways are independent enough that the one can serve as an error check on the other. (3) The space of possible language types remains too large for thorough testing. However, since our system allows for the efficient derivation of a test suite for any arbitrary language type, it is inexpensive to sample that language-type space in many different ways.

Future work – ?

References

Emily M. Bender and Dan Flickinger. 2005. Rapid prototyping of scalable grammars: Towards modularity in extensions to a language-independent core. In *Proceedings of the 2nd International Joint Conference on Natural Language Processing IJCNLP-05 (Posters/Demos)*, Jeju Island, Korea.

Emily M. Bender, Dan Flickinger, and Stephan Oepen. 2002. The grammar matrix: An open-source starter-kit for the rapid development of cross-linguistically consistent broad-coverage precision grammars. In *Proceedings of the Workshop on Grammar Engineering and Evaluation at the 19th International Conference on Computational Linguistics*, pages 8–14.

Ann Copestake, Dan Flickinger, Carl Pollard, and Ivan A. Sag. 2005. Minimal recursion semantics: An introduction. *Research on Language & Computation*, 3(2–3):281–332.

Ann Copestake. 2002. *Implementing Typed Feature Structure Grammars*. CSLI, Stanford.

Scott Drellishak and Emily M. Bender. 2005. A coordination module for a crosslinguistic grammar resource. In Stefan Müller, editor, *The Proceedings of HPSG 2005*, pages 108–128, Stanford. CSLI.

Kenneth Hale. 1981. On the position of Warlpiri in the typology of the base. Distributed by Indiana University Linguistics Club, Bloomington.

Lars Hellan and Petter Haugereid. 2003. Norsource: An exercise in matrix grammar-building design. In *Proceedings of the Workshop on Ideas and Strategies for Multilingual Grammar Development, ESSLLI 2003*, pages 41–48.

Valia Kordoni and Julia Neu. 2005. Deep analysis of Modern Greek. In Keh-Yih Su, Jun'ichi Tsujii, and Jong-Hyeok Lee, editors, *Lecture Notes in Computer Science*, volume 3248, pages 674–683. Springer-Verlag, Berlin.

Stephan Oepen and Daniel P. Flickinger. 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.

Stephan Oepen, Emily M. Bender, Ulrich Callmeier, Dan Flickinger, and Melanie Siegel. 2002a. Parallel distributed grammar engineering for practical applications. In *Proceedings of the Workshop on Grammar Engineering and Evaluation, COLING 2002*.

Stephan Oepen, Daniel Flickinger, J. Tsujii, and Hans Uszkoreit, editors. 2002b. *Collaborative Language Engineering. A Case Study in Efficient Grammar-based Processing*. CSLI, Stanford.

Stephan Oepen. 2001. [incr tsdb()] — Competence and performance laboratory. User manual. Technical report, Saarbrücken, Germany.

Carl Pollard and Ivan A. Sag. 1994. *Head-Driven Phrase Structure Grammar*. Studies in Contemporary Linguistics. The University of Chicago Press.

Ivan A. Sag, Thomas Wasow, and Emily M. Bender. 2003. *Syntactic Theory: A Formal Introduction*. CSLI, Stanford, second edition.