A Representation for Complex and Evolving Data Dependencies in Generation

C Mellish[‡], R Evans[†], L Cahill[†], C Doran[†*] D Paiva[†], M Reape[‡], D Scott[†], N Tipper[†]
[†]Information Technology Research Institute, University of Brighton, Lewes Rd, Brighton, UK
[‡]Division of Informatics, University of Edinburgh, 80 South Bridge, Edinburgh, UK
rags@itri.brighton.ac.uk
http:/www.itri.brighton.ac.uk/projects/rags

Abstract

This paper introduces an approach to representing the kinds of information that components in a natural language generation (NLG) system will need to communicate to one another. This information may be partial, may involve more than one level of analysis and may need to include information about the history of a derivation. We present a general representation scheme capable of handling these cases. In addition, we make a proposal for organising intermodule communication in an NLG system by having a central server for this information. We have validated the approach by a reanalysis of an existing NLG system and through a full implementation of a runnable specification.

1 Introduction

One of the distinctive properties of natural language generation when compared with other language engineering applications is that it has to take seriously the full range of linguistic representation, from concepts to morphology, or even phonetics. Any processing system is only as sophisticated as its input allows, so while a natural language understanding system might be judged primarily by its syntactic prowess, even if its attention to semantics, pragmatics and underlying conceptual analysis is minimal, a generation system is only as good as its *deepest* linguistic representations. Moreover, any attempt to abstract away from individual generation systems to a more generic architectural specification faces an even greater challenge: not only are complex linguistic representations required, able to support the dynamic evolutionary development of data during the generation process, but they must do so in a generic and flexible fashion.

This paper describes a representation developed to meet these requirements. It offers a formally well-defined declarative representation language, which provides a framework for expressing the complex and dynamic data requirements of NLG systems. The approach supports different levels of representation, mixed representations that cut across levels, partial and shared structures and 'canned' representations, as well as dynamic relationships between data at different stages in processing. We are using the approach to develop a high level data model for NLG systems as part of a generic generation architecture called RAGS[1].

The framework has been implemented in the form of a database server for modular generation systems. As proof of concept of the framework, we have reimplemented an existing NLG system. The system we chose was the Caption Generation System (CGS) (Mittal et al., 1995; Mittal et al., 1998). The reimplementation involved defining the interfaces to the modules of CGS in terms of the RAGS representations and then implementing modules that had the requisite input and output representations.

Generation systems, especially end-to-end, applied generation systems, have, unsurprisingly, many things in common. Reiter (1994) proposed an analysis of such systems in terms of a simple three stage pipeline. More recently, the RAGS project attempted to repeat the anal-

[*] Now at the MITRE Corporation, Bedford, MA, USA, cdoran@mitre.org.

[1] This work is supported by ESPRC grants GR/L77041 (Edinburgh) and GR/L77102 (Brighton), *RAGS: Reference Architecture for Generation Systems.* We would also like to acknowledge the contribution of Jo Calder to the ideas and formalisation described in this paper. In particular, parts of this paper are based on (Calder et al., 1999).

ysis (Cahill et al., 1999a), but found that while most systems did implement a pipeline, they did not implement the *same* pipeline – different functionalities occurred in different places and different orders in different systems. In order to accommodate this result, we sought to develop an architecture that is more general than a simple pipeline, and thus supports the range of pipelines observed, as well as other more complex control regimes (see (Cahill et al., 1999a; Cahill et al., 1999b)). In this paper, we argue that supporting such an architecture requires careful consideration of the way data representations interact and develop. Any formal framework for expressing the architecture must take account of this.

