Validation and Regression Testing for a Cross-linguisic Grammar Resource

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

Abstract

We present a validation methodology for a cross-linguistic grammar resource which produces output in the form of small grammars based on elicited typological descriptions. Evaluating the resource entails sampling from a very large space of language types, the type and range of which preclude the use of standard test suites development techniques. We produce a database from which gold standard test suites for these grammars can be generated on demand, including well-formed strings paired with all of their valid semantic representations as well as a sample of ill-formed strings. These string-semantics pairs 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.

1 Introduction

The development and maintenance of test suites is integral to the process of writing deep linguistic grammars (Oepen and Flickinger, 1998; Butt and King, 2003). Such test suites typically contain hand-constructed examples illustrating the grammatical phenomena treated by the grammar as well as representative examples taken from texts from the target domain. In combination with test suite management software such as [incr tsdb()] (Oepen, 2002), they are used for validation and regression testing of precision (deep linguistic) grammars as well as the exploration of potential changes to the grammar.

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). There are several (related) obstacles to making effective use of test suites in this scenario: (1) The Matrix core grammar 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 that addresses these obstacles, developing the ideas originally proposed in (Poulson, 2006). In its broad outlines, our methodology looks like this:

- Define an abstract vocabulary to be used for test suite purposes.
- Define an initial small set of string-semantics pairs.
- Construct a large set of variations on the string-semantics pairs.

- Define a set of filters that can delineate the legitimate string-semantics pairs for a particular language type

The filters in effect constitute a parallel grammar definition system, albeit one that creates ‘grammars’ of very limited generative capacity. As such, the output of the filters cannot be taken as ground truth. Rather, it serves as a point of comparison that allows us to find discrepancies between the filters and the Grammar Matrix which in turn can lead us to errors in the Grammar Matrix.

2 Background

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), using Minimal Recursion Semantics (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; Flickinger and Bender, 2003) 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; Drellishak and Bender, 2005), we 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 for testing purposes to maximally two coordination strategies per language), we have already defined a space of hundreds of thousands of possible grammars.[1]

3 The Non-modularity of Linguistic Phenomena

In this section we discuss our findings so far about the non-modularity of linguistic phenomena, and argue that this makes the testing of a broad sample of grammars even more pressing.

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) that inherit from types defined in the crosslinguistic core of the Matrix but add constraints appropriate for the language at hand. Usability considerations put two important constraints on this 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 bearing that constraint. In addition, all constraints pertaining to a particular type are stated in one place.

In light of the these usability considerations, we

[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 limit this space considerably.

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 that 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 ordinary VPs, but also for PPs, CPs, and auxiliary-headed VPs, etc. Consider Polish, a free word order language that nonetheless has prepositions. To allow complements on either side of the head, we instantiate both head-comp and comp-head rules, inheriting 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 that 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 it might be possible to reduce the degree of interdependence, but not to achieve completely independent libraries, because syntactic phenomena are inherently interdependent. Agreement in NP coordination provides an example. In English and many other languages, coordinated NPs are always plural and 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.

Gender systems often display a similar hierarchy of values, as with French coordinated NPs, where 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 we were able to make our analyses of different linguistic phenomena completely modular, however, we would still need to test their interaction in the analysis of particular sentences. Any sentence that 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. To approximate such thoroughness, we instead sample from the grammar space.

