SimpleNLG: A realisation engine for practical applications

Albert Gatt and Ehud Reiter
Department of Computing Science
University of Aberdeen
Aberdeen AB24 3UE, UK
{a.gatt,e.reiter}@abdn.ac.uk

Abstract

This paper describes SimpleNLG, a realisation engine for English which aims to provide simple and robust interfaces to generate syntactic structures and linearise them. The library is also flexible in allowing the use of mixed (canned and non-canned) representations.

1 Introduction

Over the past several years, a significant consensus has emerged over the definition of the realisation task, through the development of realisers such as REALPRO (Lavoie and Rambow, 1997), ALETHGEN (Coch, 1996), KPML (Bateman, 1997), FUF/SURGE (Elhadad and Robin, 1996), HALOGEN (Langkilde, 2000), YAG (McRoy et al., 2000), and OPENCCG (White, 2006).

Realisation involves two logically distinguishable tasks. *Tactical generation* involves making appropriate linguistic choices given the semantic input. However, once tactical decisions have been taken, building a syntactic representation, applying the right morphological operations, and linearising the sentence as a string are comparatively mechanical tasks. With the possible exception of template-based realisers, such as YAG, existing wide-coverage realisers usually carry out both tasks. By contrast, a *realisation engine* focuses on the second of the two tasks, making no commitments as to how semantic inputs are mapped to syntactic outputs. This leaves the (tactical) problem of defining mappings from semantic inputs to morphosyntactic structures entirely up to the developer, something which may be attractive in those applications where full control of the output of generation is required. Such control is not always easily available in wide-coverage tactical generators, for a number of reasons:

1. Many such realisers define an input formalism, which effectively circumscribes the (semantic) space of possibilities that the realiser handles. The developer needs to ensure that the input to realisation is mapped to the requisite formalism.

2. Since the tactical problem involves search through a space of linguistic choices, the broader the coverage, the more efficiency may be compromised. Where real-time deployment is a goal, this may be an obstacle.

3. Many application domains have *sublanguage* requirements. For example, the language used in summaries of weather data (Reiter et al., 2005) or patient information (Portet et al., to appear) differs from standard usage, and does not always allow variation to the same extent. Since realisers don't typically address such requirements, their use in a particular application may require the alteration of the realiser's rule-base or, in the case of statistical realisers, re-training on large volumes of appropruately annotated data.

This paper describes SimpleNLG, a realisation engine which grew out of recent experiences in building large-scale data-to-text NLG systems, whose goal is to summarise large volumes of numeric and symbolic data (Reiter, 2007). Sublanguage requirements and efficiency are important considerations in such systems. Although meeting these requirements was the initial motivation behind SimpleNLG, it has since been developed into an engine with significant coverage of English syntax and morphology, while at the same time providing a simple API that offers users direct programmatic control over the realisation process.

	Feature	Values	Applicable classes
lexical	ADJPOSITION	Attrib$_{1/2/3}$, PostNominal, Predicative	ADJ
	ADVPOSITION	Sentential, PostVerbal, Verbal	ADV
	AGRTYPE	Count, Mass, Group, Inv-Pl, Inv-Sg	N
	COMPLTYPE	AdjP, AdvP, B-Inf, WhFin, WhInf, ...	V
	VTYPE	Aux, Main, Modal	V
phrasal	FUNCTION	Subject, Obj, I-Obj, Prep-Obj, Modifier	all
	SFORM	B-Inf, Gerund, Imper, Inf, Subj	S
	INTERROGTYPE	Yes/No, How, What, ...	S
	NUMBERAGR	Plural, Singular	NP
	TENSE	Pres, Past, Fut	VP
	TAXIS (boolean)	`true` (=perfective), `false`	VP
	POSSESSIVE (boolean)	`true` (=possessive), `false`	NP
	PASSIVE (boolean)	`true`, `false`	VP

Table 1: Features and values available in SimpleNLG

2 Overview of SimpleNLG

SimpleNLG is a Java library that provides interfaces offering direct control over the realisation process, that is, over the way phrases are built and combined, inflectional morphological operations, and linearisation. It defines a set of lexical and phrasal types, corresponding to the major grammatical categories, as well as ways of combining these and setting various feature values. In constructing a syntactic structure and linearising it as text with SimpleNLG, the following steps are undertaken:

1. Initialisation of the basic constituents required, with the appropriate lexical items;

2. Using the operations provided in the API to set features of the constituents, such as those in bottom panel of Table 1;

3. Combining constituents into larger structures, again using the operations provided in the API which apply to the constituents in question;

4. Passing the resulting structure to the lineariser, which traverses the constituent structure, applying the correct inflections and linear ordering depending on the features, before returning the realised string.

