From RAGS to RICHES: exploiting the potential of a flexible generation architecture*

Lynne Cahill†, John Carroll‡, Roger Evans†, Daniel Paiva†,
Richard Power†, Donia Scott† and **Kees van Deemter†**
†ITRI, University of Brighton
Brighton, BN2 4GJ, UK
Firstname.Lastname@itri.bton.ac.uk

‡School of Cognitive and Computing Sciences, University of Sussex
Brighton, BN1 9QH, UK
johnca@cogs.susx.ac.uk

Abstract

The RAGS proposals for generic specification of NLG systems includes a detailed account of data representation, but only an outline view of processing aspects. In this paper we introduce a modular processing architecture with a concrete implementation which aims to meet the RAGS goals of transparency and reusability. We illustrate the model with the RICHES system – a generation system built from simple linguistically-motivated modules.

1 Introduction

As part of the RAGS (Reference Architecture for Generation Systems) project, Mellish et al (2000) introduces a framework for the representation of data in NLG systems, the RAGS 'data model'. This model offers a formally well-defined declarative representation language, which supports the complex and dynamic data requirements of generation systems, e.g. different levels of representation (conceptual to syntax), mixed representations that cut across levels, partial and shared structures and 'canned' representations. However RAGS, as described in that paper, says very little about the functional structure of an NLG system, or the issues arising from more complex processing regimes (see for example Robin (1994), Inuie et al., (1992) for further discussion).

NLG systems, especially end-to-end, applied NLG systems, have many functionalities in common. Reiter (1994) proposed an analysis of such systems in terms of a simple three stage pipeline. More recently Cahill et al (1999) attempted to repeat the analysis, but found that while most systems did implement a pipeline, they did not implement the *same* pipeline – different functionalities occurred in different ways and different orders in different systems. But this survey did identify a number of core *functionalities* which seem to occur during the execution of most systems. In order to accommodate this result, a 'process model' was sketched which aimed to support both pipelines and more complex control regimes in a flexible but structured way (see (Cahill et al., 1999),(RAGS, 2000)). In this paper, we describe our attempts to test these ideas in a simple NLG application that is based on a concrete realisation of such an architecture[1].

The RAGS data model aims to promote comparability and re-usability in the NLG research community, as well as insight into the organisation and processing of linguistic data in NLG. The present work has similar goals for the processing aspects: to propose a general approach to organising whole NLG systems in a way which promotes

*We would like to acknowledge the financial support of the EPSRC (RAGS – *Reference Architecture for Generation Systems*: grant GR/L77102 to Donia Scott), as well as the intellectual contribution of our partners at Edinburgh (Chris Mellish and Mike Reape: grant GR/L77041 to Mellish) and other colleagues at the ITRI, especially Nedjet Bouayad-Agha. We would also like to acknowledge the contribution of colleagues who worked on the RICHES system previously: Neil Tipper and Rodger Kibble. We are grateful to our anonymous referees for their helpful comments.

[1]More details about the RAGS project, the RICHES implementation and the OASYS subsystem can be found at the RAGS project web site: http://www.itri.bton.ac.uk/projects/rags.

the same ideals. In addition, we aim to test the claims that the RAGS *data* model approach supports the flexible *processing* of information in an NLG setting.

2 The RAGS data model

The starting point for our work here is the RAGS data model as presented in Mellish et al (2000). This model distinguishes the following five levels of data representation that underpin the generation process:

Rhetorical representations *(RhetReps) define how propositions within a text are related. For example, the sentence "Blow your nose, so that it is clear" can be considered to consist of two propositions:* BLOW YOUR NOSE *and* YOUR NOSE IS CLEAR*, connected by a relation like* MOTIVATION.