4 Methodology

This section describes in some detail our methodology for creating test suites on the basis of language-type descriptions. 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

```
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

the customization system are underspecified with respect to actual languages, i.e., 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 (well)formedness, or (un)grammaticality, of a candidate string, we are referring to its predicted status with respect to a 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 that interact with the database, and some stored SQL queries. As the set 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

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, 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. Therefore, for the test suite, we define all of the language types to draw the *forms* of their lexical items from a shared, standardized vocabulary. Table 1 illustrates the vocabulary along with the options that 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 that can be adapted for the testing of each language type.

4.3 Constructing master item set

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 (*result*) to be paired with each member of the equivalence class.

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

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

[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

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 a pre-head modifier of V.

We use the LKB in conjunction with [incr tsdb()] to parse the harvester strings from all of the equivalence classes with the appropriate grammars. Then the seed strings and the parsing results from the harvester strings, as well as their semantic tags, are stored and linked in our relational database. We use the constructor function to create new strings from these seed strings. This produces the master item set that provides the basis for the test suites.

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 that cover the range of word order variations currently permitted by the Grammar Matrix customization system. Currently, most of the other kinds of variation countenanced (e.g., adverbial v. inflectional negation or presence v. absence of determiners) is handled through the initial seed string construction. As the range of phenomena handled by the customization system expands, we will develop more sophisticated constructor functions to handle, for example, the addition of all possible case suffixes to each noun in the sentence.

4.4 Filters

The master item set 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 set, in two steps.

4.4.1 Universal Filters

The first step is the application of '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 nonconfigurational languages with discontinuous NPs (e.g., Walpiri (Hale, 1981)). Accordingly, (3) will be ungrammatical across all language types:

(3) det det n1 n2 tv

The universal filter definitions (provided by the developers) each comprise one or more regular expressions, a filter type that 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 that would catch example (3) above is defined as in (4):

(4) Filter Type: reject-unless-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 apply each filter to every item in the database. 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 on that item. We can then 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 that 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

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

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 of the language types to which they apply, in the form of feature-value declarations. For example, the filter in (5) plays a role in selecting the correct form of negated sentences for language types with both inflectional and adverbial negation 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 for the negative affixes.

(5) Filter 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 that must apply), but disjunctions are also possible. These specifications are converted to disjunctive normal form before further processing.

As with the universal filters, the results of the specific filters are stored in the database. We process each item that 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). 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.

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 that can work across many different complete language types. Complications arise, however, in examples illustrating recursive phenomena To handle such phenomena with our finite-state system, we do multiple passes 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.

4.5 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 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 in, the list of feature-value pairs defining it is compared to the list of feature-groups declared by the filters. For each group of feature-value pairs present in the language-type definition, we find all of the filters that use that group. We then query the database for all items that pass the filters relevant to the language type. This list of items represents all those in the master item set 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.

4.6 Validation of grammars

Once we have created the test suite for a particular language type, the developer runs the Matrix customization system to get a starter grammar for the same language type. The test suite is loaded into [incr tsdb()] and processed with the grammar. [incr tsdb()] allows the developer to compare the grammar's output with the test suite at varying levels of detail: Do all and only the items predicted to be well-formed parse? Do they get the same number of readings as predicted? Do they get the semantic representations predicted? A discrepancy at any of these levels points to an error in either the Grammar Matrix or the test suite generation system. The developer can query the database to find which filters passed or failed a particular example as well as to discover the provenance of the example and which phenomena it is meant to test.

This methodology provides the ability to generate test suites for any arbitrary language type on demand. Although this appears to eliminate the need to store the test suites we do, in fact, store information about previous test suites. This allows us to track the evolution of the Grammar Matrix in relation to those particular language types over time.

4.7 Investment and Return

