Test Suite Design Guidelines and Methodology

Lorna Balkan, Siety Meijer,
Doug Arnold, Eva Dauphin, Dominique Estival,
Kirsten Falkedal, Sabine Lehmann, Sylvie Regnier-Prost.

Report to LRE 62-089 (D-WP2.1)

Test Suites for Natural Language Processing (TSNLP)

University of Essex

July 1994

Summary

Title	Test Suite Design — Guidelines and Methodology
Project	LRE-62-089
Document ID	D-WP2.1
WP	Test Suite Design
Date	July, 1994
Principal Authors	Lorna Balkan and Siety Meijer
Authors	Lorna Balkan, Siety Meijer, Doug Arnold,
	Dominique Estival, Kirsten Falkedal, Sabine Lehmann
	Sylvie Regnier-Prost, Eva Dauphin
Reviewers	the consortium
Status	Intermediate Report; will undergo revisions

Contents

Test Suite Design
Guidelines and Methodology

1 Introduction

This document addresses a range of issues relevant to the design and use of test suites in NLP system assessment.[1] As we concluded in deliverable D-WP1 [3], the design of test suites can be improved in order to make their application more viable and useful. Whereas some of these improvements will be worked out in future deliverables,[2] this deliverable aims at providing initial guidelines and a methodology for test suite design which will make evaluators aware of the requirements on test suites and should assist in the construction of test data. Data constructed on the basis of the TSNLP guidelines should display amongst other things a high degree of systematicity, a well-founded approach to testing ill-formed data and a relatively neutral vocabulary. Although the issue of test data annotation is the main focus of the next deliverable this document will contain an initial discussion on the subject as one of the important steps in designing reusable test suites.

In order to achieve the suggested improvements and to draw up guidelines and a methodology for test suite design which accommodate these improvements, we have looked at a great many issues that influence test suite design (see WP2.1 deliverable "Issues in Test Suite Design").

It is of course impossible to specify requirements for every type of test suite. We concentrate below on specifying the requirements for the TSNLP test suite (section 2). Likewise, in section 3, we focus on methodological issues for the TSNLP test suite, that we hope are generalisable to other scenarios. After the general discussion in sections 2 and 3, we present application specific guidelines for the three applications of the TSNLP project – parsers, grammar checkers and controlled language checkers.

It will be clear that once a 'general' or 'multi-purpose' test suite has been constructed, it will often not be immediately applicable for the testing of just any system. Amendments will need to be made to customize the test suite to the particular application/system it will be used for. When an evaluator is using the TSNLP test data and tools, these amendments will be limited. For any other type of test data more amendments will generally be needed. The issue of test suite customization is therefore addressed rather extensively in section 5.

[1] This document represents the current thinking of the consortium on the design of test suites and a first version of a methodology for their construction. The ideas expressed in this document are likely to undergo revision in the light of devising an annotation scheme for the TSNLP data in Task 2.2, and, of course, after applying the methodology to the construction of the actual test data.

[2] The annotation of test data will be worked out in the next phase of this Work Package (WP2.2), whereas tools will be produced in Work Packages 5 and 6.

2 Requirements on Test Data

2.1 Introduction

At the present state of development any NLP system has features which make it unique and specific, and all potential users of such NLP systems have specific needs and requirements of various kinds. Moreover, testing and evaluations are performed for a variety of different purposes. This leads to the assumption or postulate which underlies the TSNLP approach to test suite design, namely that *any actual test suite to be used for some given test or evaluation will have to be more or less specific* in order to yield optimally informative and interpretable results. Therefore, the notion of a monolithic and fixed general-purpose test suite seems neither feasible nor desirable.

On the other hand, there will obviously be a rather large amount of linguistic phenomena which any test suite might want to include.

The TSNLP approach to test suite design is therefore guided by the need to provide test material which is easily reusable for a variety of purposes. In order to achieve this, the traditional notion of a test suite as a monolithic list of test items has been abandoned in favour of the notion of a data base, where test items are stored and associated with various types of annotations and from which purpose specific and application specific sets of test items can be extracted by searching on specific annotation types. Reusability will further be enhanced by the provision of tools to replace lexical material with user or application specific equivalents, as well as of tools to generate additional test items.

This approach imposes a series of requirements on the test data which will be discussed below under the headings:

- General Requirements on TSNLP Test Data
- Requirements on content and coverage of the data
- Requirements on the presentation and tools for manipulation of the data

2.2 General Requirements on TSNLP Test Data

Stating the requirements on test data is not a straightforward process. As discussed in the WP2.1 "Issues in Test Suite Design" report, there is a multitude of factors which contribute their specific requirements. All of these have to be taken into consideration when specifying the database design and organisation. This is necessary, even though the actual test material to be written will only be tested against three different applications (parsers (P), grammar checkers (GC) and controlled language checkers (CL)) and three European languages (English, French, and German).

To attain this goal, we aim to produce test data which, taken together, will represent the *sum* of some set of test suites for the three specific applications. Annotations on the test items will indicate to which applications the individual test items apply.

All this determines our view of how a 'multi-purpose', 'reusable' test suite should be defined:

A test suite is a systematically organized set of test items, where each test item conceptually is a triple consisting of:

- the actual test input;
- a set of annotations characterizing the phenomenon or set of phenomena the item represents, the applications for which the item is relevant, the level of complexity, etc.;
- some specification of what would count as a correct output.

The first item of this triple, the actual test input, relates to the *content* and *coverage* of test data and will be discussed in the next section. The second and third item form part of the *presentation* of test data and will be discussed in section 2.2.2.

2.2.1 Requirements on Content of Test Data

Part of the work we did in WP1 was to define a set of characteristics to describe existing test suites. We here reproduce those characteristics which concern the content and coverage of test suites and take this as a starting point to discuss the requirements we are establishing for TSNLP.

- Coverage
  - Breadth of coverage
    - Morphology
    - Syntax
    - Semantics
    - Extra-grammatical
  - Depth of coverage
    - Ill-formed data
    - Interaction and cooccurrence
- Source of test data
  - Lexicon
    - subject area
    - size

Note that 'breadth of coverage' does not include pragmatics, as one generally assumes that this is not testable using test suites. To some degree, this is also true for semantics, but as we will see below, there are some semantics phenomena that could be included. We shall have little to say below about morphology, except to note that morphological checking is always fundamental in NLP, and that test suites should, in theory, contain a complete morphology. The fact that hardly anybody has been interested in this area while building test suites (none of the test suites studied in WP1 treat either the full inflectional morphology nor consider derivational morphology in a systematic way) is probably a reflection of the fact that the first test suites that were developped were intended for systems dealing with English, where the morphological component was already satisfactory.

Coverage

We have already stated that the test data which we are planning to build should allow us to test three different types of applications. We assume that there is a certain degree of overlap between them, which allows general test data to be applicable to all three applications, but that there is also a degree of application specific test data required. The overlap seems to be such that if a grammar checker uses a parser then the test data for parsers is also applicable to grammar checkers, i.e. the test data needed for grammar checkers *includes* those for parsers. Grammar checkers, however, need additional test data, i.e. more ill-formed input to test whether they can achieve what they are designed for: reporting and correcting ungrammaticalities. A similar overlap/inclusion can be identified for CL checkers, as they tend to include a parser and need to report on violations of the particular controlled language *and* on ungrammaticalities. However, re-using test data for grammar checkers and parsers is problematic because the controlled language will put certain requirements on vocabulary, grammar and style, which force the evaluator to adapt the test data to the controlled language, or to start from scratch. Additional test data will still be needed, especially to test rules that specify restrictions on style and the lexicon (for which the other applications do not need to construct test data). Such application specific material will be discussed in section 4.

From this it follows that the grammatical and extra-grammatical phenomena to be tested are valid for all three applications, although some would have to be recognized as violating grammatical restrictions in CL checkers. We will then assume that, with regard to these phenomena, the TSNLP test data is required to be as *complete* as possible (i.e. within the limit of human capacity), but as time is limited TSNLP will construct data for a subset of syntactic and extra-grammatical phenomena.

An important aspect of test suite design, that has been noted elsewhere, is that the test suite must be built *systematically* and *progressively*, such that every new test item includes only *one* new phenomenon.

Ill-formed input

The way NLP systems should deal with ill-formed input depends on the system specifications. In most cases one would want a robust way to deal with this type of input, at the same time one would like the parser to be able to recognize that the processed input is ungrammatical. Either way one will want to include ill-formed input in the test suite. However, there are many ways to derive an ill-formed test item from a well-formed one, which makes it extremely difficult to be both systematic and complete.

Specification of ill-formed input is a difficult issue. Its importance and type depend largely on the task the system is meant to perform. It is useful to distinguish between two types of ill-formed input:

- Test items that violate normal grammatical feature restrictions (e.g. the wrong case marking on a noun, or the wrong number on a verb). We shall refer to this type of error as *competence* errors.

- Test items that contain errors that occur as a result of text manipulation, treatment of formatted structures, punctuation, etc.

We shall refer to this type of error as *performance* errors.

Ideally, a test suite should include examples of both *competence* and *performance* errors.

Another problem with ill-formed input is that, for each well-formed test item, there are potentially infinitely many ill-formed test items.

