Automatic Test Suite Generation

Doug Arnold, Dave Moffat, Louisa Sadler and Andrew Way,
Department of Language and Linguistics,
University of Essex,
Wivenhoe Park,
Colchester, Essex,
CO4 3SQ, U.K.

email: doug@essex.ac.uk

March 26, 1998

Abstract

A Test Suite (TS) is typically a collection of Natural Language sentences against which the coverage of a Natural Language Processing system can be evaluated. We describe a method by which such suites can be produced automatically, involving a modification and extension of the Definite Clause Grammar formalism, and describe some of the advantages of the method over the traditional method of manual construction. In particular, it turns out that automatically constructed TSs are likely to contain fewer errors than those constructed by hand.

1 Introduction

Apart from being an interesting area in its own right, evaluation of Natural Language Processing (NLP) systems is an essential part of the development process, since it provides the only possible check that a system is improving as it evolves through time. Evaluation can involve real users, corpora, or Test Suites (TSs). In this paper we describe, and begin to evaluate a method for automatically generating TSs.

An NLP TS is a collection of NL sentences against which the coverage of NLP systems can be examined. It has a similar role to a corpus of NL sentences,

and as such is a useful part of a tool box for NLP system developers and evaluators. Unlike a corpus, however, it is not naturally produced by NL speakers for independent purposes, but generated by researchers specifically for testing purposes. Typically, TSs are hand-constructed, which is both extremely tedious and error-prone, which suggests that the process is a good candidate for automation. This paper substantiates this suggestion.

The remainder of the paper is organized as follows. Section 2 describes some general issues to do with TSs, and sets out the desiderata that an automatic TS construction process should satisfy. In section 3, a computational technique based on the Prolog version of the Definite Clause Grammar formalism is described. Essentially, this involves a modification of the standard Prolog generation strategy so that sentences are produced in a reasonable order, with an extension that involves a separate 'filtering' component, which allows unwanted derivations and the associated strings to be distinguished (e.g. separated out, or simply excluded from the TS).

The third section of the paper summarises the results of a practical investigation of the approach, where a number of TSs produced automatically by this technique were compared with one produced manually by one of the authors [5, 6], for independent purposes. Not only did this confirm that TSs for various purposes could be produced and modified very rapidly, the comparison showed that the original TS, despite being carefully constructed over a period of weeks, and being used in practice over a longer period, contained several previously unnoticed errors. Not only were many grammatical sentences missed out (some of them simple, and as short as only three words), but more surprisingly some ungrammatical sentences were included. The automatically generated TS was therefore clearly better. A further advantage is that it is easy in this way to gen-

erate TSs of ungrammatical, or marginally grammatical strings, which appear nearly impossible to produce by hand, but which are important in testing some kinds of system. This suggests that the approach may provide an important addition to the system developers 'tool box'.

The final section summarises the conclusions and considers some open questions and lines of investigation.

2 The Use of Test Suites

A Test Suite (TS) is a set of sentences which individually represent specified Natural Language constructions and combinations of constructions (e.g. the combination of Dative-shift and Passive), and which provides a basis for evaluating the performance of a Natural Language Processing system. TSs provide a simple kind of reference point, or benchmark for system coverage and performance, especially as a system evolves through time, and are thus a useful part of the system developers' and evaluators' 'tool box'.[1] (For example, from a system developer's point of view, comparison with a TS provides the only safe way of deciding whether changes to a system constitute improvements, or whether one's efforts are actually making the system worse).

Given this, one might expect there to be a large number of actual TSs, and a well established methodology for TS creation. However, this is not the case: there seems to be only one TS which is generally available, that of [2], and in practice, testing of system coverage is often carried out in a haphazard fashion.

[1] In fact, the background to, and motivation for the work reported here was the need to construct TSs for monitoring development work on components of the Eurotra MT system (see, e.g. [1, 4] for general discussion), and for evaluation of a number of practical MT systems [5, 6]

The use of TSs is subject to two major criticisms. First, even if a TS is constructed which adequately reflects the kinds and combinations of construction that a system is likely to face, it will still be unrepresentative, in the sense that it will not reflect their relative frequencies and importance. This point is well taken. However, the obvious alternative, which is to use a corpus as a basis for testing, also suffers from problems of representativeness, since any particular corpus will typically omit certain important (combinations of) constructions. In general, a properly constructed TS has the advantage of systematicity, since each relevant phenomenon will occur exactly once. Second, there is a tendency for use of TSs to focus attention on syntactic and semantic phenomena, at the expense of lexical coverage. This criticism seems less well-founded – there is a sense in which estimation of lexical coverage really is less problematic, since lexical omissions are typically easier to remedy. In any case, these criticisms simply show that a TS is not the *only* tool required for system evaluation.

