*Natural Language Engineering* **1** (1): 000–000. Printed in the United Kingdom

# *Implementation Architectures for Natural Language Generation*

Chris Mellish
*Department of Computing Science*
*University of Aberdeen,*
*King's College Aberdeen AB24 3UE, UK*
cmellish@csd.abdn.ac.uk

Roger Evans
*Information Technology Research Institute,*
*University of Brighton,*
*Brighton, BN2 4GJ, UK*
Roger.Evans@itri.brighton.ac.uk

(*Received 17 February 2004*)

**Abstract**

Generic software architectures aim to support re-use of components, focusing of research and development effort, and evaluation and comparison of approaches. In the field of natural language processing, generic frameworks for understanding have been successfully deployed to meet all of these aims, but nothing comparable yet exists for generation. The nature of the task itself, and the current methodologies available to research it, seem to make it more difficult to reach the necessary level of consensus to support generic proposals. Recent work has made progress towards establishing a generic framework for generation at the functional level, but left open the issue of actual implementation. In this paper, we discuss the requirements for such an implementation layer for generation systems, drawing on two initial attempts to implement it. We argue that it is possible and useful to distinguish "functional architecture" from "implementation architecture" for generation systems.

## 1 The Case for a Generic Software Architecture for NLG

Most natural language generation (NLG) systems have some kind of modular structure. The individual modules may differ in complex ways, according to whether they are based on symbolic or statistical models, what particular linguistic theories they embrace and so on. Ideally, such modules could be reused in other NLG systems. This would avoid duplication of work, allow realistic research specialisation and allow empirical comparison of different approaches. Examples of ideas that might give rise to reusable modules include:

**A program for generating referring expressions** based on statistical analysis of a significant corpus of human-written text (Cheng *et al.* 2001)

**A surface realiser** based on one of the most comprehensive generation grammars of English (Elhadad and Robin 1992)

**A text planner** using stochastic search (Mellish *et al.* 1998) or constraint satisfaction (Power 2000)

Although *conceptually* such modules could work together in one system (researchers at the international NLG meetings are basically speaking the same language at some level), in practice it is far from easy to build an NLG system using the best modules around. This is because today's modules are written in different programming languages, assume different representations for data, are described in terms of theory-dependent terminology and are designed to run in many different overall NLG architectures. In fact, many important ideas are not even made available in software form, simply because there is no obvious single way to package such software so as to be useful to others.

An agreed NLG *software architecture* would provide a framework for building modules that were compatible and facilities to aid in the construction, operation and evaluation of modular NLG systems. Although such software architectures exist for tasks in natural language understanding (NLU) (e.g. GATE (Cunningham, Wilks, and Gaizauskas 1996)), there is as yet nothing similar for generation. There *are* success stories of reuse of NLG modules (usually involving realisation components such as MUMBLE (McDonald 1981), FUF/Surge (Elhadad and Robin 1992) and REALPRO (Lavoie and Rambow 1997)), though often the reuse involves a lot of work and fails to fully exploit strengths. There are also examples of mature and complex software systems for building NLG systems which have been used for multiple projects (e.g. KPML (Bateman 1997)), but these require one to embrace a particular theoretical outlook and express one's whole system in terms of it. We do not believe that the NLG field is mature enough yet to opt for a single linguistic theory or programming strategy to take precedence at the expense of other ideas[1].

It is important to recognise that existing architectural proposals for natural language processing (NLP) tasks fall into two broad kinds. The first kind primarily provides a functional specification of the underlying data representations and/or module definitions. Those architectures aimed at supporting the annotation of corpora (for instance, ATLAS (Laprun *et al.* 2002), the Linguistic Annotation Framework (Ide and Romary 2004) and the Text Encoding Initiative are of this nature, as are the "generic architectures" for information extraction (Hobbs 1993) or question-answering (Hirschman and Gaizauskas 2001) systems. We shall refer to such proposals as *functional architectures*. The second kind of architecture provides direct support for the development of software systems by providing underlying software services, such as optimised data storage, inter-process communication facilities and a repertoire of control structures (Cunningham, Wilks, and Gaizauskas 1996; Brew

[1] We do not believe this for NLU either, but the nature of the understanding task, and the dependence of the dominant methodology on training from pre-existing test sets, induces greater alignment of approaches than seems to have occurred for NLG to date.

*et al.* 2000; Bayer *et al.* 2001; Herzog *et al.* 2004). We shall refer to such architectures as *implementation architectures.* Implementation architectures necessarily presuppose some (often quite small) degree of functional specification, and hence some functional architecture. The combination of functional and implementation architectures into a single framework is what we take to be a *software architecture* (although we acknowledge that not all software architectures make such a clean internal distinction).

