Comparing the performance of two TAG-based surface realisers using controlled grammar traversal

Claire Gardent, Benjamin Gottesman, Laura Perez-Beltrachini

Comparing the performance of two TAG-based surface realisers using controlled grammar traversal

Claire Gardent
CNRS/LORIA
claire.gardent@loria.fr

Benjamin Gottesman
acrolinx GmbH
ben.gottesman@acrolinx.com

Laura Perez-Beltrachini
Université Henri Poincaré/LORIA
laura.perez@loria.fr

Abstract

We present GENSEM, a tool for generating input semantic representations for two sentence generators based on the same reversible Tree Adjoining Grammar. We then show how GENSEM can be used to produced large and controlled benchmarks and test the relative performance of these generators.

1 Introduction

Although computational grammars are mostly used for parsing, they can also be used to generate sentences. This has been done, for instance, to *detect overgeneration by the grammar* (**?**). Sentences that are generated but are ungrammatical indicate flaws in the grammar. This has also been done to *test a parser* (**?**; **?**). Using the sentences generated from the grammar ensures that the sentences given to the parser are in the language it defines. Hence a parse failure necessarily indicates a flaw in the parser's design as opposed to a lack of coverage by the grammar.

Here we investigate a third option, namely, the *focused benchmarking of sentence realisers* based on reversible grammars, i.e. on grammars that can be used both to produce sentences from a semantic representation and semantic representations from a sentence.

More specifically, we present a linguistically-controlled grammar traversal algorithm for Tree Adjoining Grammar (TAG) which, when applied to a reversible TAG, permits producing arbitrarily many of the semantic representations associated by this TAG with the sentences it generates. We then show that the semantic representations thus produced can be used to compare the relative performance of two sentence generators based on this grammar.

Although the present paper concentrates on Tree Adjoining Grammar realisers, it is worth pointing out that the semantic representations produced could potentially be used to evaluate any surface realiser whose input is a flat semantic formula.

Section 2 discusses related work and motivates the approach. Section 3 presents GENSEM, the DCG-based grammar traversal algorithm we developed. We show, in particular, that the use of a DCG permits controlling grammar traversal in such a way as to systematically generate sets of semantic representations covering certain computationally or linguistically interesting cases. Finally, Section 4 reports on the benchmarking of two surface realisers with respect to a GENSEM-produced benchmark.

2 Motivations

Previous work on benchmark construction for testing the performance of surface realisers falls into two camps depending on whether or not the realiser uses a reversible grammar, that is, a grammar that can be used for both parsing and generation.

To test a surface realiser based on a large reversible Head-Driven Phrase Structure Grammar (HPSG), Carroll et al. (**?**) use a small test set of two hand-constructed and 40 parsing-derived cases to test the impact of intersective modifiers[1] on generation performance. More recently, Carroll and Oepen (**?**) present a performance evaluation which uses as a benchmark the set

[1] As first noted by Brew (**?**) and Kay (**?**), given a set of n modifiers all modifying the same structure, all possible intermediate structures will be constructed, i.e., 2^{n+1}.

of semantic representations produced by parsing 130 sentences from the Penn Treebank and manually selecting the correct semantic representations. Finally, White (**?**) profiles a CCG[2]-based sentence realiser using two domain-focused reversible CCGs to produce two test suites of 549 and 276 ⟨ semantic formula, target sentence ⟩ pairs, respectively.

For realisers that are not based on a reversible grammar, there are approaches which derive large sets of realiser input from the Penn Treebank (PTB). For example, Langkilde-Geary (**?**) proposes to translate the PTB annotations into a format accepted by her sentence generator Halogen. The output of this generator can then be automatically compared with the PTB sentence from which the corresponding input was derived. Similarly, Callaway (**?**) builds an evaluation benchmark by transforming PTB trees into a format suitable for the KPML realiser he uses.

In all of the above cases, the data is derived from real world sentences, thereby exemplifying "real world complexity". If the corpus is large enough (as in the case of the PTB), the data can furthermore be expected to cover a broad range of syntactic phenomena. Moreover, the data, being derived from real world sentences, is not biased towards system-specific capabilities. Nonetheless, there are also limits to these approaches.