What is required then, is some selection procedure to decide which ill-formed test items are to be included in a test suite. In cases where the NLP system must deal with a particular type of corpus, and when it is possible to do so, criteria for selection of ill-formed test items should depend on the frequency of their appearance in that corpus. In cases where the NLP system must deal with unrestricted text, or when no reference corpus is available, the criteria for selecting ill-formed test items will have to be based on factors such as likelihood of occurrence, or linguistic plausibility. In any case, the derivation of ill-formed test items from well-formed test items must always be *systematic*.

Interaction and Cooccurrence

It is important to test the cooccurrence of phenomena as this allows you to test the rule interaction within the system. However, as with ill-formed input, the possibilities in combining phenomena are numerous. Again, some degree of systematicity and completeness should be maintained whenever possible, but it is advisable to establish an upper limit to the testing of interaction, to avoid a combinatorial explosion. Several criteria can be used to govern selection:

- frequency of cooccurrence of the phenomena in some specific text type
- centrality of the cooccurrence of the phenomena in the language

In either case, the requirement on systematicity imposes the need to define levels of complexity for the test items.

Vocabulary

It is generally accepted that test suites neither can nor should test for lexical coverage,[3] and as one generally does not want a test suite evaluation to become an exercise in adding lexical data, it is important to be specific about which criteria determine lexical selection. For example, one might wish to include examples of all the different subcategorisation frames for verbs in a given language, or choose lexical items for their lexical semantic properties (e.g. different types of aspectual verbs) although this may mean inclusion of lexical material that would rarely occur in any actual text. However, where the phenomenon that is being tested is not lexical in nature, it is advisable to keep the vocabulary as limited and stable as possible.

The choice of appropriate vocabulary for multilingual test suites (e.g. for testing an MT system for some language pair in both directions) will in addition have to take into account the specificities of each language and their existing or missing equivalences across languages. For example, whereas the two English and French verbs *to eat* and *manger* do not impose any restrictions of use, in the German language two different verbs will be used (*essen* and *fressen*) depending on the nature of the agent.

[3]For some NLP systems, however, testing vocabulary may be included in the test suite. This is certainly the case for CL Checkers which may contain very strict rules on vocabulary usage.

2.2.2 Requirements on Presentation of Test Data

The characteristics identified for WP1 which are relevant here are the following:

- Form of test data
- Annotation of test data
- Documentation

Form of test data

From a practical point of view, it would be most convenient to be able to have different types of test data forms. For some types of testing, it would be more convenient to test words (e.g. morphology), for others, parts of sentences or constituents (e.g. NPs for the testing of indexing applications), for others again, sentences, and even sequences of sentences (most notoriously for pronoun reference).

However, it may be the case that not all systems allow all these different types of inputs. The most accepted input form is of course the sentence, which is also the type of input used almost exclusively by most existing test suites. The design of the TSNLP database will, however, make it possible to store data in various forms and leave it to the user to extract the appropriate forms.

Annotations

Two types of annotation are envisaged in TSNLP:

- Annotations on the input
- Annotations on the output

Annotations on the input

The annotation scheme for describing the input should be comprehensive enough to cover all types of NLP applications, but the annotations themselves may vary depending on the type of NLP application: they probably should not be the same when a test item is meant for testing a grammar checker, a spelling checker or a machine translation system. On the other hand, it is likely that inputs still will share some common information even for different applications. This gives rise to the idea of a *core set of annotations*, with flexible extensions for the different NLP applications. Thus, the process of annotating will be realised in a modular way. As far as the annotation scheme itself is concerned, it is necessary to make use of a generally understandable vocabulary, i.e. use terminology that is not too technical or exclusively used by linguists.

The idea of using a core set of annotations with flexible extensions is also necessary to cater for the multi-application and multi-lingual character of the project. We want the annotations to be able to serve for phenomena which are included in the three languages. As there are differences between these languages, and as even similar phenomena may require different analyses, the annotations need to be flexible enough to take these facts into account.

Annotations on the output

This aspect represents in fact the most challenging and ambitious part of the TSNLP project. Obviously, the test input needs a proper annotation scheme, but until now, annotations on the expected output has not been considered absolutely necessary in building test data, even though such information is clearly desirable and helpful. Annotations indicating the expected output are much more difficult to devise than annotations on the input. They need to specify what the system is supposed to do. This strongly depends on the type of NLP application considered, since every application produces a different output: output from MT systems are typically translations, output from question-answering systems take the form of answers, output from general purpose NLP systems most often come in the form of tree or graph-shaped representations, and output from grammar checkers are normally error messages.

Moreover, even systems belonging to the same application type differ in their way of presenting the output: parsers use different types of graphic presentation and different sets of labels on the structures created, grammar checkers explain errors in different ways, MT systems produce different (although possibly correct) translations, etc.

If we only envisaged specification of expected output as a guidance to a human evaluator, the problem of providing useful information would already be considerable. When the number of test items increases, or if regularly repeated testing of system performance is desirable, the task of output checking becomes too laborious to be performed by humans. The ultimate goal is therefore to define ways of annotating for expected output that enable at least some degree of automatic comparison of expected and actual output.

Annotations allowing detailed automatic comparison may well be out of reach in a database of test items constructed with reusability and application multitude in mind, and be left as a major task of customization for any actual use of a test suite (see also section 5 below).

How should we for instance characterize the output of an MT application? MT seems indeed to be one of the hardest applications for which to specifiy what the output should look like. Listing all possible translations is one option for reusable MT test suites, but it is far from ideal. Listing specific required translations can be envisaged in the case of very application-specific purposes. Another possibility is that of more abstract and general non-lexical specifications of the output. On the other hand, it would be easier to specify the output of a spelling or a grammar checker, and for a parser, we can imagine a tree with labelled bracketing such as was proposed in the Penn TreeBank [5].

Another aspect of the characterization of the output concerns what is expected in case of ill-formed or ungrammatical input. We have to specify that it must be rejected or how it should be accepted, but of course it is difficult to state exactly where the border lies between acceptable ill-formed input and unacceptable ill-formed input.[4]

Work on these and other aspects of annotations has already started as part of the exercise in writing prototype test data which was part of the work performed in WP2.1 and on which the first version of a methodology given below is based. Elaboration and testing of an annotation scheme for test items is the sole task for WP2.2.

Note, however, that the availability of tools which compare the output of the application with

[4]For instance, proper names represent a real problem in this respect because although they are not contained in the system's dictionary, they cannot be considered as ungrammatical input.

the output of the test suites in order to assess test results is essential. When the number of test items increases, and regularly repeated testing of system performance is desirable, the task of output checking becomes too laborious to be performed by humans.

3 A Methodology for Test Suite Construction

In this section, we provide prospective test suite builders with a (first version of a) methodology for test suite construction. It follows the requirements that were set up in the preceding section and gives a step by step account of the construction process of a multi-purpose test suite which fulfills these requirements.

The methodology outlined here provides a framework that can easily be adapted for many types of usage by test suite builders, assuming that for some cases one might prefer a less elaborate process, and for others one might need to add system specific requirements or to augment the existing data base with more appropriate test data.

These tasks are probably best performed in the order given below, following the logical progression of first producing a description of the desired contents of the test suite and its presentation, then of establishing some prerequisites (addressing both some content and some presentation issues) and finally of constructing the test data in two steps, prototype and final version.

1. Establish what the test suite should contain:
   (a) Identify the linguistic phenomena that need to be included in the test data.
   (b) Decide to what extent ill-formed data is to be tested.
   (c) Decide to what extent co-occurrence and interaction of phenomena are to be tested.
2. Decide on the prerequisites for test suite construction:
   (a) Determine the domain of test data.
   (b) Determine vocabulary of test data.
   (c) Determine the form of test data.
3. Design prototype data.
4. Design a way to annotate the test data
5. Construct the test data. For each phenomenon identified earlier one should:
   (a) Construct simple test data.
   (b) Construct ill-formed test data.
   (c) Construct more complex test data (and ill-formed data).
   (d) Simultaneously, add annotations and refine them where necessary.
6. Document the test data

These tasks should be executed as sketched below.

3.1 Establish what the test suite should contain

a. Linguistic Phenomena

Task: Identify the morphological, syntactic, semantic and extra-grammatical phenomena which will be assessed. Ensure that these phenomena are:

- general enough (not dependent on a specific linguistic theory);
- well-defined; the general characteristics of each phenomenon and all aspects involved should be clear;
- fine-grained enough (i.e. not just a general overview of phenomena to be included, but an in-depth description of everything comprised under that phenomenon, to a satisfactory degree);
- relevant to the specific language (or, in multilingual test suite construction, relevant to a given language pair).

To perform this task, TSNLP uses the following sources of information:

- existing grammars (for grammar phenomena),
- texts (analysed by the TSNLP industrial participant, for grammatical and extra-grammatical phenomena),
- existing test suites (which serve as an extra checklist, not as a model).

An additional source can be added when building a test suite for a specific system: known system specifications may already provide the user with a clear enough description of the type of phenomena that the system should be able to handle.