This distinction is relevant here, because while no viable complete software architecture exists for NLG, there has been recent progress on a functional architecture for NLG, notably the RAGS proposal (Cahill *et al.* 2001b; Mellish *et al.* 2004). RAGS establishes *inter alia* some quite demanding requirements for the implementation layer, but provides no proposal for it. This paper concentrates on the question of what an implementation architecture for NLG could and should be like.

## 2 The Scope of an NLG Software Architecture

A candidate 'generic' NLG software architecture has to be broadly compatible with most work in the field and attractive to enough researchers that it will be used. In the end, the architecture will survive only if on the one hand researchers contribute good-quality modules in a form compatible with it and on the other hand other researchers actually reuse these modules using it. It is important, therefore, to limit what the architecture tries to do, in order not to stray beyond this fragile "consensus". There is a tradeoff here between the number of potential users and the amount of support it can give. The more theory-dependent the architecture is, the fewer the people that can benefit from it, but the more it can help those people. So what do current NLG researchers agree and not agree on? Some of the things on which there *doesn't* seem to be a consensus include:

**The individual modules making up an NLG system.** Just about every NLG system divides the task differently. The different modules proposed do not even seem to group consistently into larger components (Cahill and Reape 1999).

**How the modules are organised and controlled.** There have been a number of different architectures used (De Smedt *et al.* 1996). Often a pipeline architecture is used (Reiter 1994), though it is well-known that there are fundamental problems with such an architecture (Danlos 1984).

**The input to NLG.** Because of the range of applications considered, different NLG systems do not agree about the original conceptual input (e.g. on the amount of linguistically-relevant structure it has) or the role of goals or user models in the generation process (Evans *et al.* 2002).

The absence of agreement about what modules are involved in NLG means that a generally useful software architecture must allow people to define their own modules in a flexible way. Hopefully, the most useful modules will become widely used and hence achieve prominence, but it should be the *users* that make this happen, not the software architecture. Similarly a useful architecture should allow people to "plug together" modules in many ways. It should not legislate unnecessarily on the overall control regime. Finally, the problem of saying something general about the input of NLG means that the architecture can make few assumptions about this.

¿From the above, one might take the view that the only useful generic software architecture for NLG would be something like a programming language. Such a conclusion would, however, be overly pessimistic. Although NLG systems all have unique problems because of idiosyncrasies in their *input*, all of them have to produce broadly the same kind of *output*, i.e. natural language. All NLG systems have to represent their output at various levels, and many NLG modules are defined primarily in terms of the manipulations they perform on linguistic representations of some kind. A wide range of linguistic theories have been used as the basis of NLG systems' representations, but underneath the differences, there are often striking similarities — for example many semantic representation schemes are basically versions of the Predicate Calculus — and in general distinctions between syntax, semantics, rhetoric etc. are widely agreed. It is on the representation of linguistic data that NLG systems agree most, and this is the most promising aspect of NLG to be built into and supported by a software architecture.

Similarly, although the overall picture of modules and their relationship to each other in actual systems is confused, researchers *are* strongly wedded to the existence of particular functionalities within the NLG process, such as text structuring, referring expression generation or lexical choice. This fact may also provide some leverage in the development of a software architecture for NLG. The problem is that these functionalities do not seem to map into *localised* processing in a clean way — there is no clear mapping from logical functions to actual modules in implemented systems. But an architecture that allows some flexibility in this mapping, for example by separating the logical grouping of code implementing particular functionality from its deployment during the generation process, may be better able to support re-use of functional components.

## 3 A Functional Architecture for NLG

It is interesting how the above arguments about the scope of a software architecture for NLG recapitulate similar arguments made to motivate the GATE architecture for NLU (Cunningham, Wilks, and Gaizauskas 1996). The success of GATE is due to a large extent to the fact that it has not tried to do too much. Yet it supports key aspects of current NLU research methodology (corpus storage, annotation, evaluation) that all researchers must face to some extent. To put it another way, GATE combines a lightweight functional architecture, whose focus is on data interchange, with a powerful implementation architecture. A similar approach applied to NLG might have similar success. The key differences appear to be:

- The range and complexity of data structures presupposed by NLG systems.
- The subtleties of interaction between module and control structure.

The GATE functional architecture is based on an underlying theory of the nature of the data involved, as embodied in the TIPSTER architecture (Grishman

1995). TIPSTER specifies the relevant attributes of stored text and the kinds of annotations that can be attached to it. This "backbone" (honoured by the GATE implementations) is then the basis for individual researchers defining their own modules in a compatible way — the functional architecture requires modules to be indivisible processing units but does not constrain their organisation relative to each other. It is then the GATE user who chooses from the available modules and decides how to plug the chosen modules together in a specific order. If such a functional model existed for NLG, it could be the basis for a similar sort of architecture.