The input required from the developer 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 that can parse them to 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. (5) Analysis of discrepancies between the test suite and the generated grammars. This is a substantial investment on the part of the developer but we believe the investment is worth it for the return of being able to validate a library addition and test for any loss of coverage going forward.

Arnold et al. (1994) note that writing grammars to generate test suites is impractical if the test suite generating grammars aren't substantially simpler to write than the 'actual' grammars being tested. Even though this system requires some effort to maintain, we believe the methodology remains practical for two reasons. First, the input required from the developer enumerated above is closely related to the knowledge discovered in the course of building the libraries in the first place. Second, the fact that the filters are sensitive to only particular features of language types means that a relatively small number of filters can create test suites for a very large number of language types.

5 Related Work

Kinyon and Rambow (2003) present an approach to generating test suites on the basis of descriptions of languages. The language descriptions are Meta-Grammar (MG) hierarchies. Their approach appears to be more flexible than the one presented here in some ways, and more constrained in others. It does not need any input strings, but rather produces test items from the language description. In addition, it annotates the output in multiple ways, including phrase structure, dependency structure, and LFG F-structure. On the other hand, there is no apparent provision for creating negative (ungrammatical) test data and it is does not appear possible to compose new MG descriptions on the fly. Furthermore, the focus of the MG test suite work appears to be the generation of test suites for other grammar development projects, but the MGs themselves are crosslinguistic resources in need of validation and testing. An interesting area for future work would be the comparison between the test suites generated by the system described here and the MG test suites.

The key to the test-suite development process proposed here is to leverage the work already being done by the Matrix developers into a largely automated process for creating test-suite items. The information required from the developers is essentially a structured and systematic version of the knowledge that is required for the creation of libraries in the first place. This basic approach, is also the basis for the approach taken in (Bröker, 2000); the specific forms of knowledge leveraged, and the test-suite development strategies used, however, are quite different.

6 Future Work

The addition of the next library to the Grammar Matrix will provide us with an opportunity to try to quantify the effect of this methodology. With the Grammar Matrix and the filters stabilized, the validation of a new library can be carefully tracked. We can try to quantify the number of errors obtained and the source of the errors, e.g., library or filters.

In addition to this kind of quantification and error analysis as a means of validating this methodology, we also intend to undertake a comparison of the test suites created from our database to hand built created for Matrix-derived grammars by students in the multilingual grammar engineering course at the University of Washington.[4] Students in this class each develop grammars for a different language, and create test suites of positive and negative examples as part of their development process. We plan to use the lexical types in the grammars to define a mapping from the surface lexical items used in the test suites to our abstract vocabulary. We can then compare the hand built and autogenerated test suites in order to gauge the thoroughness of the system presented here.

7 Conclusion

The methodology outlined in this paper addresses the three obstacles noted in the introduction: Al-

[4] http://courses.washington.edu/ling567

though the Grammar Matrix core itself isn't a grammar (1), we test it by deriving grammars from it. Since we are testing the derived grammars, we are simultaneously testing both the Matrix core grammar, the libraries, and the customization script. Although there is no single language being modeled from which to draw strings (2), we can nonetheless find a relevant set of strings and associate these strings with annotations of expected well-formedness. The lexical formatives of the strings are drawn from a standardized set of abstract 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. The space of possible language types remains too large for thorough testing (3). 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.

Acknowledgments

This work has been supported by NSF grant BCS-0644097.

References

Doug Arnold, Martin Rondell, and Frederik Fouvry. 1994. Design and implementation of test suite tools. Technical Report LRE 62-089 D-WP5, University of Essex, UK.

Emily M. Bender and Dan Flickinger. 2005. Rapid prototyping of scalable grammars: Towards modularity in extensions to a language-independent core. In *Proc. IJCNLP-05 (Posters/Demos)*.

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 *Proc. the Workshop on Grammar Engineering and Evaluation COLING 2002*, pages 8–14.

Norbert Bröker. 2000. The use of instrumentation in grammar engineering. In *Proc. COLING 2000*, pages 118–124.

Miriam Butt and Tracy Holloway King. 2003. Grammar writing, testing, and evaluation. In *Handbook for Language Engineers*, pages 129–179. CSLI.

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.

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

Dan Flickinger and Emily M. Bender. 2003. Compositional semantics in a multilingual grammar resource. In *Proc. the Workshop on Ideas and Strategies for Multilingual Grammar Development, ESSLLI 2003*, pages 33–42.

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 *Proc. the Workshop on Ideas and Strategies for Multilingual Grammar Development, ESSLLI 2003*, pages 41–48.

Alexandra Kinyon and Owen Rambow. 2003. The metagrammar: A cross-framework and cross-language test-suite generation tool. In *Proc. 4th International Workshop on Linguistically Interpreted Corpora*.

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.

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. 2002. *Competence and Performance Profiling for Constraint-based Grammars: A New Methodology, Toolkit, and Applications*. Ph.D. thesis, Universität des Saarlandes.

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

Laurie Poulson. 2006. Evaluating a cross-linguistic grammar model: Methodology and gold-standard resource development. Master's thesis, University of Washington.