As it is difficult to identify the phenomena to be included in a test suite, even with the help of existing grammars and analysed texts, one can use as a starting point the initial checklist given in Appendix A of the WP1 report [3], containing morphological, syntactic, semantic and extra-grammatical phenomena. The phenomena presented in this checklist are general, and when testing a specific system or application, the test suite writer may want to include more specific test data. This is especially the case for semantic phenomena, which are very limited in general test suites but may be highly relevant for the system to be tested.[5]

b. Ill-formed Input

Task: Decide to what extent ill-formed data is to be tested. The TSNLP approach to ill-formed input is to derive for each positive test item the following ill-formed test data:

- 1 or more test items containing an error which violates normal grammatical feature restrictions (relevant to all 3 applications dealt with by TSNLP);
- 1 test item containing an error resulting from text manipulation (relevant to all 3 applications, but especially for grammar checkers).

[5]As already noted, phenomena concerning pragmatics, discourse issues and world knowledge, are generally not assessed using the test suite method.

c. Co-occurrence and Interaction of Phenomena

Task: Decide to what extent co-occurrence and interaction of phenomena is to be tested.

It is necessary to include test data with interacting phenomena. The TSNLP data base will already contain some instances of testing for interaction. But the test suite builders will have to add some other cases. To avoid a combinatorial explosion, two main guidelines can be applied here:

1. Only test the most frequent interaction between phenomena (frequency may be based on the analysis of typical input texts).
2. In building a test suite for a particular system, use knowledge of the system's architecture. If it is known that the application of one rule is closely related to another rule, then this interaction should be tested.

3.2 Decide on the prerequisites for test suite construction

Before the actual test suite construction, test suite builders have to decide on several issues which directly influence the content of the test data. They have to decide on the domain, the vocabulary, and the form and organisation of the test data.

Task: Determine the domain.

Task: Determine the vocabulary to be used.

Vocabulary is related to the domain chosen in the previous task. To improve the reusability of the test data and to avoid interference of system behaviour which is particular to certain lexical items, the vocalbulary should fulfill two requirements:

– be limited in size

– not include highly specific terminology

We give some initial guidelines in the selection of the vocabulary:[6]

1. Include all lexical items from closed classes like determiners, demonstratives, pronouns, etc.
2. Verbs: as many as is needed to test all verb frames.
3. Nouns: at least two "human" nouns, two "concrete" nouns and an "abstract" noun. Some proper names (bothe people and places) are required. Must include nouns that take complements (e.g. deverbal nouns) and compounds.
4. Adjectives: both attributive and predicative adjectives. Must include adjectives that take complements.
5. Adverbs: both sentential and verbal adverbs.
6. Prepositions: all prepositions, or at least the subset of the most common ones.

[6]Note that lexicon testing must be seen as a separate goal, for which a test suite is not the best tool to use. One possible way to go about it is to derive a list of words from the input texts (base forms if the system includes a morphological component, otherwise also all inflected forms) and make the system check for these.

Task: Determine the form of the test data.

Options are:

- Constituents
- Sentences
- Sequences of sentences

Ideally, all three levels should be included in a test suite. Constituents are useful because this option allows the tester to concentrate on the relevant part of the sentence, e.g. when testing agreement within NPs or relative clauses. For sentence phenomena, the whole sentence is of course the required form of test item. Sequences of sentences are required to test phenomena such as intra-sentential pronoun reference.

3.3 Construct prototype test data

Task: Choose one or two of the identified phenomena and write prototype test data in order to identify possible problems.

3.4 Design a way to annotate the test data

Task: Specify how test data (every single item) should be annotated. The following issues need to be considered:

1. What information should be present in the annotation?
2. How should that information be specified?

The following types of information can be addressed by an annotation scheme:

- The phenomenon being tested;
- The grammatical and/or non-grammatical elements in the test item;
- Whether the test item is well-formed;
- What is wrong when a test item is ill-formed;
- The required system output.

As discussed in section 2.2.2, to provide information concerning the required system output is difficult to achieve with a system independent and theory independent representation, since system output is very much dependent on the design of the system. The test suite builder can choose one of the following options:

1. Specify system specific output.

2. Leave the annotation for required system output open, leaving it up to any individual user to fill in this slot.

3. Try and specify some system independent representation.

Annotations must be detailed so as to ensure optimal usefulness, without drowning the test-user in a mass of irrelevant information. The TSNLP methodology for the annotation of test data and the full annotation scheme for TSNLP will be given in our next deliverable.

3.5 Construct the Test Data

The construction of test data requires the test suite builder to perform different tasks at more or less the same time. This is because test data construction will take up one phenomenon at the time (phenomena identified under step 1 of this methodology), and for each phenomenon we will have to 1) construct simple test data, 2) construct ill-formed test data, 3) construct more complex test data, and 4) add annotations and refine them (during 1-3).

a. Construct simple Test Data

   Task: Taking one phenomenon at the time, construct a test item which contains that phenomenon and as few other phenomena as possible. Avoiding the inclusion of any other phenomenon is impossible, but one should certainly avoid the more complex constructions, unless this is intentionally tested for.

   If different variants for one phenomenon are possible, construct other test items, where each one contains one of these variants. For example, when testing copular verbs, one would like to include items in which the verb takes an NP (e.g. *John is a lawyer*), an AP (*John is old* and a PP (*John is out of money*).

b. Construct Ill-formed Test Data

   Task: For every well-formed test item, construct ill-formed data as planned above under (1b), i.e. change the instanciation of one of the relevant characteristics of the phenomenon being tested. This will provide the first kind of ill-formed counterpart. For the second one, introduce an error which could result from text manipulation, and which would affect the treatment of the phenomenon.

   When constructing ill-formed data, ensure the following:

   1. For each phenomenon, only derive ill-formed input that is relevant to that particular phenomenon.
   2. For each phenomenon, be complete in providing ill-formed input.

   This is probably best illustrated by examples. Point 1 states that if we include *He eats a red apple* as a test item for subject verb agreement, then we do not include *He eats a red apples* as its ungrammatical counterpart, but rather *He eat a red apple*. Point 2 indicates that, for example, when testing the verb frames of verbs which take tensed subclauses as complements, we can systematically test the ungrammatical counterparts of *The lawyer agrees that the company is large* as for instance: ∗*The lawyer agrees the company to be large* and ∗*The lawyer agrees the company being large*.

Of course, how to determine the completeness of ill-formed input depends highly on the phenomenon that is being tested.

c. Depth and Interaction

Task: Construct more complicated test data and allow for the co-occurrence of phenomena within one test item. Determine the level to which interaction of phenomena is to be tested.

Both constructing more complex test data and treating interaction of phenomena will involve the buiding of "traps", i.e. test data containing phenomena which are complex to process, so that the level of sophistication of the system's rule component can be tested.

There are two kinds of traps, positive and negative. The former could induce a system to find an error when there is none in the input, while finding the error in the latter would require some sophistication.

Examples:

1. The cake for the children is on the table.
   (positive trap for subject-verb agreement)
2. The cookies for dessert is on the table.
   (negative trap for subject-verb agreement)

We can add traps to each test set, or have separate test sets where all test elements are traps. Besides making the organization of the test suite clearer, the latter option allows us to assign degrees of difficulty to each test set. If the user only wants to assess the performance of a system in simple cases, without assessing how sophisticated it is, then the test elements chosen can be the straightforward simple test sentences. The user can then choose to use the more sophisticated test sets consisting of the positive and negative traps.

d. Add annotations

Task: Annotate the test data (if necessary, refine the annotations).

3.6 Documentation

The most important task for the documentation of a properly annotated test suite is to provide outsiders with information regarding at least the following:

- the purpose(s) for which the test suite was constructed,
- the reason why it was constructed in this particular way,
- which languages are involved,
- where possible, some indication of which phenomena are included/excluded,
- some specification of the vocabulary (size, domain),
- if available, an assessment of the adequacy of the test suite (strong and weak points) with respect to the task(s) for which it was designed.

For TSNLP in particular, two types of documentation will be provided:

- a general description of the design principles and instructions for use (extraction, manipulation, extension of the test data);
- on-line documentation of high level annotations (definition of phenomenon, explanation of individual annotations used, etc.).

4 Application-specific Requirements

In addition to the 'application-neutral' methodology described in the previous section we here present requirements for the applications TSNLP is concentrating on, namely:

- Parsers
- Grammar Checkers
- Controlled Language Checkers

The requirements discussed in this section are not to be considered to replace the general TSNLP methodology, but as extra requirements which a test suite builder should take into account when the application which is going to be tested is already known.

For each application, the guidelines have the following structure:

1. The scope of test suites
2. Requirements on test data: content
3. Requirements on test data: presentation

4.1 Test Suite Requirements for Parsers

One of the particularities of parsers is that it is a tool common to most of the NL applications. In fact, it is usually one part of a whole system and very seldom an end-application in itself. The goal of a parser is to provide an intermediate representation of the input resulting from an analysis. The intermediate representation is then processed in order to provide the final output corresponding to the whole system's final objective : translation, indexing, controlling, etc.

4.1.1 Scope of test suites

Because of these "intermediary" characteristics of parsers, an important point we have to take into account seems to be the output of the parser which is quite always an intermediate output. Depending on the parsers, there exists a large variety of output types : trees, bracketed labels, featured dependency structures, etc. Each parser may present a specific way of labelling analysis results, thus making the interpretation of the results quite complex.