The RAGS architecture (Cahill *et al.* 2001b; Mellish *et al.* 2004) is an initial attempt to be such a functional architecture specification. It defines abstractly six levels of linguistic representation relevant to NLG (conceptual, document, rhetorical, semantic, syntactic and quote[2]), together with an underlying model, the "Objects and Arrows" model, to support data manipulation[3]. It also identifies seven functional modules which make up the core of an NLG application (lexicalisation, aggregation, rhetorical structuring, referring expression generation, ordering, segmentation and coherence maintenance (Cahill and Reape 1999)), but specifies no control constraints, not even that each module is indivisible.

The advantages of RAGS are that:

- It is based on substantial work attempting to understand current practice in NLG.
- It is very general, allowing for instance for the definition of datasets that mix the different levels of representation, provide partial information, use (acyclic) non-tree-shaped structures, etc.
- The data proposal is precisely defined and includes an XML "interchange format" for data.

On the other hand, RAGS is still relatively untried by the NLG community as a whole. It is also relatively complex, and as a functional architecture it lacks implementation-level detail. The rest of this paper assumes that something like RAGS will be needed for an NLG functional architecture, though it may well have to differ from RAGS in significant ways. The discussion will attempt to be independent of particular details of RAGS (though the two prototype implementations described are both based on RAGS).

[2] Quote representations are used to allow data to include pieces of fixed material which can be incorporated unchanged in the generator's output.

[3] The Objects and Arrows model is a formal definition of a well-formed RAGS dataset (what it can and cannot express; what distinctions are meaningful and what distinctions are not). It describes possible data of the six linguistic types in terms of a common low-level representation as directed acyclic graphs. Implementations following RAGS can represent this graph structure directly or they can use any other data format that preserves the relevant distinctions.

## 4 Choices for an Implementation Architecture

One possible approach to developing an NLG software architecture would be to take an architecture for NLU and "simply" replace the functional architecture by one developed with generation in mind. For instance, one could take GATE and reimplement the central database to store datasets in the manner suggested by RAGS instead of TIPSTER. Or one could take the idea of cascaded XML transducers suggested by the LT XML framework (Brew *et al.* 2000) and require that the format of XML transduced be that specified by RAGS. This latter approach gains some support from the existing examples of modular NLG systems that are implemented using XML pipelines (e.g. using XSLT) (Wilcock 2001; Seki 2001; Barrutieta *et al.* 2002). However, such an approach risks losing functionality in the NLG functional model (if it is not supported by the NLU implementation model) and limits the support provided by the implementation model to a NLU-oriented view of what may be useful.

Alternatively, instead of adapting an NLU architecture, one could devise a completely new architecture of some kind. In order to see what kind of architecture would be most useful, it is necessary to have a better understanding of NLG and current NLG practice than is probably readily available. The following are some features that an NLG implementation architecture could be expected to support in applications. According to one's conclusions about the importance of these, different architectures may be suggested. The best implementation architecture for NLG will support a combination of features which takes into account that an architecture that is unnecessarily general may also be one that is unnecessarily hard to use.

**Non-determinism.** NLG is a search problem. NLG modules themselves may or may not implement searches and may or may not wish to produce several answers. The architecture can choose to support modules producing multiple alternative results (non-determinism), or it can choose to require unique results (determinism). As an extreme case, "overgeneration" approaches to NLG require the production and storage of a very large number of alternatives (Langkilde and Knight 1998; Langkilde-Geary 2002). The NLG systems cited above that use XML pipelines are deterministic, and this is reflected in their use of XML. If the XML represents a sequential structure (a bit like a text) then alternatives tend to give rise to overlapping elements. These can't be represented in a single XML document, though they can be handled using stand-off annotation or a chart-like database as in GATE. NLG has to be able to generate alternative results, if only to achieve stylistic goals that cannot be checked until the surface is reached (Reiter 2000), and so simple uses of XML and XSLT are not theoretically adequate for all NLG tasks. The RAGS data model *is* able to represent alternatives, but the result is increased complexity, and it remains to be seen how practical transformations based, say, on XSLT are for this use of XML.

**Non-monotonicity.** Within one particular generation alternative, an architecture can require that over time the system only ever *adds* information about this alternative (monotonic), or it could also permit changes and deletions of information (non-monotonic). Many NLG systems have components that repeatedly revise and optimise structures to be generated (e.g. performing types of aggregation), and indeed some NLG systems are fundamentally based on the notion of revision (which is inherently non-monotonic) (Robin and McKeown 1996). On the other hand, GATE assumes that NLU happens through a monotonic accumulation of information about the text. A monotonic NLG architecture is possible if revision processes in an NLG system can be internalised within individual modules, or if there is a fixed number of revision stages (each with its own separate representation). Alternatively, the approach taken in RAGS is to allow revisions to be represented explicitly in an evolving dataset (Mellish *et al.* 2000), This provides another way of doing NLG in a monotonic architecture, though at the expense of complexity in the data representation.