First, they fail to support graduated performance testing on constructs such as intersective modifiers or lexical ambiguity, which are known to be problematic for surface realisation.

Second, the construction of the benchmark is in both cases time consuming. In the reversible approach, for each input sentence, the correct interpretation must be manually selected from among the semantic formulae produced by the parser. As a side effect, the constructed benchmarks remain relatively small (825 in the case of White (**?**); 130 in Carroll and Oepen (**?**)). In the case of a benchmark derived by transformation from a syntactically annotated corpus, the implementation of the converter is both time-intensive and corpus-bound. For instance, Callaway (**?**) reports that the implementation of such a processor for the SURGE realiser was the most time-consuming part of the evaluation with the resulting component containing 4000 lines of code and 900 rules.

As we shall show in the following sections, the GENSEM approach to benchmark construction aims to address both of these shortcomings. By using a DCG to implement grammar traversal, it permits both a full automation of the benchmark creation and some control over the type and the distribution of the benchmark items.

3 GenSem

As mentioned above, GENSEM is a grammar traversal algorithm for TAG. We first present the specific TAG used for traversal, namely SEMXTAG (**?**) (section 3.1). We then show how to automatically derive a DCG that describes the derivation trees of this grammar (section 3.2). Finally, we show how this DCG encoding permits generating formulae while enabling control over the set of semantic representations to be produced (section 3.3).

3.1 SemXTAG

The SEMXTAG grammar used by GENSEM and by the two surface realisers is a Feature-Based Lexicalised Tree Adjoining Grammar augmented with a unification-based semantics as described by Gardent and Kallmeyer (**?**). We briefly introduce each of these components and describe the grammar coverage.

FTAG. A Feature-based TAG (**?**) consists of a set of (auxiliary or initial) elementary trees and of two tree-composition operations: substitution and adjunction. Initial trees are trees whose leaves are labelled with substitution nodes (marked with a downarrow) or terminal categories. Auxiliary trees are distinguished by a foot node (marked with a star) whose category must be the same as that of the root node. Substitution inserts a tree onto a substitution node of some other tree while adjunction inserts an auxiliary tree into a tree. In an FTAG, the tree nodes are furthermore decorated with two feature structures (called **top** and **bottom**) which are unified during derivation as follows. On substitution, the top of the substitution node is unified with the top of the root node

[2]Combinatory Categorial Grammar

of the tree being substituted in. On adjunction, the top of the root of the auxiliary tree is unified with the top of the node where adjunction takes place; and the bottom features of the foot node are unified with the bottom features of this node. At the end of a derivation, the top and bottom of all nodes in the derived tree are unified. Finally, each sentence derivation in an FTAG is associated with both a **derived tree** representing the phrase structure of the sentence and a **derivation tree** recording how the corresponding elementary trees were combined to form the derived tree.

FTAG with semantics. To associate semantic representations with natural language expressions, the FTAG is modified as proposed by Gardent and Kallmeyer (**?**).

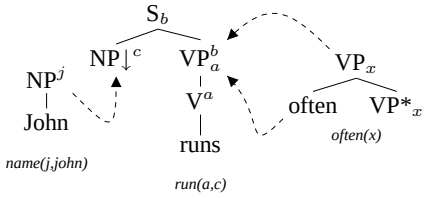

Figure 1: Flat semantics for "John often runs"

Each elementary tree is associated with a flat semantic representation. For instance, in Figure 1,[3] the trees for *John*, *runs*, and *often* are associated with the semantics *name(j,john)*, *run(a,c)*, and *often(x)*, respectively. Importantly, the arguments of a semantic functor are represented by unification variables which occur both in the semantic representation of this functor and on some nodes of the associated syntactic tree. For instance in Figure 1, the semantic index c occurring in the semantic representation of *runs* also occurs on the subject substitution node of the associated elementary tree. The value of semantic arguments is determined by the unifications resulting from adjunction and substitution. For instance, the semantic index c in the tree for *runs* is unified during substitution with the semantic index labelling the root node of the tree for *John*. As a result, the semantics of *John often runs* is {name(j,john),run(a,j),often(a)}.