2 The representational requirements of generation systems

We noted in the introduction that generation systems have to deal with a range of linguistic information. It is natural, especially in the context of a generic architecture proposal, to model this breadth in terms of discrete layers of representation: (1999a) introduce layers such as conceptual, semantic, rhetorical, syntactic and document structure, but the precise demarcation is not as important here as the principle. The different kinds of information are typically represented differently, and built up separately. However the layers are far from independent: objects at one layer are directly related to those at others, forming chains of dependency from conceptual through rhetorical and semantic structure to final syntactic and document realisation. This means that data resources, such as grammars and lexicons, and processing modules in the system, are often defined in terms of **mixed** data: structures that include information in more than one representation layer. So the ability to represent such mixed structures in a single formal framework is an important property of a generic data proposal.

In addition, it is largely standard in generation as elsewhere in language applications, to make extensive use of **partial** representations, often using a type system to capture grades of underspecification. An immediate corollary of providing support for partial structures is the notion that they may become further specified over time, that data structures **evolve**. If the framework seeks to avoid over-commitment to particular processing strategies it needs to provide a way of representing such evolution explicitly if required, rather than relying on destructive modification of a structure. Related to this, it should provide explicit support for representing **alternative** specifications at any point. Finally, to fully support efficient processing across the range of applications, from the simple to the most complex, the representation must allow for compact sharing of information in **tangled** structures (two structures which share components).

In addition to these direct requirements of the generation task itself, additional requirements arise from more general methodological considerations: we desire a representation that is formally **well defined**, allows for theoretical **reasoning** about the data and performance of systems, and supports control regimes from simple deterministic pipelines to complex parallel architectures.

3 The Representation Scheme

In this section, we present our proposal for a general representation scheme capable of covering the above requirements. Our formulation is layered: the foundation is a simple, flexible, rigorously defined graph representation formalism, on top of which we introduce notions of complex types and larger data structures and relationships between them. This much is sufficient to capture the requirements just discussed. We suppose a yet higher level of specification could capture a more constraining data model but make no specific proposals about this here, however the following sections use examples that do conform to such a higher level data model.

The lowest level of the representation scheme is:

- **relational:** the basic data entity is $x \rightarrow y$, an *arrow* representing a relation from object x to object y;
- **typed:** objects and arrows have an associated type system, so it is possible to define classes and subclasses of objects and arrows.

At the most fundamental level, this is more or less the whole definition. There is no commitment to what object or arrow types there are or how they relate to each other. So a representation allowed by the scheme consists of:

- a set of objects, organised into types;
- a set of binary relations, organised into types;
- a set of arrows, each indicating that a relation holds between one object and another object.

Sets, sequences and functions

For the next level, we introduce more structure in the type system to support sets, sequences and functions. Objects are always atomic (though they can be of type set, sequence or function) – it is not possible to make an object which actually is a set of two other objects (as you might with data structures in a computer program). To create a set, we introduce a set *type* for the object, and a set membership arrow type (*el*), that links the set's elements to the set. Similarly, for a sequence, we introduce a sequence type and sequence member arrow types (*1-el, 2-el, 3-el*, ...), and for a function, we have a complex type which specifies the types of the arrows that make up the domain and the range of the function.

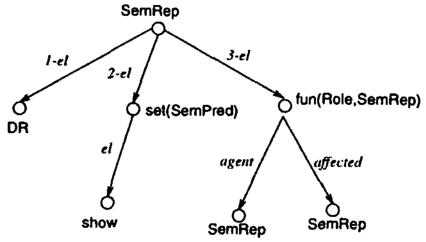

Figure 1: The partial semantic representation of "The second chart shows the number of days on the market"

As an example, consider Figure 1, which shows a semantic representation (SemRep) from the CGS reimplementation. Here, the tree nodes correspond to objects, each labelled with its type. The root node is of type SemRep, and although it is not an explicit sequence type, we can see that it is a triple, as it has three sequence member arrows (with types *1-el, 2-el* and *3-el*). Its first arrow's target is an object of type DR (Discourse Referent). Its second represents a set of SemPred (Semantic Predicate) objects, and in this case there's just one, of type show. Its third element is a (partial) function, from Role arrow types (*agent* and *affected* are both subtypes of Role) to SemReps. (In this case, the SemReps have not yet been fully specified.)