**Eclecticism.** An NLG system builder may want to make use of modules that make alternative (even incompatible) representation assumptions in different parts of the system. For instance, representation of syntactic structures may be necessary for both a referring expression generation module and a surface realisation module, but the two desired modules may do this in different ways. The system may be able to handle this by building multiple representations and/or translating between different representations. The architecture can support this kind of eclecticism by allowing the two kinds of syntactic representation to be kept separate and to have separate roles in the system as a whole. Or it can make this sort of activity difficult by forcing global consistency of the representations.

**Flexibility of roles.** The modules of an NLG system play particular roles within the system as a whole. The calling pattern of the modules (when they are called and what for) reflects this. An architecture can support very flexible roles (exactly when a module is called is determined at runtime according to the particular circumstances prevailing), or it can be very inflexible (the sequence of calls is laid down in advance). The need for flexible roles has been argued by those NLG researchers who doubt the adequacy of pipelines for NLG (Danlos 1984) and those who have used blackboards of various kinds (Wanner and Hovy 1996; Rubinoff 1992; Nirenburg *et al.* 1989). One argument is that making the decisions to produce an optimal text requires different sources of information to be taken into account in unpredictable orders (or suffers severe efficiency problems). Another argument is that language is recursive in structure and that the calling patterns will reflect this. For instance, text planning may take place before referring expression generation, but referring expression generation (if it decides to introduce a relative clause) will need to invoke more text planning and recursively more referring expression generation. The depth of this process cannot be predicted in advance, and so some flexibility has to be left in the pattern of module invocation.

**Parallelism.** For real-time NLG, one might require an architecture to support parallelism where different NLG tasks are more or less independent. Indeed, one might rely on this in a multi-engine approach that tries out different possible approaches and takes the first answer available or the best answer, such as VERBMOBIL (Wahlster 2001). Certain kinds of models of human language production may also require support for parallelism (De Smedt 1994).

**Central controllability.** In a modular system, each module may be largely in control of its own operation and deciding when it communicates with other modules. On the other hand, there will be at least some element of central control, in that the system as a whole has to start up and close down (although in web- or grid-based approaches, modules may exists independently as persistent services). Central controllability here refers to the ability of some central process to impose a complex algorithm on the operation of the modules (rather than there being a fixed algorithm). This might be implemented via a dedicated control module of some kind. An architecture will be thought of as supporting central controllability if it is possible to "program" it so as to produce behaviour such as global backtracking or constraint satisfaction. In a centrally controlled system, the individual modules need have little idea of how and when they are to be used, or of the significance of their results. On the other hand, in an architecture which is not centrally controllable, the overall behaviour will emerge from that of the individual modules with no central conductor. The *possibility* to have central controllability is motivated if one might want to implement NLG based on, for instance, constraint-satisfaction algorithms (Beale *et al.* 1998).

**Modules with state.** Some NLG modules probably are not naturally thought of in terms of "one shot" operations that are performed independently. Such modules require a notion of internal state that is maintained between invocations. For example, a lexical choice module might want to maintain a memory of previous words used in order to achieve elegant variation. A referring expression generation module might want to maintain its own internal model of the discourse and to generate referring expressions in sequential activations, one phrase at a time, rather than to be provided with a complete discourse to work on. Although there are ways to work around this, XSLT pipelines do not naturally allow modules to maintain internal state.

**Complex modules.** Some NLG modules may wish to bring together multiple functionalities in a single unit. For example, lexical choice takes place in two distinct phases in many NLG systems (early lexicalisation of semantic predicates and late lexicalisation as syntactic structure is built up – see (Cahill 1999)), but may be implemented by a single logical module, which maintains state across all the lexicalisation decisions of the system, whenever they are needed.

The choice of a software architecture also depends on the set of services it is to provide. It is unclear at present whether an NLG system needs fundamentally different services from NLU. Certainly some aspects are likely to be broadly similar – tracing of system operation, data import/export, corpus-based evaluation and statistical analysis and visualisation of results, for instance.

## 5 Two Prototype Implementation Architectures

In this section we briefly present two prototype implementation architectures for NLG that have arisen from the RAGS project. Both of these are still only research prototypes (and in particular have hardly addressed user interface issues), but they have been used to construct non-trivial NLG systems. Both architectures in fact make use of the RAGS data model, but both could easily be adapted to handle data according to another agreed data model with a similar nature to RAGS.