Moreover, as parsers are usually parts of whole systems, it may happen that the parsing results cannot be accessed at all (which is the case for some indexing or translation applications, for example). In this situation, there is no way to test the parser by itself; identifying the origin of a failure (parser or other modules of the whole system) becomes impossible. Thus, on this account, it appears that test suites for testing a parser can of course only be useful when the access to the parsing output is possible.

Due to this usual "intermediate" situation of the parser, the performances that can be tested strongly depend on the objective of the whole application: the parser of an indexing application will probably not be as sophisticated as the analysis module of an automatic translation application, because the need for a fine-grained analysis is not the same. Moreover, the need for robustness may differ from one type of end-application to another, thus influencing ill-formed input testing. The use of test suites for evaluating parsers seems to enable the assessment of these two aspects: degree of robustness of the parser and degree of refinement of the parsing (richness of information on the output).

An important point concerns the modularity of the parser, i.e. the number of intermediate outputs one may obtain during the parsing process. Some parsers first perform a morphological analysis, producing a morphologically annotated output, then they perform a syntagmatic analysis, for example identifying the major components of a sentence thus producing a second type of output, and finally a syntactic analysis producing a third output like a tree for example (a semantic analysis may also be added producing another type of output like a semantic representation of the input). That means that the information obtained on the input during the parsing process is distributed over several types of outputs (morphological, syntactic, etc.). It seems to us that this is likely to influence very much the way test suites can be annotated in terms of output and expected results.

There are several ways of envisaging the evaluation of parsers: one could evaluate the adequacy of parsing with regard to the further processes that will be applied on the parsed output. This means that you evaluate the adequacy of the parsing with regard to the final application needs; for example, does this parsing output allow the identification of style problems (for a style checker)? Or one may want to evaluate the performance of the parser by itself in order to check the adequate processing of the input and to identify the problems, thus performing diagnostic evaluation. Test suites seem to be quite appropriate to help and perform this last type of evaluation.

The major task of a parser being to assign a representation to the input, test suites are supposed to test the correctness of this representation independently of the formalism chosen for the output representation. This means that a particular effort should be made in the area of annotation on output.

We assume that a good knowledge of the possible outputs of parsers is necessary to build and above all annotate the test suites. Also, it appears that due to the complexity of parsing outputs, only a user with computational linguistics background could handle and interpret the test suites results.

As far as ill-formed input is concerned, the use of test suites will help evaluate the degree of robustness of the parser: how does it react to an ill-formed input? does it stop? does it give a representation of the ill-formed input, nevertheless signalling a problem? or does it give an erroneous representation.

4.1.2 Requirements on Test Suite Design: Test Data

Coverage

Defining the coverage of test suites for parsers implies that all the different tasks a parser may perform are covered, including extra-grammatical tasks such as segmentation of the input in analysis units. We assume that a parser is likely to minimally perform the following tasks:

- identification of the basic analysis units (segmentation)
- morphological analysis (assign labels and features)
- syntactic analysis (assign structure)

The identification of basic analysis units is the first task performed by any parser. Even if this task is a non-linguistic task, test suites can represent a means to assess the segmentation performances. Testing segmentation by means of test suites implies that these do not consist only of isolated sentences but also of sets of sentences or sets of sentence fragments.

The assignment of morphological labels and features of course strongly depends on the actual state of the data contained in the dictionary used by the parser. As testing the lexical coverage of a parser can never be complete, it is necessary to allow lexical changes in test suites so that only the morphological mechanisms are tested and not the lexicon content.

To test the adequate assignment of syntactic representation to the input, it is clear that the designing of test suites should be very systematic, going from the simplest to the most complex (including phenomena interaction) in order to achieve both breadth and in-depth coverage.

Morphological and syntactic ambiguities are two crucial points to be considered when designing test suites. The interest lies in the observation of the behaviour of the parser when dealing with ambiguities: does it produce all the possible interpretations? does it stop? does it choose only one possibility? and in this case, which one and why?

Ill-formed Input

Inclusion of ill-formed input in test suites is necessary in order to check the reaction of a parser when confronted with ill-formed input. As far as parsers are concerned, ill-formed input corresponds to input which does not conform to usual grammatical rules of a language. Other kinds of ill-formed input do exist, e.g. input not conforming to the restrictions of a controlled language, but the testing of this point will be performed in the framework of a particular application-specific use of test suites (see section 4.3). When building test suites for ill-formed input, it is very interesting to have at least one ill-formed input for each tested phenomenon, and more than one may be needed in case of phenomena interaction.

4.1.3 Requirements on Test Suite Design: Presentation

The issue of test suite presentation is considered as a very crucial point for the whole process of test suite design. A distinction can be made between several kinds of annotations on test suites :

- Informative annotations on the test suite (general information on the structure of the test data set) which we could call "meta-annotations";
- Descriptive annotations on the input (annotations for each test item)
- Annotation on the output (annotations on expected results).

The informative annotations on the test data set should be as clear as possible, indicating all the information necessary to ensure a good re-usability of the data.

The annotations on the input will clearly state what phenomenon each test item is supposed to test in a way such that it is understandable by any user. Each phenomenon should imply the building of several test-items (maybe two positive items, one negative, and if possible, one positive trap). The test data for ill-formed input should be clearly identifiable, as well as the error it is supposed to test.

As we said before, it seems that a particular attention should be paid to annotations on output because of the very complexity of possible output types of parsers. The annotations on output for parsers can certainly not reflect exactly the output produced by a parser, the only possibility is thus to determine a "standard" annotation language that would allow the comparison with any form of parser output. In this sense, the usabilty study of the Penn Treebank seems to be an interesting issue we have to investigate.

4.2 Test Suite Requirements for Grammar Checkers

What distinguishes grammar checkers from most other NLP systems (spelling checkers and controlled language checkers excepted) is the fact that they focus on detecting and correcting (or proposing corrections for) errors in unrevised text. The EAGLES Writers' Aids group gives the following definition of the role of a grammmar checker in the text production process in their interim report (EAGLES 1994, p.34):

- A **non-proofed text** prepared by the **writer** is revised by the **end-user** using **advice** produced by the **system** to produce a **proofed text**.

This is of course a trivial observation but it clearly affects the design and use of test suites in several ways, especially concerning the definition of coverage. The coverage of a grammar checker is not only to be stated in terms of its positive coverage (i.e. the text parts where no errors should be flagged) but also in terms of the types of errors the system is able to detect and propose corrections for. It is also to be noted that in the particular case of grammar checkers, the **writer** and the **end-user** are most often the same person.

We now look in turn at each of the application parameters relevant for Grammar Checkers, to determine their impact on the design of test suites for GCs.

4.2.1 Scope of Test Suites

System Tasks

We can identify the following tasks for grammar checkers (taken from [9]):

1. Assign correct (maybe partial) structure (through parsing or pattern matching) to the input string;
2. Identify errors against the grammar;
3. Report errors against the grammar;
4. Make suggestions for corrections.

According to Thurmair (1990, p.365) grammar checking tries to find ill-formedness, i.e. input which by definition is considered to contain a mistake and MUST be corrected.[7]

A grammar checker sometimes also tries to check for style. It then considers levels of acceptability according to parameters set by the user (e.g. GRAMMATIK has three such levels). In that case, the system does not necessarily encounter a failure when processing the input data for which it must report a mistake, and it should not report such style errors in the same way as grammatical errors. We must thus be aware of the difference between *grammar checker* and *style checker* (cf. also Thurmair, pp.365–367), and establish different kinds of test items for these two kinds of checking if we want test both of them.

Moreover, with a grammar checker (and that is also the case with a controlled language checker), we do not necessarily want to know whether the system is able to perform a successful analysis, but rather whether the system can recognize a particular error (see (EAGLES 1994)). In addition, for any adequacy evaluation of a gramar checker, what people want to know is whether the corrections and the advice given to the user are correct, useful and user-friendly. While test suites may help the tester assess the validity of the corrections proposed, they will not be able to contribute to the assessment of the advice given.

[7] In this sense of what a grammar checker is and should do, Thurmair correctly states that "Every ungrammatical sentence should cause a parse failure." (p.365). However, for our purposes, Thurmair's statement is besides the point: whether the system's parser fails or not is an internal state of the system which is function of the algorithm employed, but of no relevance for its behavior.

System Types

Grammar checkers currently on the market fall into two classes:

- pattern matching systems;
- parser based systems.

Besides the fact that grammar checkers of the second type are more effective (EAGLES 1994, p.32), this difference is clearly important for the design of test suites intended for developers testing their own system. For user-oriented test suites, it is less obvious that this distinction among types of systems will make a difference in writing test items, since in both cases, it is necessary to check systematically for correct recognition of syntactic patterns.

User type

At present, current grammar checkers also fall into two classes along another dimension:

- grammar checkers intended for native speakers, of varying linguistic proficiency level and professional training;
- grammar checkers intended for second language users or learners.

Designing test suites solely for the second type of grammar checkers would require knowledge of the kinds of interference from the first language of the intended user. This knowledge would be necessary both to tailor the test items to the types of mistakes due to this interference and to provide useful and informative corrections, but at present, such knowledge is not available. However, since we do not intend in TSNLP to provide advice for correction, but only to test for every possible type of mistake, the distinction between GCs for native speakers and for second-language learners will not be be pursued any further.