Local and non-local arrows

The second extension to the basic representation scheme is to distinguish two different abstract kinds of arrows – local and non-local. Fundamentally we are representing just a homogeneous network of objects and relationships. In the example above we saw a network of arrows that we might want to view as a single data structure, and other major data types might similarly appear as networks. Additionally, we want to be able to express relationships *between* these larger 'structures' – between structures of the same type (alternative solutions, or revised versions) or of different types (semantic and syntactic for example). To capture these distinctions among arrows, we classify our arrow types as local or non-local (we could do this in the type system itself, or leave it as an informal distinction). Local arrows are used to build up networks that we think of as single data structures. Non-local arrows express relationships between such data structures.

All the arrow types we saw above were local. Examples of non-local arrows might include:

realises These arrows link something more abstract to something less abstract that realises it. Chains of realises arrows might lead from the original conceptual input to the generator through rhetorical, semantic and syntactic structures to the actual words that express the input.

revises These arrows link a structure to another one of the same type, which is considered to be a 'better' solution – perhaps because it is more instantiated. It is important to note that parts of larger structures can be revised without revising the entire structure.

coreference These arrows link structures which are somehow "parallel" and which perhaps share some substructure, i.e., tangled structures. For instance, document representations may be linked to rhetorical representations, either as whole isomorphic structures or at the level of individual constituents.

Notice that the representation scheme does not enforce any kind of well-formedness with respect to local and non-local arrows. In fact, although it is natural to think of a 'structure' as being a maximal network of local arrows with a single root object, there's no reason why this should be so – networks with multiple roots represent tangled structures (structures that share content), networks that include non-local links might be *mixed* representations, containing information of more than one sort. Such techniques might be useful for improving generator efficiency, or representing canned text or templates, cf. (Calder et al., 1999).

Partial and Opaque structures

Partial structures are essential when a module needs to produce a skeleton of a representation that it does not have the competence to completely fill out. For instance, lexical choice brings with it certain syntactic commitments, but in most NLG systems lexical choice occurs some time before a grammar is consulted to flesh out syntactic structure in detail.

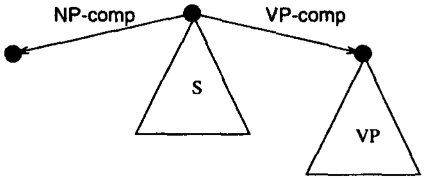

Figure 2: A partial structure

By simply leaving out local arrows, we can represent a range of partial structures. Consider Fig. 2, where the triangles represent local structure, representing a sentence object and its component verb phrase. There is a link to a subject noun phrase object, but none of the local arrows of the actual noun phrase are present. In subsequent processing this local structure might be filled in. This is possible as long as the noun phrase object has been declared to be of the right type.

An opaque structure is one which has an incomplete derivational history – for example part of a syntactic structure without any corresponding semantic structure. Three possible reasons for having such structures are (a) to allow structure to be introduced that the generator is not capable of producing directly, (b) to prevent the generator from interfering with the structure thus built (for example, by trying to modify an idiom in an inappropriate way), or (c) to improve generator efficiency by hiding detail that may lead to wasteful processing. An opaque structure is represented simply by the failure to include a `realises` arrow to that structure. Such structures provide the basis for a generalised approach to "canning".

4 Implementation

There are many ways that modules in an NLG system could communicate information using the representation scheme just outlined. Here we describe a particularly general model of inter-module communication, based around modules communicating with a single centralised repository of data called the *whiteboard* (Calder et al., 1999). A whiteboard is a *cumulative typed relational blackboard*:

- **typed and relational:** because it is based on using the above representation scheme;
- **a blackboard:** a control architecture and data store shared between processing modules; typically, modules add/change/remove objects in the data store, examine its contents, and/or ask to be notified of changes;
- **cumulative:** unlike standard blackboards, once data is added, it can't be changed or removed. So a structure is built incrementally by making successive copies of it (or of constituents of it) linked by `revises` links (although actually, there's no constraint on the order in which they are built).