In practice, constructing a TS involves attempting to generate a set of sentences which exemplify the constructions which it is intended that the system cover (normally inferable from the system specification, together with some investigation of corpora which supposedly represent the kind of input the system will receive). The process is difficult, (requiring considerable linguistic sophistication and skill), laborious, tedious, (and above all) error prone.

For example, in a simple case, one may want a TS that addresses the coverage of a system with respect to **Tense, Modality, Passive, Negation** and **Subcategorization**. Linguistic sophistication is involved in finding examples that realise the various combinations of phenomena. The process is laborious, because it involves trying to instantiate *all* combinations, and tedious because of the sheer number of examples involved (the suite originally developed in

[6] for the very limited number of constructions just mentioned contained 488 sentences). It is error prone, simply because of the number and complexity of interactions that may (or may not) be realized. As will appear below, the TS in [6], though carefully checked over a period of weeks, turned out to contain a number of errors and omissions.

In general, the alternative of taking an existing TS and trying to modify it so that it meets one's needs does not greatly simplify the task. Apart from the scarcity of existing TSs to modify, this is because any existing TS is likely to be inappropriate in various ways. Apart from minor details like differences of vocabulary, and the possibility that such a TS may contain errors, it is likely that an existing TS will fail to instantiate important (combinations of) constructions, and will contain constructions which are irrelevant for a particular application (e.g. not every application may require that a system handle interrogatives), or will be generally too ambitious, given the state of the art in a particular application (e.g. the TS given in [2] reflects the state of the art in syntactic and semantic parsing, but it is probably still too ambitious for most MT systems)[2].

With this in mind, one can see the following important desiderata for a TS:

- It must be *appropriate* to the system in question, or to the particular stage of the system in question. The constructions, including vocabulary, contained should reflect the domain, and purpose of the associated system, and its stage of development. Notice that this means that construction

[2] A special case of this arises when one is trying to adapt a TS from a different language. It obviously will not do simply to translate the TS, since this can easily produce a TS that completely ignores complexities in the second language, such as variations in word order, to take an obvious example.

of TSs must be an on-going process.

- It must be *complete*. For example, if a TS is intended to test coverage of Passive and Dative constructions, it must include all the (grammatical) combinations.

- It must be *correct*. A TS which is supposed to exemplify grammatical possibilities should not contain any that are ungrammatical.

- It should be *systematic*. It is important that the sentences in a TS are grouped together, or ordered in some sensible way, from the point of view of the users of the TS.

- It should not be limited to positive instances (e.g. grammatical sentences). It is sometimes assumed, at least implicitly, that a TS should be a collection of grammatical sentences, rather in the manner of an error-free corpus, but this is not correct. Ideally, a TS should consist of a number of different *kinds* of example, depending to some extent on the purpose of the associated system. This point is obvious if one considers a system such as a style or spelling checker, where the ability to correctly classify examples as 'good' or 'bad' is critical, but it holds generally. For example, for some kinds of analysis system it may be useful for part of the TS to consist of ungrammatical strings, so that one can check that the system rejects them (or accepts them, as a test of the robustness of the system). Similarly, it may be useful to have a collection of ungrammatical, or marginally grammatical examples whose absence from the yield of a generator is important. In an application like MT one may wish to have a TS of examples which one should be able to analyse and translate, with a special subset that one can analyse, translate, and synthesise. Quite

> generally, the behaviour of a system should not be judged only on the things it handles, but on the things it fails to handle, and on the fineness of its ability to discriminate.

To summarize, construction of TSs is an important part of system development process. It is both demanding, and tedious; it requires careful attention to detail, in checking the individual sentences, as well as a global view of organisation, and an ability to keep track of very large numbers of interacting factors. All of this makes it an excellent candidate for automation.

2.1 Automatic Generation of Test Suites

For automatic construction of TSs to be viable a number of constraints must be satisfied.

One must be able to make a reasonably formal description of the kind of linguistic skill involved. Fortunately, this skill is not very different from that involved in normal grammar construction, so, subject to one caveat, one is simply faced with the problem of designing an appropriate grammar formalism.