What Test Suites can do in the case of Grammar Checkers

In looking at this type of application, we actually reach one of the limitations of the use of test suites. It cannot be stressed enough that, in the general case, the result of applying a test suite must never be taken as the result of an adequacy evaluation of a system, even if for some applications it can sometimes be taken more or less straightforwardly to indicate the level of performance of a system and to give some idea of how “good” it is. If test suites cannot be used for general adequacy testing purposes, they can even less constitute the unique tool involved in assessing the performance and usefulness of a system which by definition interacts with the user at a meta-linguistic level, and whose usability is directly related to its user-friendliness.

In the case of grammar checkers, a large part of the system's performance lies outside the domain which may be covered by test suites. Other tools and techniques must be used to check these other aspects, and the result of applying a test suite can only be taken as indicative of the linguistic adequacy and coverage of the system.

Nevertheless, test suites can be used as tools to obtain a subset of the required information for an evaluation of the system. Test suites for grammar checkers can provide information on the coverage of errors correctly or incorrectly identified, i.e. they can be used to obtain measures of *recall* and *precision*.

Test suites can also be used as a systematic way of investigating how a grammar checker explains errors and proposes corrections for certain types of errors, but judging the adequacy of these explanations for different types of users falls beyond the scope of test suite design and use.

4.2.2 Requirements on test suite design: Test Data

Coverage

As with all the other types of NLP applications to be considered, in defining test suites for grammar checkers, we have to face the problem of completeness.

First, we address the question of completeness of testing. It is clear that in testing a grammar checker, all textual phenomena must have been eliminated from the input text before testing. We thus assume that such phenomena are outside the scope of the coverage of test suites. In particular, we assume that spelling mistakes are "pre-handled" by other tools. However, since a spelling checker is often integrated with a grammar checker, it is not always clear to the user how to make the distinction.

Nevertheless, such a distinction between spelling mistakes and grammatical errors must be maintained. Even though grammar checkers may check for both spelling and grammar errors, the use of test suites is better reserved for the latter type of errors. The former may result from many factors for which it would be very difficult to design test suites, e.g. typing mistakes (metathesis, adjacent keys, doubling, etc.), errors of attention, text processing problems (cut and paste). These mistakes fall under the category "textual phenomena". Using test suites to discover whether a system can recognize spelling mistakes would not be efficient, and building test suites for that purpose would be a waste of time. Other means of testing for this type of problems are more appropriate. Still there are some textual mistakes for which we can easily write simple test sentences (e.g. *the the* or *form/from*, which a simple spelling checker would not detect.

Second, there is the issue of lexical coverage. Like spelling checkers, grammar checkers are by definition expected to be able to deal with the widest possible range of lexical items. This is another area where test suites are not the appropriate tool to use: as is the case with the testing of any other kind of NLP system, using test suites cannot determine whether the lexical coverage of a system is complete. We must stress again that a complete adequacy evaluation of the system under test cannot be accomplished with test suites as the only tool. Here, the possibility of modifying the lexicon, or at least of adding lexical entries, is of crucial importance in the design of an NLP system, but this is a feature which is clearly beyond the scope of test suites.[8]

However, test suites are very well suited for testing the behavior of a system in the face of ill-formed input, which is one defining aspect of this type of application. The methodology for deriving ill-formed test items from well-formed ones has been one area of work during Task 2.1,

[8] Independently of testing the lexical coverage of a particular system, but related to the issue of modifying a lexicon, we must add here that the possibility of performing lexical replacement in a "Virtual Test Suite", so that a particular instanciation may be tailored to the domain of a given application, is one of the major features of the system we envisage for TSNLP.

and will be pursued during Task 2.2.

4.2.3 Requirements on test suite design: Presentation

Even more than with other NLP applications, where we might be interested in other aspects of processing, test suites for grammar checking must include negative test items, since we are interested in the behavior of the system when faced with ill-formed input.

We thus propose that a test item is actually a "test set", which comprises at least the following three elements:[9]

- 1 positive input (1 sentence WITHOUT error)
- 2 negative inputs (2 examples WITH an error)

In addition, we also want to have "traps", i.e. test items where a simple-minded pattern-matching algorithm would fail.

Annotations

As for all NLP applications for which a test suite will be used as an help in testing, in the case of grammar checkers, it is useful to have annotations giving information not only on the input part of a test item (i.e. describing what the test item is and what it tests for), but also on the output part (i.e. indicating what the behavior of the system should be).

Annotations on the input and output parts of a test item will of course always use consistent terminology in the TSNLP test suites. It is difficult to create annotations which are both detailed and specific enough for testing purposes and still remain user-friendly. Therefore, the application specific annotations on the output part must be adapted to the users, at least to some extent.

In the case of grammar checkers, we may take advantage of the fact that the people who have designed and produced grammar checkers also had to produce some user-friendly output. Since they know exactly what users want and have been able to create annotations which are understandable by them, we should take some inspiration from the output of commercial grammar checkers in designing our set of linguistic annotations. Of course, these annotations may also be used in the test items for other applications.

The annotations on the input and output parts of the test elements follow the same presentation and are related to each other. Annotations on the various elements of a test set are also related to each other in the way described below.

Input

The input part of a test suite item for Grammar Checkers should consist of:

[9] The idea of using paired positive/negative test elements is already present in the EAGLES *Writers' Aids Interim Report* (p.44-45), and the idea of having TWO negative inputs comes from the *PC Magazine* report on testing grammar checkers (p.158).

A number of such test sets must be created for each phenomenon being tested. The choice of the positive test elements for these test sets and the choice of which error to test in each test set is not discussed here, as it is part of the general methodology.

• 1 positive test element

This is to be taken as a "standard" example of a correct instanciation of the phenomenon under consideration. The annotations on this test element only show the information which is pertinent for this phenomenon. Other information is derivable automatically from the information contained in the lexical entries for that input.

• 2 negative test elements

These are derived in a systematic way from the positive test element, and the annotations given for each of them indicate how this test element differs from the positive one. All other information is derived automatically either from the positive test element or from the lexical entries.

• the name of the error to be found

We will not deal with the problem of actually naming the errors in this WP (this is to be done in WP3), but it is clear that this name is part of the annotations on the input.

The question of the classification of errors is related to the question of the choice of linguistic annotations on the input and the output parts of the test items. We should of course make sure our own classification is compatible with the list given by (Douglas 1990, pp.41-42), and those of (EAGLES 1994, pp.36-37) or (Douglas & Pedersen 1994, p.9). We can use as a starting point the list of errors provided by GRAMMATIK: grammar errors, typographical errors and style errors. As was said above, a test suite for grammar checkers should not be concerned with typographical errors, and these will not be included in the test suite.

Output

The output part of a test item in a test suite indicates how the system under test is expected to perform, when given the input. In the case of a grammar checker, the output of the system consists in:

- the recognition of the exact error in the test input (or no error flagging, if the input does not contain an error);
- suggested correction(s) to the user;
- possibly, some advice generalizing over the type of grammatical error contained in the input.

Since it is not the purpose of TSNLP to test grammar checkers (or any other application) for their adequacy, we will not attempt here to give indications on how to check or test the corrections and the advice given by the system. However, we must and can propose some correct sentences as suggested output corrections. These could later be used in automatic testing of the actual corrections suggested by the system, and otherwise serve as a benchmark for manual comparison.

The output part of the test suite items for grammar checkers thus does not correspond to the actual output of the system, but should consist of:

• the name of the error

As the test of a commercial grammar checker reported in the Appendix of (EAGLES 1994) shows, system testers can be confronted with unexpected errors. For instance, when the tester might expect that the system would complain about lack of agreement in *many dog*, it instead reported "*DOG as a verb*". This is of course a problem related to the interpretation of results, and TSNLP will not deal with it as such. However, as this problem is extremely important for the people who produce grammar checkers, it must be kept in mind while designing test suites for grammar checkers especially in the "output" part of the test set.

• the correct version of the 2 negative test elements

The annotations in the output part of the test elements only give information on the corrected parts of the ill-formed input and suggest replacements for the incorrect forms.

4.3 Test Suite Requirements for Controlled Language Checkers

CL checkers are generally designed on the basis of a particular controlled language. As there are different controlled languages, the rules implemented in the CL checker will differ from system to system. As a consequence, the test material required will also differ from system to system. It is our task to determine to what extent we can provide general test material for CL checkers, and which part needs to be system specific. We will come back to this later.

In order to assess CL checkers it is important to establish which tasks the system performs. We can identify the following:

1. Assign correct (partial) parse to the input string.
2. Identify errors against the controlled language. We can subdivide this as follows:
   - Errors against vocabulary restrictions
   - Errors against grammar restrictions
   - Errors against style restrictions
3. Identify errors against grammar (it is not clear whether all CL checkers do this, but e.g. Boeing does.)
4. Making suggestions for corrections (this is not done by all systems.)

Most of the tasks performed by CL checkers during processing are relevant to all CL checking, but since this is still very much a development area not all tasks are performed by all systems.

Task 1 and 3 have already been discussed in the previous sections on requirements for parsers and grammar checkers. Here we will concentrate on error identification and correction.