### *5.1 RAGSOCKS*

#### *5.1.1 Basic Mechanisms*

The RAGSOCKS architecture implements a method of indirect point-to-point communication between software modules which makes use of a central server to process all communications. In some ways, this makes it similar to the DARPA Communicator architecture (Bayer *et al.* 2001), a distributed hub-and-spoke architecture based on message passing. However, whereas in Communicator the hub is a programmable resource that runs a tailored script to control the overall message traffic, in RAGSOCKS the central "hub" is supposed to be invisible, modules communicating as if they had direct access to one another. The presence of the central server limits the knowledge that modules need to have of one another. This makes it easy to unplug a module and plug in a replacement. The underlying communication mechanism is sockets, which connect modules running concurrently, on the same machine or on different machines connected by a network. Code for the central server and to support the writing of RAGSOCKS modules in Java, LISP and Prolog is freely available from the RAGS website `http://www.itri.brighton.ac.uk/rags/`.

#### *5.1.2 Module Conception*

Each RAGSOCKS module has a name, the *process name*, which identifies its role in the overall system and it also has names for (any number of, unidirectional) logical communication channels that it will use. It needs to have no knowledge of the names or whereabouts of other modules. A module sends and receives RAGS representations via the central server (identifying itself by its process name and specifying the relevant logical communication channel) using facilities in its own programming language (Prolog, Java or LISP). It is assumed that the module is representing RAGS-compatible data in either one of the native programming language formats supported or in some other native format that it can translate into one of the supported formats. The actual communication of data is done via sockets, using XML as an interchange language, with the translation to and from XML happening invisibly. The communication model is a simple one – the sender sends a complete set of RAGS representations into the channel as its half of the transaction (sending data cannot be interleaved with other processing). These can be representations of any size or type, as long as they are legal RAGS representations. The sender need not wait for the data to be received before then doing something else (which could include sending data in a new transaction). The receiver, on deciding that it is time for an input, hangs until all the data has been transferred and then is able to process that data. If data has been sent in several different transactions, the receiver will get the first set when it first looks, the second set when it next looks, etc. It needs to know when to look for input data (for instance, when it has nothing else to do).

### *5.1.3 Central Control*

A single *configuration file* describes the desired configuration of modules, indicating how the communication channels are plugged together ("process A's channel with name NA plugs into process B's channel with name NB"). The configuration file is interpreted by the central server to allow the actual communication to happen. The server does not need to know the nature of the actual modules – any process can connect to the server, claiming a process name that is present in the configuration file, and it can then use the logical channels associated with that process. Of course, the person constructing the configuration file has to know enough about the modules to ensure that output channels are connected to input channels, that the types of information provided at an output correspond to what is required at the corresponding input and that any synchronisation requirements between the communications are met by the modules (e.g. "whenever module A gets an input on channel 1, it produces an output on either channel 2 or channel 3").

The central server has no control over the operation of the modules, which are started up and closed down by the user independently of it.

### *5.1.4 Services*

The existence of a central server through which all communications pass means that there is a straightforward way to watch what is happening in the system. The server provides a display which shows the state of play for all the connections listed in the configuration file. This indicates by colours which modules are waiting for input, which are producing output and what communication is currently taking place via the server. Because the display can change quickly, one has the option to temporarily "freeze" the system (i.e. the server, which means that no new communications can be initiated; the modules will continue running until they need the server) and then let it continue again. If one is interested in a particular connection in the system, it is possible via the server's display panel to initiate tracing, which means that all traffic along the connection is displayed in a dedicated window (in raw XML form). A connection can also be reset (if there is an error) via the display. The only direct global user control is over the server and its connections, though the user also starts and stops the modules and may interact with them individually in whatever ways they allow.

Just as a module can send and receive RAGS representations via logical channels connected to other modules, facilities are provided so that they can be loaded and stored from disc files. This must, however, be programmed within the individual

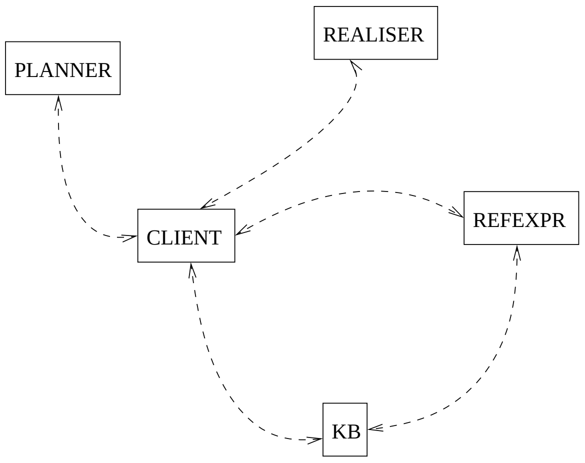

Fig. 1. The architecture of a RAGSOCKS system

modules, and there is no special support for the organisation of offline data, batch processing of multiple datasets or collation of results.

### *5.1.5 Example System*

RAGSOCKS has been used to implement a simple reconstruction of the ILEX system (O'Donnell *et al.* 2001) using some pre-existing generation components. The architecture of this system is shown in figure 1. Dashed lines indicate the communication channels between modules. All of these in fact go via the central server, which is not shown. The following briefly describes the modules and their interaction.