The caveat is this: it must be *much* easier to specify a TS than it is to create the sort of system one is intending to evaluate (call this the 'target' system). A consequence of this is that the formalism chosen must be simple, and familiar to potential users, and widely available. This is partly just an obvious question of time and resources. But there is also a question of complexity: the TS construction system must be simple enough that one can be reasonably sure that its behaviour satisfies the desiderata given above. This will not be the case for a typical 'target' system (if it was, one would have less need of TSs). Fortunately, this *is* reasonably easy to guarantee. First, because the task one

has in mind, i.e. generation, and perhaps classification, of sentences (strings) is a much easier application than most (compare it with question answering, or MT, for example). Second, because considerations like efficiency are less important than for many applications (e.g. there will not be much objection to a TS construction system that operates in batch mode).

The TS construction system must not undergenerate, since it is very hard for a human to check for omissions. However, it can overgenerate, since (a) a human checker will typically find it easy to locate ungrammatical sentences, and (b) in any case, collections of marginal and ungrammatical examples are a useful part of a TS.

In the rest of the paper we present and discuss an experimental TS generator which we have built.

3 The Generator and Grammar

The simplest approach to the automatic generation of TSs is to write a simple CFG grammar that has the specified coverage and an associated generator.

There are a number of special issues which arise in the automatic generation of TSs (for example, the need to put upper bounds on the numbers of strings generated, in a sensible way, and the practical usefulness of specifying a particular ordering for the output set of sentences), as well as issues of general concern (for example, the need to exclude left-recursive loops in a top-down generation algorithm).

As far as the grammar is concerned, keeping it context-free is important, because this makes it easy to write. But writing a grammar for purposes of TS generation is special in a number of respects, especially concerning the useful-

ness of generating what is actually ungrammatical output in terms of the object NL. Likewise, we are interested only in the stringset yield of the grammar, and not in the associated structures.

In this section we describe this approach and discuss a number of these issues.

3.1 The Generator

The generator is required to take a simple grammar and generate strings (usually sentences) with it, comprehensively and in an ordered sequence.

Since in general we expect the grammar to define an infinite set of strings, some scheme must be adopted whereby the more "interesting" ones are generated first, so that the generation can be stopped when the strings being produced become less useful for testing.

We have chosen an established simple and flexible technology to represent and process the grammar: Prolog DCGs [8]. The backbone of a DCG grammar is just a context-free rewriting system, but its augmentation with Prolog term unification makes it practically more powerful; and the possible inclusion of arbitrary calls to Prolog providing a further extension.

It was also important to us that the technology is widely available, so that a generator based on it would be easily transportable to anybody who might wish for a tool to help them test their NLP system.

One problem with the usual DCG processing strategy is that it will loop on a left-recursive grammar, because the search method it uses is the most basic top-down one. To deal with this, and also to help ensure that the simpler, more interesting and typical strings should be produced first, we chose a variant of a depth-first iterative-deepening search strategy. If the depth of a derivation

tree is taken to be the number of derivations or rewrite-rules used in its longest branch, then the generator first finds all strings of depth 1, then those of depth 2, and so on up to the maximum specified by the user. The user may also specify the minimum, in fact. The results reported in this paper were arrived at by simply setting the minimum and maximum to be the same, and finding all strings at that level of the search space, and then incrementing the counters to find the next most complex strings, and so on.[3]

3.2 The Grammar

The grammar we used is a simple CFG approach written in DCG notation. We decided that a DCG would be a good candidate for this type of exercise as it is simple to write and could be expected to overgenerate (with respect to the object NL) quite considerably. This is normally the crucial problem to solve in the field of generation [7], but it is actually useful here, given the value of negative instances for testing synthesis components, for example, in MT. The grammar we used to test the feasibility of our approach in an experiment in English TS generation was very simple, being closely based on that in [3], pp. 114-116.

It should be noted that the purpose of TSs such as those we are interested in generating automatically is essentially to test grammatical coverage – that is, the coverage (by the application grammar under development) of the syntactic constructions of the language in question, rather than the scope or sensitivity

[3] The platform we used for the system was as follows. The machine was a Sun4 Sparcstation-1, the Prolog was the SICStus interpreter, version 0.6. The generator itself was also written in the form of an interpreter. Development of a partially-executable version is underway, and this should speed it up considerably.

of the lexical coverage. Essentially we are interested in the *types* of sentences produced, *not* the *tokens*, and this can be achieved with a quite restricted lexical coverage.