A whiteboard allows modules to add arrows (typically forming networks through arrows sharing source or target objects), to inspect the set of arrows looking for particular configurations of types, or to be informed when a particular type of arrow (or group of arrows) is added.

The whiteboard is an active database server. This means that it runs as an independent process that other modules connect to by appropriate means. There are essentially three kinds of interaction that a module might have with the whiteboard server:

- **publish** – add an arrow or arrows to the whiteboard;

- **query** – look for an arrow or arrows in the whiteboard;
- **wait** – register interest in an arrow or arrows appearing in the whiteboard.

In both `query` and `wait`, arrows are specified by type, and with a hierarchical type system on objects and relations, this amounts to a pattern that matches arrows of subtypes as well. The `wait` function allows the whiteboard to take the initiative in processing – if a module `waits` on a query then the whiteboard waits until the query is satisfied, and then tells the module about it. So the module does not have to continuously scan the whiteboard for work to do, but can let the whiteboard tell it as soon as anything interesting happens.

Typically a module will start up and register interest in the kind of arrow that represents the module's input data. It will then wait for the whiteboard to notify it of instances of that data (produced by other modules), and whenever anything turns up, it processes it, adding its own results to the whiteboard. All the modules do this asynchronously, and processing continues until no module has any more work to do. This may sound like a recipe for confusion, but more standard pipelined behaviour is not much different. In fact, pipelining is exactly a data-based constraint – the second module in a pipeline does not start until the first one produces its output.

However, to be a strict pipeline, the first module must produce *all* of its output before the second one starts. This can be achieved simply by making the first module produce all its output at once, but sometimes that is not ideal – for example if the module is recursive and wishes to react to its own output. Alternative strategies include the use of markers in the whiteboard, so that modules can tell each other that they've finished processing (by adding a marker), or extending the whiteboard architecture itself so that modules can tell the whiteboard that they have finished processing, and other modules can wait for that to occur.

5 Reconstruction of the Caption Generation System

In order to prove this representation scheme in practice, we have implemented the whiteboard in Sicstus Prolog and used it to support data communications between modules in a reconstruction of the Caption Generation System (Mittal et al., 1995). CGS is a system developed at the University of Pittsburgh, which takes input from the SAGE graphics presentation system (Roth et al., 1994) and generates captions for the graphics SAGE produces. We selected it for this effort because it appeared to be a fairly simple pipelined system, with modules performing clearly defined linguistic tasks. As such, we thought it would be a good test case for our whiteboard specification.

Although the CGS is organised as a pipeline, shown in Figure 3, the representations communicated between the modules do not correspond to complete, separate instances of RAGS datatype representations. Instead, the representations at the various levels accumulate along the pipeline or are revised in a way that does not correspond exactly to module boundaries. Figure 3 gives a simple picture of how the different levels of representation build up. The labels for the RAGS representations refer to the following:

- I = conceptual;
- II = semantic;
- III = rhetorical;
- IV = document;
- V = syntactic.

For instance, some semantic (II) information is produced by the Text Planning module, and more work is done on this by Aggregation, but the semantic level of representation is not complete and final until the Referring Expression module has run. Also, for instance, at the point where the Ordering module has run, there are partially finished versions of three different types of representation. It is clear from this that the interfaces between the modules are more complex than could be accounted for by just referring to the individual levels of representation of RAGS. The ability to express combinations of structures and partial structures was fundamental to the reimplementation of CGS. We highlight below a few of the interesting places where these features were used.