**client -** A central module implemented in LISP that communicates with the other modules (which essentially act as servers) as required. This module has logical channels for output to the other modules (`tokb, toplanner, torealiser, torefexpr`) and for input from the other modules (`fromkb, fromplanner, fromrealiser, fromrefexpr`).

**kb -** The domain knowledge base required for content determination and certain linguistic decisions (in fact, a reformulation of the ILEX knowledge base as a Prolog database). This acts a server implementing the RAGS API for knowledge bases, with logical channels `in` (for queries) and `out` (for responses).

**planner -** A text planner that assembles a set of facts and possible rhetorical relations into an overall text plan (in fact, a Prolog implementation of a stochastic text planner (Mellish *et al.* 1998)). This has logical channels `in` and `out`.

**realiser -** A surface realiser for mixed syntactic and semantic representations which returns strings ready for output (in fact, the LISP implementation of the FUF/Surge system (Elhadad and Robin 1992)). Again with logical channels `in` and `out`.

**refexpr -** A module for generating referring expressions (various possibilities have been implemented in LISP and Java). This module also needs to access the `kb` module. It has logical channels `in` and `out` (for the main input/output), as well as `kbin` and `kbout` (for the communications with the KB).

In this case, RAGSOCKS has been used to implement a Communicator-like system, where **client** is the hub, performing some generation tasks (e.g. content determination) and orchestrating the operation of the other main modules (**planner, realiser** and **refexpr**). However, **refexpr**, as well as **client**, has direct access to the knowledge base module, which diverges from the hub-and-spoke pattern.

## *5.2 OASYS*

### *5.2.1 Basic Mechanisms*

OASYS ("Object and Arrows SYStem") provides a blackboard architecture supporting event-driven activation of software modules. A "blackboard" is a monotonically increasing set of data items, and the principal event type is publication of data items on a blackboard. Multiple blackboards are supported to allow shared or private communication between modules. Modules are implemented as event handlers, registering interest in particular event patterns with the central OASYS server, and processing event instances as they occur, generally resulting in new publication events which other modules may respond to. In general, modules need have no knowledge of other modules – their interfaces can be defined solely in terms of the events they respond to and produce. However, OASYS does provide mechanisms for modules to communicate more directly: as well as "publish" events, OASYS supports "lifecycle" events, indicating when component modules start up or terminate, and "synthetic" events, containing arbitrary content negotiated between modules[4] An application is constructed and coordinated by the central server. Modules are "plumbed together" using one or more blackboards, then each is sent an "initialise" (lifecycle) event. The server then enters an event processing loop until all generated events have been dispatched, and then the application exits. The current implementation runs as a single Prolog process, although the architecture is potentially extendable to multi-process/multi-language scenarios.

### *5.2.2 Module Conception*

Individual modules are implemented as event handlers, that is, code to process each single event as it occurs. Initially, a module is registered to receive just lifecycle events. It is guaranteed to receive an "initialise" event before anything else and will typically handle this by registering interest in "publish" and "synthetic" events it is

[4] Synthetic events have no meaning to the OASYS server – it merely receives them and dispatches them to any module which has registered interest in them. They can be used by cooperating modules to pass messages which do not correspond to creation of new blackboard data or module lifecycle events.

interested in. In addition, each module has a name and a type which other modules can use to locate a particular component of a system, and private global state data which the OASYS server maintains between calls to the module handler.

There is no formal requirement that modules have any particular functionality or even that they do just one thing - in principle a single module could service multiple kinds of events. Module granularity and functionality is a matter for the user community of OASYS modules, not a requirement of OASYS itself.

Modules run asynchronously and independently: the only guarantee the system offers is that a module will receive events it is interested in only once, and in the same order as the events were generated. There is no guarantee that another module will or will not have run beforehand, nor that another module may or may not have processed exactly the same event. All such coordination is achieved by explicit cooperation between modules. For example, one way to implement a strict pipeline is using lifecycle events: module $N$ in the pipeline is activated by the "terminate" event of module $N-1$.

OASYS blackboards are strictly monotonically increasing: it is only possible to add information to a blackboard, and the only way to do so is via a "publish" event. Thus items in a blackboard are ground terms (so that they cannot be further instantiated indirectly, i.e. without explicit publication), and cannot be changed or removed once added. A module which wishes to "revise" a data structure must use additional meta-data protocols to indicate the relationship between the original and the revised version in the blackboard, which the other modules in the application must respect when accessing the data. The OASYS implementation provides specific support for this using the RAGS "Objects and Arrows" representation, which includes "revised-to" arrows to represent changes, but other approaches are also possible.