Constituents in SimpleNLG can be a mixture of canned and non-canned representations. This is useful in applications where certain inputs can be mapped to an output string in a deterministic fashion, while others require a more flexible mapping to outputs depending, for example, on semantic features and context. SimpleNLG tries to meet these needs by providing significant syntactic coverage with the added option of combining canned and non-canned strings.

Another aim of the engine is robustness: structures which are incomplete or not well-formed will not result in a crash, but typically will yield infelicitous, though comprehensible, output. This is a feature that SimpleNLG shares with YAG (McRoy et al., 2000). A third design criterion was to achieve a clear separation between morphological and syntactic operations. The lexical component of the library, which includes a wide-coverage morphological generator, is distinct from the syntactic component. This makes it useful for applications which do not require complex syntactic operations, but which need output strings to be correctly inflected.

2.1 Lexical operations

The lexical component provides interfaces that define a `Lexicon`, a `MorphologicalRule`, and a `LexicalItem`, with subtypes for different lexical classes (`Noun`, `Preposition` etc). Morphological rules, a re-implementation of those in MORPHG (Minnen et al., 2001), cover the full range of English inflection, including regular and irregular forms[1]. In addition to the range of morphological operations that apply to them, various features can be specified for lexical items. For example, as shown in the top panel of Table 1, adjectives and adverbs can be specified for their typical syntactic positions. Thus, an adjective such as *red* would have the values *Attrib*$_2$, indicating that it usually occurs in attribute position 2 (following *Attrib*$_1$ adjectives such as *large*), and *Predicative*. Similarly, nouns are classified to indicate

[1]Thanks are due to John Carroll at the University of Sussex for permission to re-use these rules.

their agreement features (count, mass, etc), while verbs can be specified for the range of syntactic complement types they allow (e.g. bare infinitives and WH-complements).

A typical development scenario involves the creation of a `Lexicon`, the repository of the relevant items and their properties. Though this can be done programmatically, the current distribution of SimpleNLG provides an interface to a database constructed from the NIH Specialist Lexicon[2], a large ($>$ 300,000 entries) repository of lexical items in the medical and general English domains, which incorporates information about lexical features such as those in Table 1.

2.2 Syntactic operations

The syntactic component of SimpleNLG defines interfaces for `HeadedPhrase` and `CoordinatePhrase`. Apart from various phrasal subtypes (referred to as `PhraseSpec`s) following the usage in Reiter and Dale (2000)), several grammatical features are defined, including `Tense`, `Number`, `Person` and `Mood` (see Table 1). In addition, a `StringPhraseSpec` represents a piece of canned text of arbitrary length.

A complete syntactic structure is achieved by initialising constituents with the relevant features, and combining them using the operations specified by the interface. Any syntactic structure can consist of a mixture of `Phrase` or `CoordinatePhrase` types and canned strings. The input lexical items to phrase constructors can themselves be either strings or lexical items as defined in the lexical component. Once syntactic structures have been constructed, they are passed to a lineariser, which also handles basic punctuation and other orthographic conventions (such as capitalisation).

The syntactic component covers the full range of English verbal forms, including participals, compound tenses, and progressive aspect. Subtypes of `CoordinatePhrase` allow for fully recursive coordination. As shown in the bottom panel of Figure 1, subjunctive forms and different kinds of interrogatives are also handled using the same basic feature-setting mechanism.

The example below illustrates one way of constructing the phrase *the boys left the house*, initialising a sentence with the main verb *leave* and setting a `Tense` feature. Note that the `SPhraseSpec` interface allows the setting of the main verb, although this is internally represented as the head of a `VPPhraseSpec` dominated by the clause. An alternative would be to construct the verb phrase directly, and set it as a constituent of the sentence. Similarly, the direct object, which is specified directly as a constituent of the sentence, is internally represented as the object of the verb phrase. In this example, the direct object is an `NPPhraseSpec` consisting of two words, passed as arguments and internally rendered as lexical items of type `Determiner` and `Noun` respectively. By contrast, the subject is defined as a canned string.

(1)
```
Phrase s1 =
  new SPhraseSpec('leave');
s1.setTense(PAST);
s1.setObject(
  new NPPhraseSpec('the', 'house'));
Phrase s2 =
  new StringPhraseSpec('the boys');
s1.setSubject(s2);
```

Setting the INTERROGATIVETYPE feature of sentence (1) turns it into a question. Two examples, are shown below. While (2) exemplifies a simple yes/no question, in (3), a WH-constituent is specified as establishing a dependency with the direct object (*the house*).

(2)
```
s1.setInterrogative(YES_NO);
```
(Did the boys leave home?)

(3)
```
s1.setInterrogative(WHERE, OBJECT);
```
(Where did the boys leave?)