Document representations *(DocReps) encode information about the physical layout of a document, such as textual level (paragraph, orthographic sentence, etc.), layout (indentation, bullet lists etc.) and their relative positions.*

Semantic representations *(SemReps) specify information about the meaning of individual propositions. For each proposition, this includes the predicate and its arguments, as well as links to underlying domain objects and scoping information.*

Syntactic representations *(SynReps) define "abstract" syntactic information such as lexical features (*FORM, ROOT *etc.) and syntactic arguments and adjuncts (*SUBJECT, OBJECT *etc.).*

Quote representations *These are used to represent literal unanalysed content used by a generator, such as canned text, pictures or tables.*

The representations aim to cover the core common requirements of NLG systems, while avoiding over-commitment on less clearly agreed issues relating to conceptual representation on the one hand and concrete syntax and document rendering on the other. When one considers processing aspects, however, the picture tends to be a lot less tidy: typical modules in real NLG systems often manipulate data at several levels at once, building structures incrementally, and often working with 'mixed' structures, which include information from more than one level. Furthermore this characteristic remains even when one considers more purely functionally-motivated 'abstract' NLG modules. For example, Referring Expression Generation, commonly viewed as a single task, needs to have access to at least rhetorical and document information as well as referencing and adding to the syntactic information.

To accommodate this, the RAGS data model includes a more concrete representational proposal, called the 'whiteboard' (Calder et al., 1999), in which all the data levels can be represented in a common framework consisting of networks of typed 'objects' connected by typed 'arrows'. This *lingua franca* allows NLG modules to manipulate data flexibly and consistently. It also facilitates modular design of NLG systems, and reusability of modules and data sets. However, it does not in itself say anything about *how* modules in such a system might interact.

This paper describes a concrete realisation of the RAGS object and arrows model, OASYS, as applied to a simple but flexible NLG system called RICHES. This is not the first such realisation: Cahill et al., (2000) describes a partial re-implementation of the 'Caption Generation System' (Mittal et al., 1999) which includes an objects and arrows 'whiteboard'. The OASYS system includes more specific proposals for processing and inter-module communication, and RICHES demonstrates how this can be used to support a modular architecture based on small scale functionally-motivated units.

3 OASYS

OASYS (Objects and Arrows SYStem) is a software library which provides:

- an implementation of the RAGS Object and Arrows (O/A) data representation,
- support for representing the five-layer RAGS data model in O/A terms,
- an event-driven active database server for O/A representations.

Together these components provide a central core for RAGS-style NLG applications, allowing separate parts of NLG functionality to be specified in independent modules, which communicate exclusively via the OASYS server.

The O/A data representation is a simple typed network representation language. An O/A database consists of a collection of *objects*, each of which has a unique identifier and a type, and

arrows, each of which has a unique identifier, a type, and source and target objects. Such a database can be viewed as a (possibly disconnected) directed network representation: the figures in section 5 give examples of such networks.

OASYS pre-defines object and arrow types required to support the RAGS data model. Two arrow types, `el` (element) and `el(<integer>)`, are used to build up basic network structures – `el` identifies its target as a member of the set represented by its source, `el(3)`, identifies its target as the third element of the tuple represented by its source. Arrow type `realised_by` relates structures at different levels of representation. for example, indicating that this SemRep object is realised by this SynRep object. Arrow type `revised_to` provides for support for non-destructive modification of a structure, mapping from an object to another of the same type that can be viewed as a revision of it. Arrow type `refers_to` allows an object at one level to indirectly refer to an object at a different level. Object types correspond to the types of the RAGS data model, and are either atomic, tuples, sets or sequences. For example, document structures are built out of DocRep (a 2-tuple), DocAttr (a set of DocFeatAtoms – feature-value pairs), DocRepSeq (a sequence of DocReps or DocLeafs) and DocLeafs.

The active database server supports multiple independent O/A databases. Individual modules of an application publish and retrieve objects and arrows on databases, incrementally building the 'higher level', data structures. Modules communicate by accessing a shared database. Flow of control in the application is *event-based*: the OASYS module has the central thread of execution, calls to OASYS generate 'events', and modules are implemented as event handlers. A module registers interest in particular kinds of events, and when those events occur, the module's hander is called to deal with them, which typically will involve inspecting the database and adding more structure (which generates further events).

OASYS supports three kinds of events: **publish events** occur whenever an object or arrow is published in a database, **module lifecycle events** occur whenever a new module starts up or terminates, and **synthetic events** – arbitrary messages passed between the modules, but not interpreted by OASYS itself – may be generated by modules at any time. An application starts up by initialising all its modules. This generates *initialise* events, which at least one module must respond to, generating further events which other modules may respond to, and so on, until no new events are generated, at which point OASYS generates *finalise* events for all the modules and terminates them.