### *5.2.3 Central Control*

In an OASYS-based application, the central OASYS server module is firmly "in control": the application is built by registering modules (event handlers) with the server, and then control passes to the OASYS mainloop and stays there, typically until the application completes (i.e. no further events are generated). The server creates and manages blackboards; registers modules, and events they are interested in; generates lifecycle events; dispatches events to relevant module handlers and receives back new events to be processed; and handles error conditions. However, the support for module configuration and event dispatching is completely generic, declarative and essentially non-deterministic (although the actual Prolog implementation will follow its usual deterministic depth-first backtracking search). This allows for maximum flexibility over control at the application level, but with the inevitable overhead that modules which want to impose control constraints need to know more about each other and agree protocols for doing so. The OASYS implementation provides additional support functions to achieve this, such as a utility to construct a simple pipeline architecture, and libraries to interface correctly to "Objects and Arrows" data representations in the blackboard. In principal, the

OASYS server module could be replaced with a variant using a different control strategy, but the underlying assumption in OASYS is that control at this level is not important.

### *5.2.4 Services*

As with RAGSOCKS, the central server architecture readily supports centralised monitoring of system activity. Indeed it is possible to monitor event activity entirely transparently, by adding a module interested in every event that occurs. In addition the use of declarative blackboards makes it possible to consider off-line storage of system state, for example using an XML representation. This allows snapshots of system activity to be stored and examined, and the creation of complex test environments for individual modules.

However, because the central server is so neutral with regard to control issues, centralised *management* of processing is not possible without the explicit cooperation of modules, except at a very crude stop/go level. For example adding a module which seeks to pause another module when a particular event occurs may not succeed, as there is no way of knowing which of the two modules will run first.

### *5.2.5 Example System*

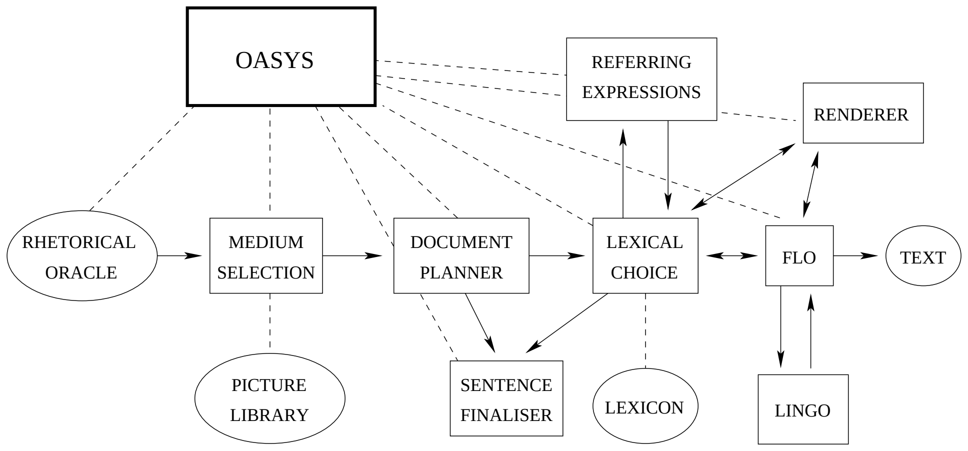

Fig. 2. The architecture of the RICHES system

OASYS has been used in the construction of the RICHES generator (Cahill *et al.* 2001a). The architecture of RICHES is shown in figure 2. Here the dashed lines indicate flow of information between the individual modules and the OASYS server, using a single shared blackboard. The solid arrows indicate approximately flow of control between modules. The following briefly describes the modules and their interaction – see (Cahill *et al.* 2001a) for further information.

**Rhetorical Oracle -** The input to the system, a rhetorical structure, is simply accessed from a data file and published to initialise the OASYS database.

**Media Selection -** As soon as the rhetorical representation becomes available, this module examines it and decides whether parts can be illustrated. If so, it creates document structure elements to link in the pictures.

**Document Planner -** The Document Planner, based on the ICONOCLAST text planner (Power 2000) takes the rhetorical representation, and produces a document structure. This module is pipelined after Media Selection, so it can take account of any picture elements already on the blackboard.

**Lexical Choice -** Lexical choice happens in two stages, both handled by a single logical module. In the first stage, lexical items are chosen for each predicate specified by the document structure. This fixes the basic syntactic structure of the proposition, and provides enough information for the Renderer and Sentence Planner to start processing. The second phase of lexical choice is interleaved with Referring Expression generation, lexicalising noun-phrases as they are generated. As each whole sentence is completed, the Lexical Choice module passes it on to FLO for final realisation.

**Referring Expressions -** The Referring Expression module adds information on the form of each noun phrase, deciding whether it should be a pronoun, a definite noun phrase or an indefinite noun phrase.

**Sentence Finaliser -** The Sentence Finaliser completes the high level sentential organisation combining the lexicalised predicates and noun phrases according to the specified rhetorical and document structure specifications.