4.3.1 The Scope of Test Suites

We can identify quite an extensive number of system specific issues which are very relevant in the assessment of a CL checker. As it stands, controlled language checkers are not aimed at dealing with all types of inputs, users, etc., so in order for test material to be complete one should include material which takes into account:

1. Type of input text, e.g. manual;

2. The particular controlled language used;
3. The typical user of the system (e.g. whether is it used by many non-native speakers).

As these are all system specific matters this means a lot of work is required to complement an application specific (geared towards usage for CL checkers) test suite with system specific material.

Point 2 of the above list is a considerable complication. The problem resides in the fact that CL checkers are dependent on a theory, or better, on the underlying CL. As there are several different CLs in existence, it is difficult to draw up a test suite which can be general enough to be of use for CL checkers which are based on different CLs. The differences do not only affect the grammar rules (and related test items), but also vocabulary and style. Therefore, we should not attempt to draw up a test suite for CL checkers in general. Instead, our goal should be to build test suites for those checkers which are based on the same CL. This means a serious limitation to our aims of reusability and generality, which is indeed a major disadvantage.

4.3.2 Requirements on Test Suite design: Test Data

Identifying CL Errors

When evaluating the system's capability to identify errors we have to assess the flagging of errors by so-called error critiques. What needs to be established during assessment is:

- whether any critiques are incorrectly reported (these are called *spurious critiques*);
- whether the system failed to identify errors (*missed critiques*).[10]

Another aspect that needs to be taken into account are those error messages which indicate something is wrong with a sentence, but cannot indicate what exactly the problem is. We will call these *vague critiques*.

The errors against CL restrictions can be subdivided according to the three different types of restrictions which are generally imposed by controlled languages (see e.g. [1]):

1. Restrictions on vocabulary
2. Restrictions on syntax
3. Restrictions on style

We will look at these restrictions one by one.

Vocabulary Vocabulary restrictions vary from CL to CL, but it is quite common for them to contain very strict vocabulary rules. Some CLs depend largely on their vocabulary restrictions. Therefore the assessment of this task of the CL checker is certainly important.

[10] Both these terms were taken from [9].

The question is whether we can employ test suites for this. If we want to assess whether a CL checker correctly identifies all errors against the restrictions on vocabulary ,we would need to run quite an extensive vocabulary test, which falls outside the usual scope of the test suite method. In order to determine to what extent test suites can be used to detect violations on vocabulary restrictions we should divide lexical rules into 4 classes (the examples which illustrate classes 2 and 3 are taken from [2]):

1. Lexical rules that specify which words are disallowed (or allowed; but in that case they will probably belong to class 3).
2. Lexical rules that specify which words are disallowed and which suggest which word to use instead. For example, a rule could suggest to the writer to use the verbs *point* or *try* instead of the verb *aim*, depending on its meaning.
3. Lexical rules which include a specification indicating under which circumstances a certain word is allowed, depending on the *category* or *meaning* of the word. Examples of this is *aid*, which is allowed as a noun but not as a verb, and *follow*, which is only allowed if it means 'to come after'.
4. Lexical rules that specify under which circumstances a certain word is allowed, depending on the (syntactic or semantic) *context* of the word. The following exampe of this has been taken from [1]:

   Use only **because**, *never* **since** *in a subclause of reason:*
   - Since a DBCS manages the System 12 database, physical storage is transparent to the users
   + Because a DBCS manages the System 12 database, physical storage is transparent to the users

   Here, the decision of whether the word has been used in the right way depends on the context.

Now the suitability of test suites for vocabulary assessment differs for the different types of lexical rules. Rules in class 4 are often regarded as grammar rules, and will be therefore be included under that heading. Class 3 can be tested using the test suite method, but does require quite an extensive list of test items; the words that fall under this heading, and their disallowed counterparts, might be numerous, but it will still be a limited amount. Note that for those words whose usage depends on the meaning of the word, the number of alternative meanings might be considerable and one may want to restrict the number of possible test items for this class.

We would like to consider class 1 and 2 as one class for the time being, as the distinction is only relevant for the task of making suggestions for corrections, which is discussed below under "Error Correction". As the class of disallowed words is unlimited the system's performance in this area cannot be assessed using the test suite method. As a consequence, additional test material will be necessary. Using real texts to assess this issue seems to be the obvious approach.

Syntax To identify errors against the restrictions on syntax the test suite method can usefully be applied. Four types of test items need to be included:

1. Test items which are grammatically correct and which comply with the CL rules (also with regard to vocabulary and style).

2. Test items which are grammatically correct but which violate the CL grammar rules.
3. Test items which are grammatically incorrect but which do not violate the CL rules (this in fact is the task of the grammar checker).
4. Test items which are grammatically incorrect and which violate the CL grammar rules.

For point 3, it will be clear that the test items used for testing grammar checkers can be applied. For point 1, an existing test suite could be used as a basis, but to a certain degree, depending on the checker's underlying CL, amendments will be necessary in order to make the test items consistent with that CL. As for the other two points, special test items will need to be constructed. An existing "general" test suite and the CL guidelines are useful to achieve some degree of completeness in coverage.

Note that not all syntactic restrictions are easy to implement. Recall the vocabulary example given above to illustrate this. The example requires the CL checker to distinguish between subclauses of reason and time, a non-trivial task which can be problematic even for a human reader. Constructing test items for such phenomena however is relatively easy.

> *Use only* **because**, *never* **since** *in a subclause of reason:*
>
> *- Since a DBCS manages the System 12 database, physical storage is transparent to the users*
>
> *+ Because a DBCS manages the System 12 database, physical storage is transparent to the users*

Style Although CL checkers will attempt to identify errors in style there are some major problems with this subtask. This will be clear if we give some examples of style rules, taken from [2].

Punctuation, e.g.:

> *If you start an instruction with a descriptive statement (dependent clause), you must separate that statement from the rest of the instruction with a comma.*

Sentence structure (sentence length, contents, etc), e.g.:

> *Write only one instruction per sentence.*

Paragraph structure (e.g. paragraph length, contents), e.g.:

> *Each paragraph must have only one topic.*
> *Always start the paragraph with the topic sentence.*

Layout (e.g. headings), e.g.:

> *If you want to show that some steps are closely related, use the tabular layout (vertical layout) for the text.*

Note that all rules given above are of a descriptive nature. This is of course not true for all style rules (some are more concrete in the sense that they just require counting words or sentences), but it is a very common type of rule. Because of their descriptive nature, they are difficult to implement. Drawing up test data for such rules however should be fairly easy.

Another problem here is the fact that quite a few rules require large pieces of texts in order to be tested properly, e.g. (again from [1]):

> *Do not use one-sentence paragraphs more than once in every 10 paragraphs.*

It depends on the number and importance of this type of rule, but in most such cases it is not desirable to include such large pieces of test material in a test suite. Corpus testing would be needed for this.

It will be clear from the above that there are many problems related to the implementation of style rules. Although some of them may be fairly straightforward to implement and test (e.g. sentence length restrictions) others will be problematic. Depending on the type of style rules which a CL includes, it is probably best to include only part of them in the test suite to be used.

Error Correction We can distinguish between three types of CL checkers:

1. Checkers which flag mistakes but do not make any suggestions for error correction.
2. Checkers which flag mistakes and make suggestions for error correction.
3. Checkers which flag mistakes, make suggestions for error correction and actually perform some amount of automatic correction in case error detection is fairly straightforward.

Most CL checkers will attempt to make suggestions for error correction rather than just flag the mistakes, but the quality of these suggestions often leaves much to be desired. Though a certain number of them may be right, it is not uncommon for systems to come up with too vague or plainly wrong suggestions.

Some more advanced checkers, like the SECC system which is currently being developed, attempt a degree of automatic error correction. There are pro's and con's for error correction, as was mentioned in [8]:

> *Not all CL checkers do error correction, as this seems to have several drawbacks. One of the criticisms is that the user may feel that the system is too rigid (they prefer to do their corrections themselves). Another problem is the fact that it is not always clear how to correct syntax errors. One could also comment that users do not learn how to write in a controlled language correctly if the system automatically corrects all mistakes. However, a CL checker for a strict controlled language can throw out quite a few mistakes per sentence, and correcting the most obvious ones (especially lexical mistakes) will help the user in dealing with the other mistakes. Correcting vocabulary errors automatically is an obvious step as it is fairly straightforward for the system to match unallowed words to their controlled language synonyms.*

If we distinguish between lexical, syntactic and stylistic error correction it is clear that the first type can be achieved quite easily. The second and third type are much more complicated and automatic correction is in many cases not possible. For some of these rules it might be possible to suggest a correction, but more research results are required in order to predict what one could reasonably expect from a CL checker which performs correction.

If we want to use test suites for those CL checkers that make suggestions for error correction or even do some automatic correction, it seems desirable to reuse the test material which we used to test the error identification task of the CL- and grammar checker.

4.3.3 Overview of required Test Data

To start with, we can identify several requirements for test suites for CL checkers. Firstly, as we will see below, the number of possible interactions of phenomena in test items is much larger than for a test suite intended for use with a parser or a grammar checker, as rules on vocabulary, style and grammar interact with the usual grammar phenomena. The testing of interaction is necessary, but restrictions will need to be imposed.[11] Secondly, a test suite for CL checkers will require test items that allow for testing sentence length and style rules on paragraphs. Thirdly, testing of ill-formed input (either grammatically or according to the CL) and their correct counterparts is of utmost importance. This is taken account in the description of test material given below.