SemXTAG. SEMXTAG is an FTAG for English augmented with a unification-based compositional semantics of the type described above. Its syntactic coverage approaches that of XTAG, the FTAG developed for English by the XTAG group (**?**). Like this grammar, it contains around 1300 elementary trees and covers auxiliaries, copula, raising and small clause constructions, topicalization, relative clauses, infinitives, gerunds, passives, adjuncts, ditransitives and datives, ergatives, it-clefts, wh-clefts, PRO constructions, noun-noun modification, extraposition, sentential adjuncts, imperatives and resultatives.

3.2 Converting SemXTAG to a DCG

We would like to be able to traverse SEMXTAG in order to generate semantic representations that are licensed by it. In the DCG formalism, a grammar is represented as a set of Prolog definite clauses, and Prolog's query mechanism provides built-in grammar traversal. We take advantage of this by deriving a DCG from SEMXTAG and then using Prolog queries to generate semantic representations that are associated with sentences in the language described by it.

Another advantage of the DCG formalism is that arbitrary Prolog goals can be inserted into a rule, to constrain when the rule applies or to bind variables occurring in it. We use this to ground derivations with lexical items, which are represented using Prolog assertions. We also use it to control Prolog's grammar traversal in such a way as to generate sets of semantic formulae covering certain computationally interesting cases (see section 3.3).

Our algorithm for converting SEMXTAG to a DCG is inspired by Schmitz and Le Roux (**?**), who derive from an FTAG a feature-based regular tree grammar (RTG) whose language is the derivation trees of the FTAG. Indeed, in our implementation, we derive a DCG from such an RTG, thereby taking advantage of a SEMXTAG-to-RTG converter previously implemented by Sylvain Schmitz.

[3] C^x/C_x abbreviate a node with category C and a top/bottom feature structure including the feature-value pair { **index :** x}.

TAG to RTG. In the conversion to RTG[4], each elementary tree in SEMXTAG is converted to a rule that models the contribution of the tree to a TAG derivation. A TAG derivation involves the selection of an initial tree, which has some nodes requiring substitution and some permitting adjunction. Let us think of the potential adjunction sites as requiring, rather than permitting, adjunction, but such that the requirement can be satisfied by 'null' adjunction. Inserting another tree into this initial tree satisfies one of the substitution or adjunction requirements, but introduces some new requirements into the resulting tree, in the form of its own substitution nodes and adjunction sites.

Thus, intuitively, the RTG representation of a SEMXTAG elementary tree is a rule that rewrites the satisfied requirement as a local tree whose root is a unique identifier of the tree and whose leaves are the introduced requirements. A requirement of a substitution or adjunction of a tree of root category X is written as X_S or X_A, respectively. Here, for example, is the translation to RTG of the TAG tree (minus semantics) for *runs* in Figure 1, using the word anchoring the tree as its identifier (the superscripts abbreviate feature structures: b/t refers to the bottom/top feature structure and the upper case letters to the semantic index value, so $[idx : X]$ is abbreviated to X):

$$S_S^{[t:T]} \rightarrow runs(S_A^{[t:T,b:B]}\ NP_S^{[t:C]}\ VP_A^{[t:B,b:A]}\ V_A^{[t:A]})$$

The semantics of the SEMXTAG tree are carried over as-is to the corresponding RTG rule. Further, the feature structures labelling the nodes of SEMXTAG trees are carried over to the RTG rules so as to correctly interact with substitution and adjunction (see Schmitz and Le Roux (**?**) for more details on this part of the conversion process).

To account for the optionality of adjunction, there are additional rules allowing any adjunction requirement to be rewritten as the symbol ϵ, a terminal symbol of the RTG.

The terminal symbols of the RTG are thus the tree identifiers and the symbol ϵ, and its non-terminals are X_S and X_A for each terminal or non-terminal X of SEMXTAG.