3.3 The Filter Mechanism

On top of this basic machinery (a depth-regulated generator and a simple grammar) an extra feature was built in, by which certain types of production could be effectively filtered. That is, they could be marked as ungrammatical, or in some sense "special", by the user. For instance, perhaps because they involve some particular grammatical construction that the user has no wish to test because the NLP system being benchmarked cannot (as yet) cope with it. The approach we are investigating therefore conceptually involves a two stage process of (over) generation and filtering. It is important to note that this overgeneration can be of various different types (in terms of the relative coverage of TS grammar and object grammar for example, or straightforward overgeneration with respect to the object NL).

Of course, the user might in some cases be able to change the TS grammar so that it no longer produces the unwanted strings, but still produces everything else. This is not a general solution, however. For one thing, we see a role for a filtering mechanism in permitting the production of strings which are labelled as being ungrammatical in the object NL (cf above). Furthermore, we wanted the system to be usable by someone without much particular knowledge of the DCG formalism. The idea is that such a user would be able to add filters with little knowledge of the DCG formalism and grammar. We also had an eye on using the grammar and generator system for benchmarking purposes, in which case the core of the TS grammar should of course be inviolable.

Another reason was that while adding a filter post-hoc to the end of the generator's production of a single string (or actually part-way through, as we saw a way to manage that) could be guaranteed to rule out just those productions specified by the filter, on the other hand changing the grammar itself to achieve the same end might well inadvertently also cut out other valid productions, and this would be a difficult thing for the user to judge. To change the whole grammar requires global knowledge of it all, while to add a filter simply required local knowledge; that is, knowledge of the structure of the first unwanted production seen.

The generator also produces, at the user's request, a linguistic structure for any strings it produces. The user can use this structure to abstract out of it the offending features that cause the string to be considered a starred one, and can inform the generator of this so that it will in future produce no more strings of that type. (Or the user might prefer to divert the starred or filtered productions to an alternative cache of negative or ungrammatical strings, as these can also be made to test an NLP system, conversely.)

A variety of methods of specifying these constraints on the generated strings can be imagined. To start with we chose a simple one, with a view to experimenting with more complicated schemes later if it turned out to be insufficient. The user gives the generator a Prolog term, possibly including variables, which may or may not be co-referring. The generator then notes that any production with a structure that matches this partial structure should be considered "starred". Matching is tested for by Prolog unification, but again more sophisticated types of matching are possible.

This feature of allowing certain grammatical constructions, admissible by the base DCG grammar, to be starred, and filtered out, was also done in such a way

that, if the starred strings are not wanted at all, even as negative productions, then their production is halted at the earliest possible stage. Thus execution time is saved, and depending on the starred construction, there can be quite a saving in execution time.

This feature can be viewed as an extension to the DCG grammar formalism. It remains to be seen how useful it would be in practice, in other applications, but the results of our experiments with it so far in this application have been encouraging.

4 Evaluation

As a way of testing the viability of the approach outlined above we decided to run a series of experiments with the TS generator, replicating the hand-built TS described in [6] for the purposes of comparison. A TS grammar to cover all possible permutations of 1, 2 and 3 place predicates, tenses, modality, passive and sentential negation was written (in a few hours).

Running the generator with this grammar, up to depth 9, produced a TS of some 4752 sentences. Comparison with the handbuilt TS showed a number of interesting results. We noted first of all that the original suite contained 20 strings which did not figure in the automatic TS either because they were duplicates or ungrammatical. Secondly, some 136 grammatical sentences had been omitted from the hand-built suite. Thirdly, a very large proportion (4128) of the automatically produced TS sentences were in fact ungrammatical or superfluous (as expected).

We then used the filtering capacity to control the number of unacceptable strings produced by the generator, stipulating the patterns which were not

to be generated. The following example blocks all occurrences of a string of two auxiliaries with the negation marker (i.e. 'not') occurring after the *second* auxiliary.

```
star(vp(!(_),
         aux(!(_),_),
         vp(!(_),
            aux(!(_),_),
            neg(!,
                [not]),
            vp(!(_),_)))).
```

Using the filtering capacity to prevent the production of unwanted sentences slowed down the speed of the system (because of the extra checking involved), but only 1296 strings were produced. The original suite contained 488 'legal' sentences, and we showed in the previous experiment that a further 136 sentences were omitted, giving a "full" suite of 624 sentences. This makes the grammatical generated sentences just under 50% of the total generated TS. The ungrammatical sentences were kept because we felt they would be particularly testing for an NLP parser/grammar.