This framework supports a wide range of architectural possibilities. Publish events can be used to make a module wake up whenever data of a particular sort becomes available for processing. Lifecycle events provide, among other things, an easy way to do pipelining: the second module in a pipeline waits for the *finalise* event of the first and then starts processing, the third waits similarly for the second to finalise etc. Synthetic events allow modules to tell each other more explicitly that some data is ready for processing, in situation where simple publication of an object is not enough.

RICHES includes examples of all three regimes: the first three modules are pipelined using lifecycle events; LC and RE, FLO and REND interact using synthetic events; while SF watches the database specifically for publication events.

4 RICHES

The RICHES system is a simple generation system that takes as input rhetorical plans and produces patient advice texts. The texts are intended to resemble those found at the PharmWeb site (`http://www.pharmweb.net`). These are simple instructional texts telling patients how to use certain types of medicines, such as nosedrops, eye drops, suppositories etc.. An example text from PharmWeb is shown in figure 1, alongside the corresponding text produced by RICHES.

The main aim of RICHES is to demonstrate the feasibility of a system based on both the RAGS data model and the OASYS server model. The modules collectively construct and access the data representations in a shared blackboard space and this allows the modules to be defined in terms of their functional role, rather than say, the kind of data they manipulate or their position in a processing pipeline. Each of the modules in the sys-

PharmWeb

How to Use Nose Drops

1. Blow your nose gently, so that it is clear.
2. Wash your hands.
3. Unscrew the top of the bottle and draw some liquid into the dropper.
4. Tilt your head back.
5. Hold the dropper just above your nose and put the correct number of drops into your nostril.
6. DO NOT let the dropper touch the inside of your nose.
7. Keep your head tilted back for two to three minutes to help the drops run to the back of your nose.
8. Replace the top on the bottle.

- KEEP ALL MEDICINES OUT OF THE REACH OF CHILDREN

PharmWeb - Copyright©1994-2001. All rights reserved

Blow your nose so that it is clear.

Wash your hands

Unscrew the top. Then draw the liquid into the dropper.

Tilt your head back

Hold the dropper above your nose. Then put the drops into your nostril.

The dropper must not touch the inside.

Keep your head tilted back for two to three minutes so that the drops run to the back.

Replace the top on the bottle

Generated by RICHES version 1.0 (9/5/2001) on 9/5/2001
©2001, ITRI, University of Brighton

Figure 1: An example text from PharmWeb, together with the corresponding text generated by RICHES

tem is in itself very simple – our primary interest here is in the way they interact.

Figure 2 shows the structure of the system[2]. The functionality of the individual modules is briefly described below.

Rhetorical Oracle (RO) The input to the system is a RhetRep of the document to be generated: a tree with internal nodes labelled with (RST-style) rhetorical relations and RhetLeaves referring to semantic proposition representations (SemReps). RO simply accesses such a representation from a data file and initialises the OASYS database.

Media Selection (MS) RICHES produces documents that may include pictures as well as text. As soon as the RhetRep becomes available, this module examines it and decides what can be illustrated and what picture should illustrate it. Pictures, annotated with their SemReps, are part of the picture library, and Media Selection builds small pieces of DocRep referencing the pictures.

Document Planner (DP) The Document Planner, based on the ICONOCLAST text planner (Power, 2000) takes the input RhetRep and produces a document structure (DocRep). This specifies aspects such as the text-level (e.g., `paragraph`, `sentence`) and the relative ordering of propositions in the DocRep. Its leaves refer to SynReps corresponding to syntactic phrases. This module is pipelined after MS, to make sure that it takes account of any pictures that have been included in the document.

Lexical Choice (LC) Lexical choice happens in two stages. In the first stage, LC chooses the lexical items for the predicate of each SynRep. This fixes the basic syntactic structure of the proposition, and the valency mapping between semantic and syntactic arguments. At this point the basic document structure is complete, and the LC advises REND and SF that they can start processing. LC then goes into a second phase, in-

[2] The dashed lines indicate flow of information, solid arrows indicate approximately flow of control between modules, double boxes indicate a completely reused module (from another system), while a double box with a dashed outer indicates a module partially reused. Ellipses indicate information sources, as opposed to processing modules.