As for the different tasks within the CL checker, the following list describes the type of test material required. Note that examples are based on [2]:

Parsing The parser, whether complex or basic, can be tested as an individual component, in which case the evaluator can make use of a general test suite for parsers as described in section 4.1. If the parser is tested in combination with the other system tasks, the test material as described below (CL and grammar checking specific) is required.

CL Vocabulary: The test suite should contain items which will allow us to test the following vocabulary rules

1. Lexical rules that specify which words are disallowed and which suggest which word to use instead. Possible test sentences for this type of rules are:
   a. Adhere to the safety instructions.
   b. Obey the safety instructions.
   c. Aim the open end of the line away from personnel.
   d. Point the open end of the line away from persons.

   We would like the system to flag the word *adhere* in sentence (a) and suggest its replacement by *bond*, *obey* or *keep*. Automatic correction is not desirable in this case as 3 alternatives are suggested. In sentence (c), two disallowed words occur: *aim* and *personnel*. For *aim*, the alternatives *try* and *point* should be suggested, *personnel* should be replaced by *persons*. The correct counterparts of (a) and (c) are (b) and (d), which should be passed by the system without flagging or correction.
2. Lexical rules which include a specification indicating under which circumstances a certain word is allowed, depending on the *category* or *meaning* of the word. Consider the following possible test items:

[11] This is why the interaction between vocabulary rules, CL grammar rules and CL style rules is not really discussed here.

a. Apply sand to the planks to aid tire traction.
b. Get medical aid immediately.
c. Follow the indicator on the screen.
d. This test follows the installation of landing gear.

In this case we would like the system to flag (and correct) the occurrences of *aid* in sentence (a) and *follow* in sentence (c), but not the ones in sentence (b) and (d).

The suite should not attempt to test all words disallowed by the checker, as there is an indefinite number of them and this falls outside the scope of the test suite technique. It should also not include any company specific terminology. If such terminology tests are desired, then they should be constructed seperately.

CL Grammar Rules: This should contain the following types of test items:

1. Test items which are grammatically correct and which comply with the CL rules (also with regard to vocabulary and style). For this an adapted parser test suite can be used.
2. Test items which are grammatically correct but which violate the CL grammar rules. As this primarily aims at finding CL violations, the important requirement is to include one (or more if interaction is tested) CL violation in each test item. If each violation is tested in combination with different grammatical phenomena, a combinatorial explosion would take place. The desired approach is probably to test all violations systematically while ensuring some variation in grammatical phenomena. Some possible examples of test items are:

   a. This test follows the installation of aircraft landing gear.
   b. The controls can be adjusted.

   Sentence (a) contains a disallowed complex noun cluster. As a compound may only consist of two nouns, this should be rephrased as *the landing gear of aircraft*. In sentence (b) the rule that past participles may not be preceeded by an auxiliary is violated; the alternative here is *You can adjust the controls*.[12].
3. Test items which are grammatically incorrect but which do not violate the CL rules (this is described below under 'Grammar Checking').
4. Test items which are grammatically incorrect and which violate the CL grammar rules. The examples we use here are ungrammatical counterparts of the ones given above.

   a. This test follow the installation of aircraft landing gear.
   b. The controls not can be adjusted.

   For the first example we want the system to flag both the complex noun cluster *aircraft landing gear* (as seen above), since a compound can consist of no more than two nouns according to the CL specifications, and the non-agreement between the subject noun and the verb. In the second case, *can be adjusted* should be recognized as a disallowed phrase (it should be *you can adjust*) and at the same time the displacement of *not* should be indicated.

[12]The problem of interaction however remains: sentence (b) may be correctly treated by the system, but this may not be the case if a more complex test item had been constructed

CL Style Rules: These can only be tested to some extent, depending on the length and amount of test material required (for some rules this may be several paragraphs). Example:

Instructions in procedures (two rules apply):

1. Write only one instruction per sentence.
2. Write the verb in the imperative form.

Test sentences could include:

a. Set the switch to on and remove the bolts. (violates rule 1)
b. Set the switch to on. Remove the bolts. (OK)
c. You must be careful with the concentrated acid. (violates rule 2)
d. Be careful with the concentrated acid.

Grammar Checking: For the description of test material for this task we refer to the section on test suites for grammar checking 4.2. The requirements described there will apply here, with the additional requirement that test data should contain no violations of the CL, unless the evaluator is aiming at testing the interaction of the two.

4.3.4 Requirements on Test Suite Design: Presentation

The annotation used for grammar checkers can be used, but needs to be extended for CL errors, indicating e.g. whether the error violates vocabulary, grammar or style restrictions.

A *classification of errors* is not available.

5 The Customization of Test Suites

Several test suites have been designed as so-called general purpose test suites, intended for usage accross different applications by different users (see for example [4, 7]. Generality is a desirable feature, as it reduces the effort of test suite building that are required when test suites are not reusable. But the generality of such test suites does not guarantee their direct usefulness when applied to a certain system or type of system. It only guarantees that a certain number of linguistic phenomena have been listed without taking into account for what type of system the list may be used. In practice, this means that a number of aspects needs to be considered before the suite is used for a particular NL application. We will discuss these aspects here, mainly concentrating on the customization of test suites to a particular application, although sometimes some issues concerning system specific customization are discussed. From the discussion of these aspects, it will become clear that it is often necessary to adapt the test suite in order to make it more suitable for the testing of the application at hand.

5.1 Do Test Data Meet the Input Requirements?

Customizing a test suite in order to meet the requirements which a certain system puts on the format of its input is almost always necessary. Note that this is not so much a problem which affects the customization of a test suite for a certain application, but rather the customization for a specific system. We will give a short account of the problems that need to be considered under this heading.

The main problem is the format of the test data. This includes a wide range of issues, like the ones listed below. The evaluator has to decide how to deal with input that might cause problems.

- Are the test data in plain ASCII, or something else?
- Are the test data marked up according to any mark-up language (e.g. SGML)?
- What character set is used (especially from the point of view of accents, umlauts, special characters such as cédilles, etc)?
- Do the test data contain any lists, formulaes, figures or other elements that might be problematic?
- Are the test data annotated in any way, and does this annotation affect the way the input is read?

Another issue is the status of the test data: do they merely consist of sentences, or can we also find parts of sentences or pieces of discourse. Most systems are designed to deal with sentences, so non-sentential input may cause problems. If this is the case, this type of input should be excluded.

5.2 Coverage of the Test Data

One would hope that a general test suite is indeed general in the sense that it covers all relevant phenomena for all different applications. This is, however, an idealised and unrealistic view, as no existing test suite does this. It is actually probably unpractical and not even desirable. Customizing the test data in such a way that they cover all phenomena relevant to the application is therefore required in many cases.[13]

5.2.1 Syntactic Coverage

Although most test suites will probably aim at the inclusion of all (core) syntactic phenomena, gaps are likely to occur. It is up to the test suite builder to establish what changes and/or additions will need to be made. In this section we will only consider the application specific changes which are required, and leave system specific changes out of consideration.[14]

The first task of the test suite builder is to identify what phenomena are specific to the application and may therefore not be covered by a general test suite, or not be covered extensively enough. After establishing whether or not these phenomena are covered sufficiently by the test suite, the test suite builder will need to make the necessary extensions and additions.

[13] Of course it is not possible to cover all relevant phenomena, but one could aim for a thorough coverage of the relevant syntactic and extra-grammatical phenomena.

[14] Some system specific issues are discussed in section 5.3.

In order to establish in which areas of coverage a test suites falls short with regard to a particular application, it is extremely useful to have at one's disposal a test suite which indicates clearly what phenomena are included. Most test suites will give only a sketchy indication of this, as thorough annotation of the data is hardly ever done despite its obvious usefulness. Similarly, it is important to have an insight in what phenomena an application will aim to deal with. This type of knowledge may be acquired through:

1. The study of natural language texts which the application typically has to deal with.
2. The application of such texts as additional test material, so as to identify problem areas that may not yet be covered by the test suite.
3. In an individual system, the documentation of the rules in the grammars.

When extending a test suite with new material, evaluators must make sure that the new data test exactly what they wants to test, nothing more and nothing less. The 'nothing more' is hard to achieve as sentences always include a number of phenomena, so the evaluator should avoid including anything but phenomena already identified as unproblematic. Some useful rules for the extension of test suites are:[15]

1. Make sure test data for a certain phenomenon includes all possible variations that are desired for the application (completeness).
2. Make sure test data are as simple as possible, containing each only one problematic phenomenon, namely the one the evaluator wants to test. In this way it is easier to identify why a certain test sentence failed.
3. Make sure test data do not contain any phenomena that the application is not required to deal with.
4. Check all new data for grammar or spelling mistakes.

Note that point 3 also refers to test data already present in the general purpose test suite. If the test suite builder decides to omit all test items containing phenomena the system does not need to deal with, then it is necessary to check whether these phenomena might be present not only in test items labelled as dealing with this phenomenon, but also in test items that are labelled as dealing with other phenomena. In those cases, the test item must be replaced by another, more suitable, one.