In a further development, we improved the filtering mechanism, to allow the statement of more general constraints. By changing the format of the linguistic structures the generator automatically built to hold the derivation history, we permit the filter-writer to leave unspecified the number of daughters in a construction as well as their types. Even the type of the mother can be left unspecifed. Thus, permitting a Prolog variable such as DAUGHTERS1 to account for any number of daughters, as in the following example (equivalent to the previous one):

```
star( vp(A) ===> { aux(B) ===> DAUGHTERS1,
                   vp(E) ===> { aux(F) ===> DAUGHTERS2,
```

```
                                    neg ===> DAUGHTERS3,
                                    MOREDAUGHTERS} }
    ).
```

enabled the number of filters to be reduced to 12.

5 Conclusions and Further Work

In this paper we have argued that automatic TS generation is both feasible and advantageous. The chief advantages are the following:

First, automatic generation of TSs is *faster*, and *easier* than manual construction. This has a number of consequences, apart from the saving of resources. Manual construction of TSs is so difficult and tedious that there is a powerful incentive to manage with a single TS (or avoid constructing one at all). This can clearly hamper one's evaluation of a system as it develops through time.

A special instance of the ease of automatic TS construction can be seen when one tries to modify an existing TS. Doing this by hand is extremely tedious and difficult (e.g. adding Passives to a TS by hand would involve looking at every sentence, and adding a passive variant for the majority of them), but at least some changes to a TS grammar are very easy to implement (e.g. simply adding a rule, or a filter), since the system ensures that interactions with existing rules are considered. In fact, automatic generation of TSs seems so easy that it is possible to think of constructing TSs which test the effects of making even quite small changes to systems, even when the systems are very complex, and the task of producing a TS by hand which tests the interaction of the change with all the other features of the system would be so daunting that it would not be undertaken. Thus, automatic generation of TSs promises to make system construction better, as well as easier.

Second, it appears that automatically generated TSs are *better* than manually constructed ones, in the sense that manually produced TSs tend to be incomplete, and even contain errors. We observed that this was the case even with the very simple TS of [6], one would expect the improvements to be even more marked and significant when more, and more varied, phenomena are involved.

A third advantage of this approach to TS generation is that it can be used to produce "negative" TSs, of sentences that are not grammatical, but *nearly* grammatical. These are useful for NLP testing and evaluation purposes, because they are likely to challenge the target system. However, they are very much harder for a human to generate. To think of 2000 grammatical sentences, all with different structures, is very hard. But to think of the same number of *nearly* grammatical sentences, again with different structures, is a problem of an altogether greater magnitude.

Though already useful, the approach we have described has a number of limitations. In particular, the expressivity of the filtering mechanism could be improved. There are also a number of interesting possibilities if one considers TSs not as static objects, but as being generated dynamically, perhaps only when needed for testing. (The system is quick enough for this to be realistic). This opens the possibility of using the system on large lexicons, perhaps with some "randomisation", depending on an input random seed, say. This would affect the completeness of the TSs produced but might still be very useful. Exploiting the possibilities for dynamic TS construction would also make it possible to hold and transmit extremely large 'virtual' suites, in the order of many millions of sentences, providing standard benchmarks for system testing.

References

[1] Doug Arnold. Eurotra: a European Perspective on MT. *Proceedings of the IEEE*, 74(7):979–992, July 1986.

[2] Dan Flickinger, John Nerbonne, Ivan Sag, and Tom Wasow. Toward Evaluation of NLP Systems. Special Session at the *25th Annual Meeting of the Association for Computational Linguistics*, 1987.

[3] F.C.N. Pereira and S.M. Shieber. *Prolog and Natural Language Analysis.* CSLI, Stanford, 1987.

[4] A. Raw, B. Vandecapelle, and F. Van Eynde. Eurotra: An Overview. *Interface: Journal of Applied Linguistics*, 3.1:5–32, 1988.

[5] Andrew Way. Developer-Oriented Evaluation of MT Systems. Working Papers in NLP, Department of Language and Linguistics, The University of Essex, 1991.

[6] Andrew Way. A Practical Developer-Oriented Evaluation of two MT Systems. Working Papers in NLP, Department of Language and Linguistics, The University of Essex, 1991.

[7] M. Zock and G. Sabah, editors. *Advances in Natural Language Generation*, volumes 1 and 2. Pinter, London, 1988.

[8] Fernando C.N. Pereira and David H.D. Warren. Definite clause grammars for language analysis - a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13.3: 231–78, 1980.