RTG to DCG. Since the right-hand side of each RTG rule is a local tree – that is, a tree of depth no more than one – we can flatten each of them into a list consisting of the root node followed by the leaves without losing any structural information. This is the insight underlying the RTG-to-DCG conversion step. Each RTG rule is converted to a DCG rule that is essentially identical except for this flattening of the right-hand side. Here is the translation to DCG of the RTG rule above[5]:

```
rule(s,init,Top,Bot,Sem;S;N;VP;V)
  -->  [runs],
       {lexicon(runs,n0V,[run])},
       rule(s,aux,Top,[B],S),
       rule(np,init,[C],_,N),
       rule(vp,aux,[B],[A],VP),
       rule(v,aux,[A],_,V),
       {Sem =.. [run,A,C]}.
```

We represent non-terminals of the DCG using the `rule` predicate, whose five (non-hidden)[6] arguments, in order, are the category, the subscript (`init` for subscript S, `aux` for subscript A), the *top* and *bottom* feature values, and the semantics. Feature structures are represented using Prolog lists with a fixed argument position for each attribute in the grammar (in this example, only the index attribute). The semantics associated with the left-hand-side symbol (here, `Sem;S;N;VP;V`, with the semicolon representing semantic conjunction) are composed of the semantics associated with this rule and those associated with each of the right-hand-side symbols.

The language of the resulting DCG is neither the language of the RTG nor the language of SEMXTAG, and indeed the language of the DCG does not interest us but rather its derivation trees. These are correlated one-to-one with the trees in the language described by the RTG, i.e. with the derivation trees of SEMXTAG, and the latter can be trivially reconstructed from the DCG derivations. From a SEMXTAG derivation tree, one can

[4]For a more precise description of the FTAG to RTG conversion see Schmitz and Le Roux (**?**).

[5]In practice, the lexicon is factored out, so there is no rule specifically for *runs*, but one for intransitive verbs (n0V) in general. Each rule hooks into the lexicon, so that a given invocation of a rule is grounded by a particular lexical item.

[6]The $-->$ notation is syntactic sugar for the usual Prolog $:-$ definite clause notation with two hidden arguments on each predicate. The hidden arguments jointly represent the list of terminals dominated by the symbol.

compose the semantic representation of the associated sentence, and in fact this semantic composition occurs as a side effect of a Prolog query against the DCG, allowing semantic representations licensed by SEMXTAG to be returned as query results.

We define a Prolog predicate for querying against the DCG, as follows. Its one input argument, `Cat`, is the label of the root node of the derivation tree (typically `s`), and its one output argument, `Sem`, is the semantic representation associated with that tree[7].

```
genSem(Cat,Sem) :-
    rule(Cat,init,_,_,Sem,_,[]).
```

3.3 Control parameters

In order to give the users some control over the sorts of semantic representations that they get back from a query against the DCG, we augment the DCG in such a way as to allow control over the TAG family[8] of the root tree in the derivation tree, over the number and type of adjunctions in the derivation, and over the depth of substitutions. To implement control over the family is quite simple: we need merely to index the DCG rules by family and modify the GENSEM call accordingly. For instance, the above DCG rule becomes :

```
rule(s,init,Top,Bot,n0V,Sem;S;NP;VP;V)
  -->  [runs],
       {lexicon(runs,n0V,[run])},
       ...
```

We implement restrictions on adjunctions by adding an additional argument to the grammar symbols, namely a vector of non-negative integers representing the number of non-null adjunctions of each type that are in the derivation subtree dominated by the symbol. By 'type' of adjunction, we mean the category of the adjunction site. In DCG terms, a non-null adjunction of a category `X` is represented as the expansion of an `x/aux` symbol other than as ϵ. So, for example, a DCG symbol associated with the vector `[1,0,0,0,0]`, where the five dimensions of the vector correspond to the `n`, `np`, `v`, `vp`, and `s` categories, respectively, dominates a subtree containing exactly one `n/aux` symbol expanded by a non-epsilon rule, and no other `aux` symbol expanded by a non-epsilon rule. We link the vector associated with the root of the derivation to the query predicate.

We define a special predicate to handle the divvying up of a mother node's vector among the daughters, taking advantage of the fact that the DCG formalism permits the insertion of arbitrary Prolog goals into a rule.

Finally, we add an additional argument to the DCG rule and to the GENSEM's call to control the traversal depth with respect to the number of substitutions applied. The overall depth of each derivation is therefore constrained both by the user defined adjunctions and substitution depth constraints.