In summary, building syntactic structures in SimpleNLG is largely a question of feature setting, with no restrictions on whether representations are partially or exclusively made up of canned strings.

2.2.1 Interaction of lexicon and syntax

The phrasal features in the bottom panel of Table 1 determine the form of the output, since they are automatically interpreted by the realiser as instructions to call the correct morphological operations on lexical items. Hence, the syntactic and morphological components are closely integrated (though distinct). Currently, however, lexical features such as ADJPOSITION are not fully integrated with the syntactic component. For example, although adjectives in the lexicon are specified for their position relative to other modifiers, and nouns are

[2] http://lexsrv3.nlm.nih.gov/SPECIALIST/index.html

specified for whether they take singular or plural agreement, this informaiton is not currently used automatically by the realiser. Full integration of lexical features and syntactic realisation is currently the focus of ongoing development.

2.3 Efficiency

As an indication of efficiency, we measured the time taken to realise 26 summaries with an average text length of 160.8 tokens (14.4 sentences), and sentences ranging in complexity from simple declaratives to complex embedded clauses[3]. The estimates, shown below, average over 100 iterations per text (i.e. a total of 2600 runs of the realiser) on a Dell Optiplex GX620 machine running Windows XP with a 3.16 GHz Pentium processor. Separate times are given for the initialisation of constituents based on semantic representations, along the lines shown in (1), (SYN), and linearisation (LIN). These figures suggest that a medium-length, multiparagraph text can be rendered in under a second in most cases.

	MEAN (ms)	SD	MIN	MAX
SYN	280.7	229.7	13.8	788.34
LIN	749.38	712.6	23.26	2700.38

3 Conclusions and future work

This paper has described SimpleNLG, a realisation engine which differs from most tactical generators in that it provides a transparent API to carry out low-level tasks such as inflection and syntactic combination, while making no commitments about input specifications or input-output mappings.

The simplicity of use of SimpleNLG is reflected in its community of users. The currently available public distribution[4], has been used by several groups for three main purposes: (a) as a front-end to NLG systems in projects where realisation is not the primary research focus; (b) as a simple natural language component in user interfaces for other kinds of systems, by researchers who do not work in NLG proper; (c) as a teaching tool in advanced undergraduate and postgraduate courses on Natural Language Processing.

SimpleNLG remains under continuous development. Current work is focusing on the inclusion of output formatting and punctuation modules, which are currently handled using simple defaults. Moreover, an enhanced interface to the lexicon is being developed to handle derivational morphology and a fuller integration of complementation frames of lexical items with the syntactic component.

[3]The system that generates these summaries is fully described by Portet *et al.* (to appear).

[4]SimpleNLG is available, with exhaustive documentation, at the following URL: `http://www.csd.abdn.ac.uk/~ereiter/simplenlg/`.

References

J. A. Bateman. 1997. Enabling technology for multilingual natural language generation: the KPML development environment. *Natural Language Engineering*, 3(1):15–55.

J. Coch. 1996. Overview of AlethGen. In *Proceedings of the 8th International Natural Language Generation Workshop*.

M. Elhadad and J. Robin. 1996. An overview of SURGE: A reusable comprehensive syntactic realization component. In *Proceedings of the 8th International Natural Language Generation Workshop*.

I. Langkilde. 2000. Forest-based statistical language generation. In *Proceedings of the 1st Meeting of the North American Chapter of the Association for Computational Linguistics*.

B. Lavoie and O. Rambow. 1997. A fast and portable realizer for text generation systems. In *Proceedings of the 5th Conference on Applied Natural Language Processing*.

S.W. McRoy, S. Channarukul, and S. Ali. 2000. YAG: A template-based generator for real-time systems. In *Proceedings of the 1st International Conference on Natural Language Generation*.

G. Minnen, J. J. Carroll, and D. Pearce. 2001. Applied morphological processing of English. *Natural Language Engineering*, 7(3):207–223.

F. Portet, E. Reiter, A. Gatt, J. Hunter, S. Sripada, Y. Freer, and C. Sykes. to appear. Automatic generation of textual summaries from neonatal intensive care data. *Artificial Intelligence*.

E. Reiter and R. Dale. 2000. *Building Natural Language Generation Systems*. Cambridge University Press, Cambridge, UK.

E. Reiter, S. Sripada, J. Hunter, J. Yu, and I. Davy. 2005. Choosing words in computer-generated weather forecasts. *Artificial Intelligence*, 167:137–169.

E. Reiter. 2007. An architecture for Data-to-Text systems. In *Proceedings of the 11th European Workshop on Natural Language Generation*.

M. White. 2006. Chart realization from disjunctive inputs. In *Proceedings of the 4th International Conference on Natural Language Generation*.