In order to give an example of what might be identified as application specific phenomena, a short list discussing the applications relevant to us is given below:

Parsers This depends very much on the coverage of individual parsers. At first glance, we might want to assume that the coverage of most general purpose test suites is supposed to coincide with the list of phenomena dealt with by a general purpose parser. The term 'general' however, is in both cases rather vague and in practice discrepancies can be expected, although it is difficult to foresee where.

[15] See also [6].

Grammar Checkers A grammar checker should be able to 1) assign a correct (partial) parse to the input sentences and 2) identify the syntactic errors that occur in the input. The first task can be compared with the task of a parser, although syntactic analysis is generally fairly superficial. The second task has implications for system assessment, as it is necessary to establish whether the system gives the correct error message and what the rate of missed errors is (see [9]). In case the system also suggests corrections, the quality of these suggestions cannot yet be assessed by using test suites.

As the main function of this application is to pass a grammaticality judgement on input presented to it (which could be from any type of text type or subject area), it is important to include systematic testing of ill-formed input. It is difficult to establish how far one should take the testing of ill-formed input, but for most existing general test suites extension in this area would be necessary.

Other test data that may be required are long sentences (not desirable for general test suites) and those containing complex syntactic combinations. For this latter point, one might prefer to use test corpora rather than suites.

Controlled Language Checkers CL checkers are to some extent like grammar checkers and therefore the issues given above also apply here. But for CL checkers, the 'ungrammatical' input must augmented with input which does not comply with the constraints on the input. In order to be able to assess whether the system can deal with those constraints, the evaluator will have to 1) make considerable changes to the vocabulary of the test suite so as to match the vocabulary allowed by the system, 2) establish whether test data contain any un-allowed phenomena and replace those with others when necessary, and 3) include test data which are lexically and grammatically correct but are not allowed according to the input restraints of the system.

Although points 1-3 are application specific in the sense that they are necessary for the assessment of any CL checker, they are system specific in the sense that the changes to be made depend highly on the type of controlled language (and the related implemented grammar) that is being used. As far as we can see, it is not possible to make generalizations that are valid for all controlled language grammars.

5.2.2 Extra-grammatical Coverage

The treatment of extra-grammatical phenomena is essential for every system that is required to deal with real texts. But most existing test suites consider these phenomena as falling outside their scope, although some test suites may include some abbreviations, or punctuation marks. In general, however, this is an area where a lot of customization is required for those systems for which extensive testing of extra-grammatical phenomena is desirable, e.g. systems which are commercial or about to be commercialized.

One also needs to consider the possibility of including incorrect input in this area, e.g. the absence of a full stop, the absence of a closing bracket or parenthesis, the absence or conversely the addition of capital letters, etc. We do not consider this to be essential, but it might be for some systems.

5.3 System-specific Testing

A system may require extra testing in order to test specifically for a certain grammatical theory that has been adopted. For example, in the Eurotra Project, a large number of different verb frames were implemented which were crucial to the performance of the system. Most of these are not tested in any general test suite.

launch = subj_obj
give = subj_obj_iobj
cover = subj_obj_obl
drive = subj_obj_origin_goal

In such cases, quite extensive adaptations to the test suite are required. It could be argued that distinctions of the type given above should always be included in a general test suite, but this would bring us to another discussion, on the very complex topic of completeness. As it is highly unlikely that a truly complete test suite could be designed in the future, we will have to assume that discrepancies of the type illustrated above will occur. It is then up to the individual system evaluators to extend the test suite with data enabling them to assess those phenomena which are specific to their particular system.

5.4 Customizing Vocabulary

As many NLP systems tend to restrict the vocabulary they handle to certain domains, it is very likely that the evaluator will need to solve the problem of mismatched vocabulary between the test suite and the system to which it will be applied. There are two ways around this problem: 1) the vocabulary of the system is extended in order to include the unknown words of the test suite, 2) the vocabulary of the test suite is changed so that the system can deal with it. In most cases the second solution is chosen for obvious reasons.[16] We will therefore assume here that this is the standard procedure. This may however not be the case for systems which aim to be general, but then we may assume that lexical coverage is unproblematic.

In order to change the vocabulary of the test suite so that the system to be tested can deal with it we need to ensure the following:

- A lexical item which is chosen to replace a word from the original test suite should be a common word in the sense that it has no other features that could influence the results of the performance. After all, the test suite aims at covering general phenomena rather than phenomena that are relevant to specific lexical choices. This point is probably best illustrated with some examples.

  When replacing the vocabulary of test sentences containing agreement phenomena, it is important to choose nouns which do distinguish between singular and plural: *John bought this book* cannot be replaced with *John bought this salmon* as the noun *salmon* can be both plural and singular and thus agreement is not properly tested.

[16]It may not be desirable to extend vocabulary to subject areas that one does not intend to deal with. Also, dictionary updating can be rather time consuming and new, and untested lexical entries may not perform as well as might be expected.

We should also avoid possible idiomatic interpretations, e.g. when we want to include verbs that take two arguments, we should not replace *John broke a vase* with *John kicked the bucket*. It is quite possible that the system will deal with the second example in a completely different way than with the first, and then that particular argument frame would not be tested adequately.

It will be clear from these examples that a well-considered choice of vocabulary that is suitable to replace the old vocabulary is essential.

- A new lexical item should indeed be a proper replacement of the previous word, in the sense that it should have the same syntactic (and in some cases also semantic) requirements as the word it is replacing. This is very clear in the simple example of verbs taking two arguments, where *John visited a doctor* has a different argument frame from *John is a doctor*. Also, it is important to check that the (un-)grammaticality of the test item is preserved; for example, when replacing the vocabulary of test sentences containing relative pronouns, it is important not to replace an ungrammatical sentence like *List all employees which are working* with *List all printers which are working* as the word *printers* can be both human and non-human and therefore the ungrammatical sentence has become grammatical.

Again, annotation is an important issue here. It is very helpful if the test sentences indicate clearly what phenomena they contain, as this allows us to select new vocabulary which shows the same behaviour. It is similarly helpful if the system dictionaries have some means of indicating in what ways a particular word can be used (e.g. syntactic category, argument frame, selection of prepositions, etc); it helps to verify that the word selected is indeed appropriate. In some cases it is even necessary to establish whether the system's dictionaries do not exclude a particular usage of a word, thus making it impossible for the system to deal with a sentence that contains that usage.

5.5 Relating Weightings to Test Data

In some cases, test suites contain information concerning the importance of the phenomena included in the test data. This may be a simple marker indicating whether a certain sentence contains a core or a periphery phenomenon (as in the HP test suite), or a finer-grained code indicating different levels of importance. Of course, there are differences between applications; what might be a very common phenomenon for a NL query interface may very well be highly uncommon for a CL checker, especially if there are certain restrictions on text types.

If the test suite builders consider it desirable to have such a classification, it will be necessary to check the existing classification and adapt it to the frequency information that is available for the particular application, with regard to the relevant phenomena. In case the original test suite contains no information on the importance of test data, the test suite builders will need to design their own system. Of course, they may always decide to do even when the test suite already contains this information.

6 Conclusions

As mentioned at the beginning, this document only represents our current stage of thinking about the way to design and construct test suites. Our ideas will necessarily evolve and be subject to revision in the light of the experience of fully specifying and testing the annotation scheme in the next phase of the project, Task 2.2.

The subject of customization of test suites is also likely to be part of the reinforced testing in the planned extension to TSNLP, TSNLP+.

References

[1] G. Adriaens and D. Schreurs (1992), From Cogram to Alcogram: Toward a Controlled English Grammar Checker, in *Proceedings of Coling 1992*, Nantes.

[2] AECMA/AIA Simplified English

[3] Estival, Falkedal et al. (1994), *Analysis of Existing Test Suites*, Report to LRE 62-089 (D-WP1), University of Essex.

[4] Flickinger, D., J. Nerbonne, I.A. Sag & T. Wasow (1987). "Toward Evaluation of NLP Systems". Hewlett-Packard Laboratories, distr. at the 24th Annual Meeting of the Association for Computational Linguistics (ACL), Stanford.

[5] Marcus, Mitchell P., Mary Ann Marcinkiewicz, Beatrice Santorini (1993): "Building a Large Annotated Corpus of English: The Penn Treebank", *Computational Linguistics*, 19(2), pp.313-330.

[6] Neal et al. (1992), An Evaluation Methodology for Natural Language Processing Systems, RL-TR-92-308, Final Technical Report, Rome Laboratory, Air Force Material Command, Griffiss Air Force Base, New York.

[7] J. Nerbonne, K. Netter, A Kader Diagne, J. Klein, L. Dickman, *A Diagnostic Tool for German Syntax.* DFKI D-92-03, Saarbrücken. Also in: J. Neal / S. Walter. eds. (1991): *Natural Language Processing Systems Evaluation Workshop, Berkeley.* Rome Laboratory, RL-TR-91-362, New York

[8] SECC Technical Annex, 1993.

[9] Richard H. Wojcik, Philip Harrison, John Bremer (1993), Using Bracketed Parses to Evaluate a Grammar Checking Application, in: *Proceedings of ACL*, Boeing Computer Services Research and Technology Division, Seattle.