Our query predicate now has four input arguments and one output argument:

```
genSem(Cat,Fam,[N,NP,V,VP,S],Dth,Sem):-
    rule(Cat,init,_,_,Fam,
         [N,NP,V,VP,S],Dth,Sem,_,[]).
```

4 Using GENSEM for benchmarking

We now show how GENSEM can be put to work for comparing two TAG-based surface realisers, namely GENI (**?**) and RTGEN (**?**). These two realisers follow globally similar algorithms but differ in several respects. We show how GENSEM can be used to produce benchmarks that are tailored to test hypotheses about how these differences might impact performance. We then use this GENSEM-generated benchmark to compare the performance of the two realisers.

4.1 GenI and RTGen

Both GENI and RTGEN use the SEMXTAG grammar described in section 3.1. Moreover, both realisers follow an algorithm pipelining three main phases. First, **lexical selection** selects from the grammar those elementary trees whose semantics subsumes part of the input semantics. Second, the **tree combining** phase systematically tries to combine trees using substitution and adjunction. Third, the **retrieval phase** extracts the yields of

[7] The 6th and 7th arguments of the rule call are the hidden arguments needed by the DCG.

[8] TAG families group together trees which belong together, in particular, the trees associated with various realisation of a specific subcategorisation type. Thus, here the notion of TAG family is equivalent to that of subcategorisation type.

the complete derived trees, thereby producing the generated sentence(s).

There are also differences however. We now spell these out and indicate how they might impact the relative performance of the two surface realisers.

Derived vs. derivation trees. While GENI constructs derived trees, RTGEN uses the RTG encoding of SEMXTAG sketched in the previous section to construct derivation trees. These are then unraveled into derived trees at the final retrieval stage. As noted by Koller and Striegnitz (**?**), these trees are simpler than TAG elementary trees, which can favourably impact performance.

Interleaving of feature constraint solving and syntactic analysis. GENI integrates in the tree combining phase a filtering step in which the initial search space is pruned by eliminating from it all combinations of TAG elementary trees that cover the input semantics but cannot possibly lead to a valid derived tree. This filtering eliminates all combinations of trees such that either the category of a substitution node cannot be cancelled out by that of the root node of a different tree, or a root node fails to have a matching substitution site. Importantly, filtering ignores feature information and tree combining takes place after filtering. RTGEN, on the other hand, directly combines derivation trees decorated with full feature structure information.

Handling of intersective modifiers. GENI and RTGEN differ in their strategies for handling modification.

Adapting Carroll and Oepen's (**?**) proposal to TAG, GENI adopts a two-step tree-combining process such that in the first step, only substitution applies, while in the second, only adjunction is used. Although the number of intermediate structures generated is still 2^n for n modifiers, this strategy has the effect of blocking these 2^n structures from multiplying out with other structures in the chart.

RTGEN, on the other hand, uses a standard Earley algorithm that includes sharing and packing. Sharing allows intermediate structures common to several derivations to be represented once only while packing groups together partial derivation trees with identical semantic coverage and similar combinatorics (same number and type of substitution and adjunction requirements), keeping only one representative of such groups in the chart. In this way, intermediate structures covering the same set of intersective modifiers in a different order are only represented once and the negative impact of intersective modifiers is lessened.

4.2 Two GENSEM benchmarks

We use GENSEM to construct two benchmarks designed to test the impact of the differences between the two realisers and, more specifically, to compare the relative performance of both realisers (i) on cases involving intersective modifiers and (ii) on cases of varying overall complexity.

The MODIFIERS benchmark focuses on intersective modifiers and contains semantic formulae corresponding to sentences involving an increasing number of modifiers. Recall that GENSEM calls are of the form *gensem(Cat,Family,[N,NP,V,VP,S],Dth,Sem)* where *N,NP,V,VP,S* indicates the number of required adjunctions in *N*, *NP*, *V*, *VP* and *S*, respectively, while *Family* constrains the subcategorisation type of the root tree in the derivations produced by GENSEM. To produce formulae involving the lexical selection of intersective modifiers, we set the following constraints. *Cat* is set to s and *Family* is set to either n0V (intransitive verbs) or n0Vn1 (transitive verbs). Furthermore, N and VP vary from 0 to 4 thereby requiring the adjunction of 0 to 4 N and/or VP modifiers. All other adjunction counters are set to null. To avoid producing formulae with identical derivation trees but distinct lemmas, we use a restricted lexicon containing one lemma of each syntactic type, e.g. one transitive verb, one intransitive verb, etc. Given these settings, GENSEM produces 1 789 formulae whose adjunction requirements vary from 1 to 6. For instance, the semantic formula *{sleep(b,c),man(c),a(c),blue(c),sleep(i,c),carefully(b)}* (*A sleeping blue man sleeps carefully*) extracted from the MODIFIERS benchmark contains two NP adjunctions and one VP adjunction.