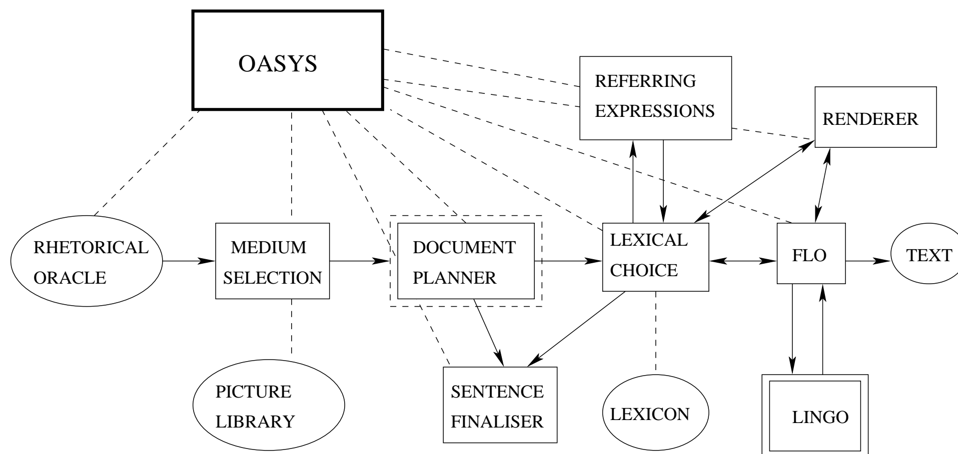

Figure 2: The structure of the RICHES system

terleaved with RE and FLO: for each sentence, RE determines the referring expressions for each noun phrase, LC then lexicalises them, and when the sentence is complete FLO invokes LinGO to realise them.

Referring Expressions (RE) The Referring Expression module adapts the SynReps to add information about the form of a noun phrase. It decides whether it should be a pronoun, a definite noun phrase or an indefinite noun phrase.

Sentence Finaliser (SF) The Sentence Finaliser carries out high level sentential organisation. LC and RE together build individual syntactic phrases, but do not combine them into whole sentences. SF uses rhetorical and document structure information to decide how to complete the syntactic representations, for example, combining main and subordinate clauses. In addition, SF decides whether a sentence should be imperative, depending on who the reader of the document is (an input parameter to the system).

Finalise Lexical Output (FLO) RICHES uses an external sentence realiser component with its own non-RAGS input specification. FLO provides the interface to this realiser, extracting (mostly syntactic) information from OASYS and converting it to the appropriate form for the realiser. Currently, FLO supports the LinGO realiser (Carroll et al., 1999), but we are also looking at FLO modules for RealPro (Lavoie and Rambow, 1997) and FUF/SURGE (Elhadad et al., 1997).

Renderer (REND) The Renderer is the module that puts the concrete document together. Guided by the document structure, it produces HTML formatting for the text and positions and references the pictures. Individual sentences are produced for it by LinGO, via the FLO interface. FLO actually processes sentences independently of REND, so when REND makes a request, either the sentence is there already, or the request is queued, and serviced when it becomes available.

LinGO The LinGO realiser uses a wide-coverage grammar of English in the LKB HPSG framework, (Copestake and Flickinger, 2000). The tactical generation component accepts input in the Minimal Recursion Semantics formalism and produces the target text using a chart-driven algorithm with an optimised treatment of modification (Carroll et al., 1999). No domain-specific tuning of the grammar was required for the RICHES system, only a few additions to the lexicon were necessary.

5 An example: generation in RICHES

In this section we show how RICHES generates the first sentence of the example text, *Blow your nose so that it is clear* and the picture that accompanies the text.

The system starts with a rhetorical representation (RhetRep) provided by the RO (see Figure 3)[3]. The first active module to run is MS

[3]In the figures, labels indicate object types and the subscript numbers are identifiers provided by OASYS for each

which traverses the RhetRep looking at the semantic propositions labelling the RhetRep leaves, to see if any can be illustrated by pictures in the picture library. Each picture in the library is encoded with a semantic representation. Matching between propositions and pictures is based on the algorithm presented in Van Deemter (1999) which selects the most informative picture whose representation contains nothing that is not contained in the proposition. For each picture that will be included, a leaf node of document representation is created and a `realised_by` arrow is added to it from the semantic proposition object (see Figure 4).