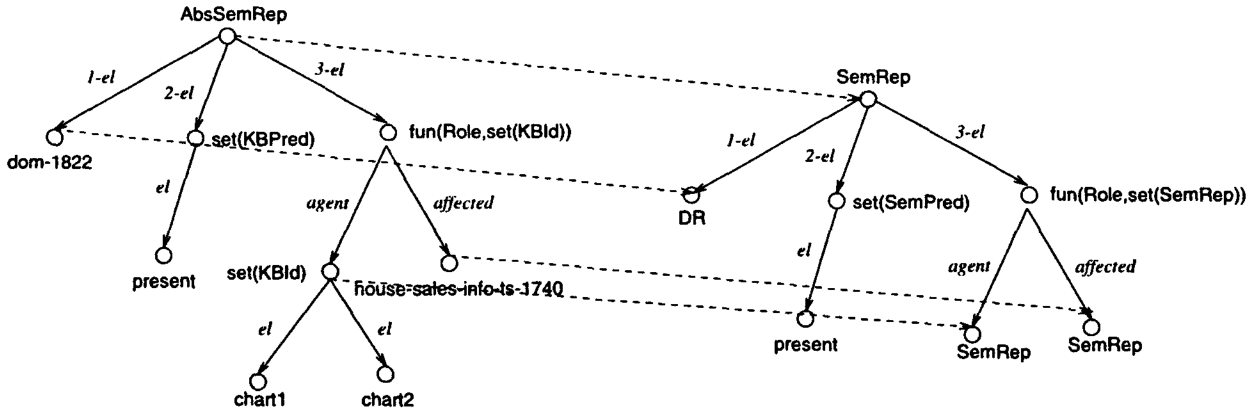

Figure 4: Combined Abstract Semantic Representation and Concrete Semantic Representation for the output: "These two charts present information about house sales from data-set ts-1740"

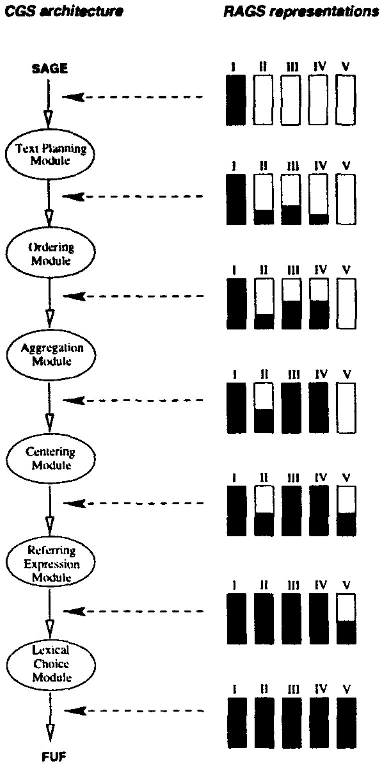

Figure 3: A RAGS view of the CGS system

5.1 Referring Expression Generation

In many NLG systems, (nominal) referring expression generation is an operation that is invoked at a relatively late stage, after the structure of individual sentences is fairly well specified (at least semantically). However, referring expression generation needs to go right back to the original world model/knowledge base to select appropriate semantic content to realise a particular conceptual item as an NP (whereas all other content has been determined much earlier). In fact, there seems to be no place to put referring expression generation in a pipeline without there being some resulting awkwardness.

In RAGS, pointers to conceptual items can be included inside the first, "abstract", level of semantic representation (AbsSemRep), which is intended to correspond to an initial bundling of conceptual material under semantic predicates. On the other hand, the final, "concrete", level of semantic representation (SemRep) is more like a fully-fledged logical form and it is no longer appropriate for conceptual material to be included there. In the CGS reimplementation, it is necessary for the Aggregation module to reason about the final high-level semantic representation of sentences, which means that this module must have access to "concrete" semantic representations. The Referring Expression generation module does not run until later, which means that these representations cannot be complete.

Our way around this was to ensure that the initial computation of concrete semantics from abstract semantics (done as part of Aggregation here) left a record of the relationship by including **realises** arrows between corresponding structures. That computation could not be completed whenever it reached conceptual material – at that point it left a "hole" (an object with no further specification) in the concrete semantic representation linked back to the conceptual material. When referring expression was later invoked, by following the arrows in the resulting mixed structure, it could tell exactly which conceptual entity needed to be referred to and where in the semantic structure the resulting semantic expression should be placed. Figure 4 shows the resulting arrangement for one example CGS sentence. The dashed lines indicate **realises**, i.e. non-local, arrows.