The MODIFIERS benchmark is tailored to focus on cases involving a varying number of intersective modifiers. To support a comparison of

the realisers on this dimension, it displays little or no variation w.r.t. other dimensions, such as verb type and non-modifying adjunctions.

To measure the performance of the two realisers on cases of varying overall complexity, we construct a second benchmark (COMPLEXITY) displaying such variety. The GENSEM parameters for the construction of this suite are the following. The verb type (*Family*) is one of 28 possible verb types[9]. The number and type of required adjunctions vary from 0 to 4 for N adjunctions, 0 to 1 for NP, 0 to 4 for VP and 0 to 1 for S. The resulting benchmark contains 890 semantic formulae covering an extensive set of verb types and of adjunction requirements.

4.3 Results

Using the two GENSEM-generated benchmarks, we now compare GENI and RTGEN. We plot the average number of chart items against both the number of intersective modifiers present in the input (Figure 3) and the size of the Initial Search Space (ISS), i.e., the number of combinations of elementary TAG trees covering the input semantics to be explored after the **lexical selection** step (Figure 2). In our case, the ISS gives a more meaningful idea about the complexity than considering only the number of input literals. In an FTAG, the number of elementary trees selected by a given literal may vary considerably depending on the number and the size of the tree families selected by this literal. For instance, a literal selecting the n0Vn2n1 class will select many more trees than a literal selecting the n0V family because there are many more ways of realising the three arguments of a ditransitive verb than the single subject argument of an intransitive one. Chart items include all elementary trees selected by the lexical selection step as well as the intermediate and final structures produced by the tree combining phase. In RTGEN, we distinguish between the number of structures built before unpacking (packed chart) and the number of structures obtained after unpacking (unpacked chart).

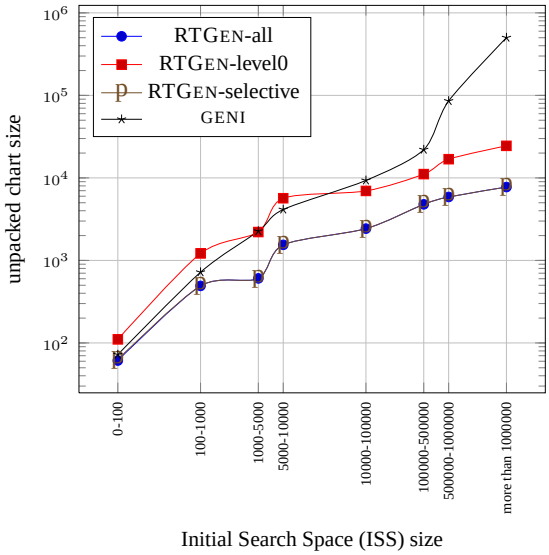

Figure 2: Performance of realisation approaches on the COMPLEXITY benchmark, average unpacked chart size as a function of the ISS complexity.

Both realisers are implemetned in different programming languages, GENI is implemented in Haskell whereas RTGEN in Prolog. As for the time results comparison, preliminary experiments show that GENI is faster is simple input cases. On the other hand, in the case of more complex cases, the point of producing much less intermediate results pays off compared to the overhead of the chart/agenda operations.

Overall efficiency. The plot in Figure 2 shows the results obtained by running both realisers on the COMPLEXITY benchmark. Recall (cf. section 4.2) that the COMPLEXITY benchmark contains input with varying verb arity and a varying number of required adjunctions. Hence it provides cases of increasing complexity in terms of ISS to be explored. Furthermore, test cases in the bench-