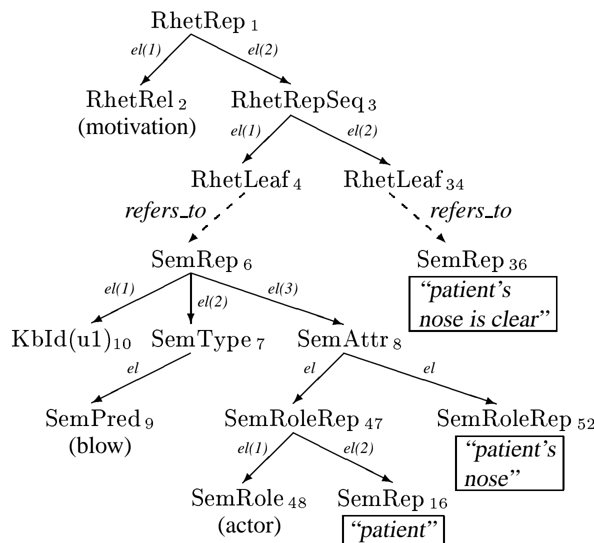

Figure 3: Initial rhetorical and semantic representations

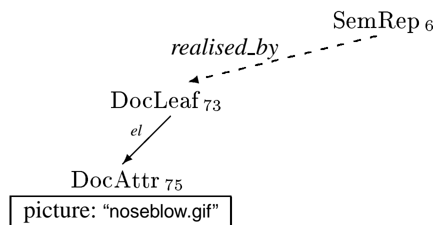

Figure 4: Inclusion of a picture by MS

The DP is an adaptation of the ICONOCLAST constraint-based planner and takes the RhetRep as its input. The DP maps the rhetorical representation into a document representation, deciding how the content will be split into sentences, paragraphs, item lists, etc., and what order the elements will appear in. It also inserts markers that will be translated to cue phrases to express some rhetorical relations explicitly. Initially the planner creates a skeleton document representation that is a one-to-one mapping of the rhetorical representation, but taking account of any nodes already introduced by the MS module, and assigns finite-domain constraint variables to the features labelling each node. It then applies constraint satisfaction techniques to identify a consistent set of assignments to these variables, and publishes the resulting document structure for other modules to process.

In our example, the planner decided that the whole document will be expressed as a paragraph (that in this case consists of a single text sentence) and that the document leaves will represent text-phrases. It also decides that these two text-phrases will be linked by a 'subordinator' marker (which will eventually be realised as "so that"), that "patient blows patient's nose" will be realised before "patient's nose is clear". At this stage, the representation looks like Figure 5.

The first stage of LC starts after DP has finished and chooses the lexical items for the main predicates (in this case "blow" and "clear"). These are created as SynReps, linked to the leaves of the DocRep tree. In addition the initial SynReps for the syntactic arguments are created, and linked to the corresponding arguments of the semantic proposition (for example, syntactic SUBJECT is linked to semantic ACTOR). The database at this stage (showing only the representation pertinent to the first sentence) looks like Figure 6.

Until this point the flow of control has been a straight pipeline. Referring Expression Generation (RE) and the second stage of Lexical Choice (LC) operate in an interleaved fashion. RE collects the propositions in the order specified in the document representation and, for each of them, it inspects the semantic entities it contains (e.g., for our first sentence, those entities are 'patient' and 'nose') to decide whether they will be realised as a definite description or a pronoun. For our example, the final structure for the first argument in the first sentence can be seen in Figure 7 (although note that it will not be realised explicitly because

object. Those parts inside boxes are simplifications to the actual representation used in order not to clutter the figures.