5.2 Handling Centering Information

The CGS Centering module reasons about the entities that will be referred to in each sentence and produces a representation which records the forward and backward-looking centers (Grosz et al., 1995). This representation is later used by the Referring Expression generation module in making pronominalisation decisions. This information could potentially also be used in the Realisation module.

Since Centering is not directly producing referring expressions, its results have to sit around until they can actually be used. This posed a possible problem for us, because the RAGS framework does not provide a specific level of representation for Centering information and therefore seems on first sight unable to account for this information being communicated between modules. The solution to the problem came when we realised that Centering information is in fact a kind of abstract syntactic information. Although one might not expect abstract syntactic structure to be determined until the Realisation module (or perhaps slightly earlier), the CGS system starts this computation in the Centering module.

Thus in the reimplementation, the Centering module computes (very partial) abstract syntactic representations for the entities that will eventually be realised as NPs. These representations basically just indicate the relevant Centering statuses using syntactic features. Figure 5 shows an example of the semantics for a typical output sentence and the two partial abstract syntactic representations computed by the Centering module for what will be the two NPs in that sentence[2]. As before, dashed lines indicate **realises** arrows. Of course, given the discussion of the last section, the semantic representation objects that are the source of these arrows are in fact themselves linked back to conceptual entities by being the destination of **realises** arrows from them.

When the Referring Expression generation module runs, it can recover the Centering information by inspecting the partial syntactic representations for the phrases it is supposed to generate. These partial representations are then further instantiated by, e.g., Lexical Choice at later stages of the pipeline.

6 Conclusion

The representation scheme we have proposed here is designed specifically to support the requirements of the current state-of-the-art NLG systems, and our pilot implementation demonstrates the practical applicability of the proposal. Tangled, partial and mixed structures are of obvious utility to any system with a flexible control strategy and we have shown here how the proposed representation scheme supports them. By recording the derivational history of computations, it also supports decisions which partly depend on earlier stages of the generation process (e.g., possibly, lexical choice) and revision-based architectures which typically make use of such information. We have shown how the representation scheme might be the basis for an inter-module communication model, the *whiteboard*, which supports a wide range of processing strategies that require the representation of complex and evolving data dependencies. The fact that the whiteboard is *cumulative*, or *monotonic* in a logical sense, means that the whiteboard also supports reasoning about the behaviour of NLG systems implemented in terms of it. This is something that we would like to exploit directly in the future.

The reimplementation of the CGS system in the RAGS framework was a challenge to the framework because it was a system that had already been developed completely independently. Even though we did not always understand the detailed motivation for the structure of CGS being as it was, within a short time we reconstructed a working system with modules that corresponded closely to the original CGS modules. The representation scheme we have proposed here was a key ingredient in giving us the flexibility to achieve the particular processing scheme used by CGS whilst remaining faithful to the (relatively simple) RAGS data model.

[2] FVM = Feature Value Matrix.