[9]The 28 verb types are En1V,n0BEn1,n0lVN1Pn2,n0V,n0Va1,n0VAN1,n0VAN1Pn2, n0VDAN1,n0VDAN1Pn2,n0VDN1,n0VDN1Pn2,n0Vn1, n0VN1,n0Vn1Pn2,n0VN1Pn2,n0Vn2n1,n0Vpl,n0Vpln1, n0Vpn1,n0VPn1,n0Vs1,REn1VA2,REn1VPn2,Rn0Vn1A2, Rn0Vn1Pn2,s0V,s0Vn1,s0Vton1. The notational convention for verb types is from XTAG and reads as follows. Subscripts indicate the thematic role of the verb argument. n indicates a nominal, Pn a PP and s a sentential argument. pl is a verbal particle. Upper case letters describe the syntactic functor type: V is a verb, E an ergative, R a resultative and BE the copula. Sequences of upper case letters such as VAN in n0VAN1 indicate a multiword functor with syntactic categories V, A, and N. For instance, n0Vn1 indicates a verb taking two nominal arguments (e.g., *like*) and n0VAN1 a verb locution such as *to cry bloody murder*.

mark trigger sentence realisation involving certain TAG families, which have a certain number of trees. Those trees within a family often have identical combinatorics but different features. Consequently, the COMPLEXITY benchmark also provides an appropriate testbed for testing the impact of feature structure information on the two approaches to tree combination.

The graphs show that as complexity increases, the performance delta between GENI and RTGEN increases. We conjecture that as complexity grows, the filtering used by GENI does not suffice to reduce the search space to a manageable size. Conversely, the overhead introduced by RTGEN's all-in-one, tree-combining Earley with packing strategy seems compensated throughout by the construction of a derivation rather than a derived tree and pays off increasingly as complexity increases.

Modifiers. Figure 3 plots the results obtained by running the realisers on the MODIFIERS benchmark. Here again, RTGEN outperforms GENI and the delta between the two realisers grows with the number of intersective modifiers to be handled. A closer look at the data shows that the global constraints set by GENSEM on the number of required adjunctions covers an important range of variation in the data complexity. For instance, there are cases where 4 modifiers modify the same NP (or VP) and cases where the modifiers are distributed over two NPs. Similarly, literals introduced into the formula by a GENSEM adjunction requirement vary in terms of the number of auxiliary trees whose selection they trigger. The steep curve in GENI's plot suggests that although the delayed adjunction mechanism helps in avoiding the proliferation of intermediate incomplete modifiers' structures, the lexical ambiguity of modifiers still poses a problem. In contrast, RTGEN's packing uniformly applies to word order variations and to the cases of lexical ambiguity raised by intersective modifiers because the items have the same combinatoric potential and the same semantics.

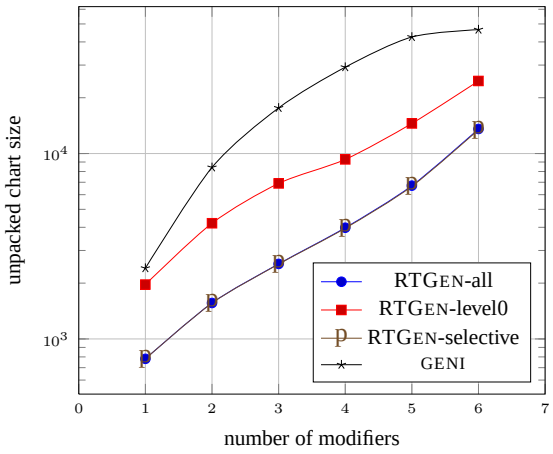

Figure 3: Performance of realisation approaches on the MODIFIERS benchmark, average unpacked chart size as a function of the number of modifiers.

5 Conclusion

Surface realisers are complex systems that need to handle diverse input and require complex computation. Testing raises among other things the issue of coverage – how can the potential input space be covered? – and of test data creation – should this data be hand tailored, created randomly, or derived from real world text?

In this paper, we presented an approach which permits automating the creation of test input for surface realisers whose input is a flat semantic formula. The approach differs from other existing evaluation schemes in two ways. First, it permits producing arbitrarily many inputs. Second, it supports the construction of grammar-controlled, linguistically focused benchmarks.

We are currently working on further extending GENSEM with more powerful (recursive) control restrictions on the grammar traversal; on combining GENSEM with tools for detecting grammar overgeneration; and on producing a benchmark that could be made available to the community for testing surface realisers whose input is either a dependency tree or a flat semantic formula.