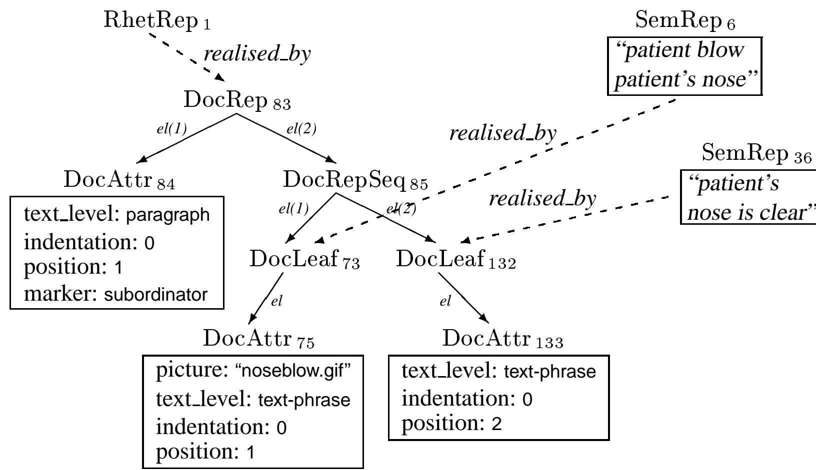

Figure 5: Document representation

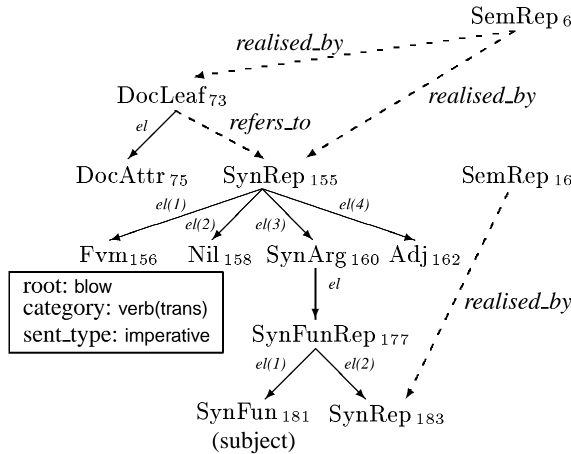

Figure 6: First stage of Lexical Choice – part of sentence 1

the sentence is an imperative one).

SF waits for the syntactic structure of indvidual clauses to be complete, and then inspects the syntactic, rhetorical and document structure to decide how to combine clauses. In the example, it decides to represent the rhetorical 'motivation' relation within a single text sentence by using the subordinator 'so that'. It also makes the main clause an imperative, and the subordinate clause indicative.

As soon as SF completes a whole syntactic sentence, FLO notices, and extracts the information required to interface to LinGO with an MRS structure. The string of words returned by LinGO, is stored internally by FLO until REND requests it.

Finally, REND draws together all the information from the document and syntactic structures, and the realiser outputs provided by FLO, and produces HTML. The entire resultant text can be seen on the right hand side of figure 1.

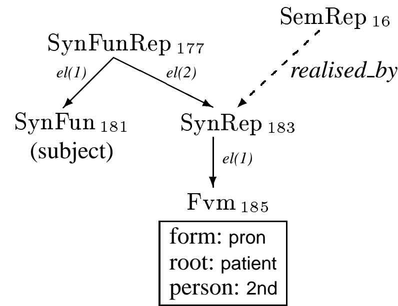

Figure 7: Second stage of Lexical Choice – entity 1 of sentence 1

6 Summary

In this paper, we have described a small NLG system implemented using an event-driven, object-and-arrow based processing architecture. The system makes use of the data representation ideas proposed in the RAGS project, but adds a concrete proposal relating to application organisation and process control. Our main aims were to develop this 'process model' as a complement to the RAGS 'data model,' show that it could be implemented and used effectively, and test whether the RAGS ideas about data organisation and development can actually be deployed in such a system. Although the RICHES generator is quite

simple, it demonstrates that it is possible to construct a RAGS-style generation system using these ideas, and that the OASYS processing model has the flexibility to support the kind of modularised NLG architecture that the RAGS initiative presupposes.

Some of the complexity in the RICHES system is there to demonstrate the potential for different types of control strategies. Specifically, we do not make use of the possibilities offered by the interleaving of the RE and LC, as the examples we cover are too simple. However, this setup enables RE, in principle, to make use of information about precisely how a previous reference to an entity has been *realised*. Thus, if the first mention of an entity is as “the man”, RE may decide that a pronoun, “he” is acceptable in a subsequent reference. If, however, the first reference was realised as “the person”, it may decide to say “the man” next time around.