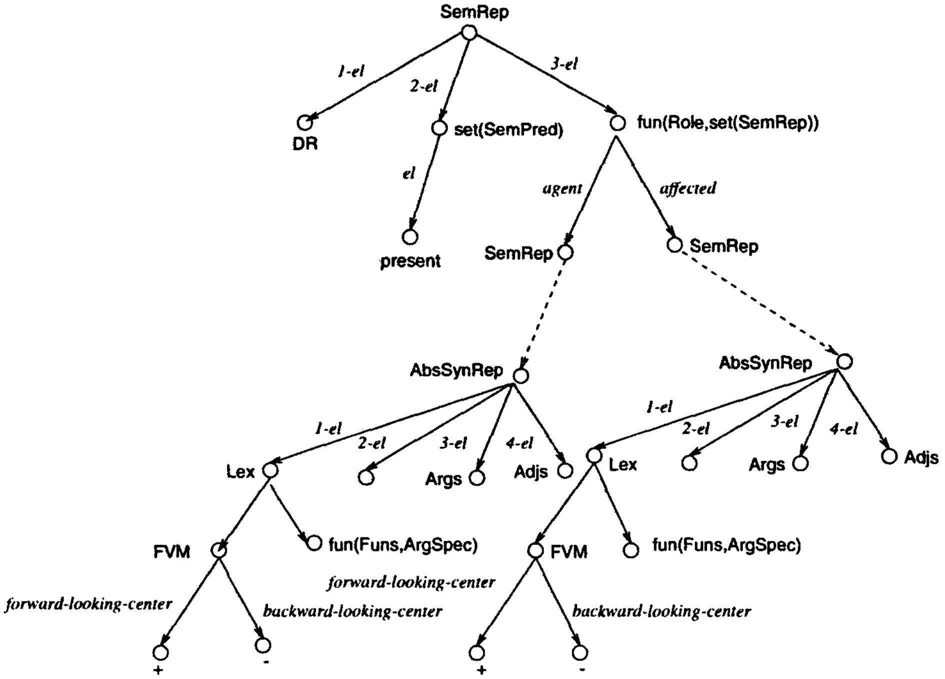

Figure 5: Arrangement of centering information for the output sentence above

The representation scheme is useful in situations where modules need to be defined and implemented to work with other modules, possibly developed by different people. In such cases, the representation scheme we propose permits precise definition of the interfaces of the modules, even where they are not restricted to a single 'level' of representation. Even though the control structure of CGS is quite simple, we found that the use of a centralised whiteboard was useful in helping us to agree on interfaces and on the exact contribution that each module should be making. Ultimately, it is hoped that the use of a scheme of this type will permit much more widespread 'plug-and-play' among members of the NLG community.

References

Lynne Cahill, Christy Doran, Roger Evans, Chris Mellish, Daniel Paiva, Mike Reape, Donia Scott, and Neil Tipper. 1999a. In Search of a Reference Architecture for NLG Systems. In *Proceedings of the 7th European Workshop on Natural Language Generation*, pages 77–85, Toulouse.

Lynne Cahill, Christy Doran, Roger Evans, Chris Mellish, Daniel Paiva, Mike Reape, Donia Scott, and Neil Tipper. 1999b. Towards a Reference Architecture for Natural Language Generation Systems. Technical Report ITRI-99-14, Information Technology Research Institute (ITRI), University of Brighton. Available at `http://www.itri.brighton.ac.uk/projects/rags`.

Jo Calder, Roger Evans, Chris Mellish, and Mike Reape. 1999. "Free choice" and templates: how to get both at the same time. In *"May I speak freely?" Between templates and free choice in natural language generation*, number D-99-01, pages 19–24. Saarbrücken.

B.J. Grosz, A.K. Joshi, and S. Weinstein. 1995. Centering: a framework for modelling the local coherence of discourse. *Computational Linguistics*, 21(2):203–226.

V. O. Mittal, S. Roth, J. D. Moore, J. Mattis, and G. Carenini. 1995. Generating explanatory captions for information graphics. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI'95)*, pages 1276–1283, Montreal, Canada, August.

V. O. Mittal, J. D. Moore, G. Carenini, and S. Roth. 1998. Describing complex charts in natural language: A caption generation system. *Computational Linguistics*, 24(3):431–468.

Ehud Reiter. 1994. Has a consensus NL generation architecture appeared and is it psycholinguistically plausible? In *Proceedings of the Seventh International Workshop on Natural Language Generation*, pages 163–170, Kennebunkport, Maine.

Steven F. Roth, John Kolojejchick, Joe Mattis, and Jade Goldstein. 1994. Interactive graphic design using automatic presentation knowledge. In *Proceedings of CHI'94: Human Factors in Computing Systems*, Boston, MA.