At the beginning of this paper we mentioned systems that do not implement a standard pipeline. The RICHES system demonstrates that the RAGS model is sufficiently flexible to permit modules to work concurrently (as the REND and LC do in RICHES), alternately, passing control backwards and forwards (as the RE and LC modules do in RICHES) or pipelined (as the Document Planner and LC do in RICHES).

The different types of events allow for a wide range of possible control models. In the case of a simple pipeline, each module only needs to know that its predecessor has finished. Depending on the precise nature of the work each module is doing, this may be best achievable through publish events (e.g. when a DocRep has been published, the DP may be deemed to have finished its work) or through lifecycle events (e.g. the DP effectively states that it has finished). A revision based architecture might require synthetic events to “wake up” a module to do some more work, after it has finished its first pass.

References

Lynne Cahill, Christine Doran, Roger Evans, Chris Mellish, Daniel Paiva, Mike Reape, Donia Scott, and Neil Tipper. 1999. In search of a reference architecture for NLG systems. In *Proceedings of the Seventh European Natural Language Generation Workshop*, Toulouse, France.

Lynne Cahill, Christine Doran, Roger Evans, Chris Mellish, Daniel Paiva, Mike Reape, Donia Scott, and Neil Tipper. 2000. Reinterpretation of an existing NLG system in a Generic Generation Architecture. In *Proceedings of the First International Natural Language Generation Conference*, pages 69–76, Mitzpe Ramon, Israel.

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.

John Carroll, Ann Copestake, Dan Flickinger, and Victor Poznanski. 1999. An efficient chart generator for (semi-)lexicalist grammars. In *Proceedings of the 7th European Workshop on Natural Language Generation (EWNLG’99)*, pages 86–95, Toulouse, France.

Ann Copestake and Dan Flickinger. 2000. An open source grammar development environment and broad-coverage English grammar using HPSG. In *Proceedings of the 2nd International Conference on Language Resources and Evaluation*, Athens, Greece.

Michael Elhadad, Kathleen McKeown, and Jacques Robin. 1997. Floating constraints in lexical choice. *Computational Linguistics*, 23(2):195–240.

K. Inui, T. Tokunaga, and H. Tanaka. 1992. Text revision: A model and its implementation. In R. Dale, E. Hovy, D. Rosner, and O. Stock, editors, *Aspects of Automated Natural Language Generation*, number LNAI-587. Springer-Verlag.

B. Lavoie and O. Rambow. 1997. A fast and portable realizer for text generation systems. In *Proceedings of the Fifth Conference on Applied Natural Language Processing*, pages 265–68, Washington, DC.

Chris Mellish, Roger Evans, Lynne Cahill, Christy Doran, Daniel Paiva, Mike Reape, Donia Scott, and Neil Tipper. 2000. A representation for complex and evolving data dependencies in generation. In *Language Technology Joint Conference, ANLP-NAACL2000*, Seattle.

V. O. Mittal, J. D. Moore, G. Carenini, and S. Roth. 1999. Describing complex charts in natural language: A caption generation system. *Computation Linguistics*.

Richard Power. 2000. Planning texts by constraint satisfaction. In *Proceedings of the 18th International Conference on Computational Linguistics (COLING-2000)*, pages 642–648, Saarbrücken, Germany.

RAGS. 2000. Towards a Reference Architecture for Natural Language Generation Systems. Technical report, Information Technology Research Institute (ITRI), University of Brighton. Available at http://www.itri.brighton.ac.uk/projects/rags .

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.

J. Robin. 1994. Revision-Based Generation of Natural Language Summaries Providing Historical Background:Corpus-Based Analysis, Design, Implementation and Evaluation. Technical Report CUCS-034-94, Columbia University.

K. van Deemter. 1999. Document generation and picture retrieval. In *Procs. of Third Int. Conf. on Visual Information Systems (VISUAL-99), Springer Lecture Notes in Computer Science no. 1614*, pages 632–640, Amsterdan, Netherlands.