**Finalise Lexical Output (FLO) -** FLO provides an interface between the RAGS representation on the blackboard and the external sentence realiser, LinGo (Carroll *et al.* 1999),

**Renderer -** The Renderer 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, asynchronously.

This example demonstrates a range of different control strategies — an initial three-stage pipeline is followed a complex set of interactions between modules building different parts of the final output document. It would be possible to impose more control structure on the architecture, for example the Renderer could in this example run after everything else has finished, but we note that (a) this is an additional constraint, not a simplification (b) more complex generators might require earlier rendering, for example in order to include document deixis (reference to page or section numbers etc.).

## 6 Discussion and Open Issues

RAGSOCKS and OASYS are just two of many possible generic implementation architectures for supporting the building of NLG systems. In this section, we compare them and reflect on wider issues that they raise.

### *6.1 Comparison of Prototypes*

Table 1 presents a comparison of RAGSOCKS and OASYS according to the features previously identified. For reference, the capabilities of XSLT pipelines have also been included in the table. In the table, “Y” indicates a feature supported, “N” a feature not supported and “S” a feature not directly supported but which can be simulated. The sequence XSLT pipelines – RAGSOCKS – OASYS is in order of increasing complexity, where each member of the sequence can in principle simulate the behaviour of its predecessors. Thus a choice between these architectures is really a decision about the tradeoff between ease of use (simplicity) and flexibility.

| | XSLT pipelines | RAGSOCKS | OASYS |
|---:|:---:|:---:|:---:|
| Non-determinism | Maybe | Y | Y |
| Non-monotonicity | Y | Y | S |
| Eclecticism | Y | Y | S |
| Role flexibility | N | Some | Y |
| Parallelism | N | Y | N |
| Central Controllability | N | N | Some |
| Modules with state | N | Y | Y |
| Complex modules | N | N | Y |

Table 1. *Comparison of Features Supported by Architectures*

Interestingly, although both RAGSOCKS and OASYS support non-determinism, neither architecture has used this significantly in the systems discussed. It remains to be seen how these will cope with large-scale non-determinism, and indeed this may require special support well beyond what these implementations currently offer.

For non-monotonicity and eclecticism, RAGSOCKS gives direct support. OASYS also supports these via its meta-data protocols and multiple blackboards, though these facilities make the architecture complex and lose something of the elegance of the basic model.

XSLT pipelines offer no flexibility in the roles of the modules – the exact sequence of operations must be specified in advance. In RAGSOCKS, a module will, however, be called (by giving it an input) when it is needed at runtime. This means that the number of times a module is called can vary, for instance. However, it has to be called via a request from another module and has to yield results as expected by that module. In OASYS, on the other hand, a module runs when *it* wants to and produces the kinds of results it decides. Although this only works well when there are some conventions about how material will appear in the blackboard, nevertheless it seems to represent the ultimate in role flexibility.

Parallelism is not supported by the current OASYS implementation, though there is nothing in the model to prevent an implementation using genuine parallelism (indeed, in some ways this would be more faithful to the underlying model than the current implementation, which imposes arbitrary serialisation on an essentially non-deterministic dispatching algorithm).

OASYS is the only one of these architectures that has significant central controllability. The range of possible control options is very large, though the modules definitely have to cooperate in any centrally-imposed control strategy. The manner of "programming" central control may be somewhat unfamiliar to many users, and in the current OASYS implementation, the assumption is that control at this level is not required.

### *6.2 What is a module?*

The three different architectures differ in the notion of "module" that they support and assume. In an XSLT pipeline, a module is a process that is called exactly once, has a single input and output and is responsible for acting over the whole of the text to be generated. In RAGSOCKS and OASYS, however, a module can have internal state and be invoked a variable number of times during generation. Thus a module can act to deal with just a local part of the generation process. In addition OASYS modules can implement multiple functions, with no formal requirement that they are related to each other at all. This allows OASYS to support applications where the functional and control modules do not correlate elegantly (cf. the discussion at the end of section 2 above). An OASYS module is responsible for knowing when it needs to be activated (which events to register interest in), whereas a RAGSOCKS module is responsible for producing an appropriate result when it is activated. XSLT modules have a responsibility to retain information from previous modules that will be needed later. RAGSOCKS modules have no such obligation, whereas OASYS modules have no choice but to leave what other modules have placed in the blackboard (monotonicity).

It follows from the above that the definitions of modules are likely to look very different in the different architectures. Indeed, the architecture adopted may have a significant impact on the way one sets about creating modules (i.e. what modules one thinks of and what their scope is). These observations suggest that there is no architecture that does not limit to some extent what one considers natural or possible, although in general, the more powerful architectures can simulate the less powerful ones with sufficient effort.

### *6.3 The Central Server*

Both RAGSOCKS and OASYS make use of central servers, and indeed this feature is probably essential for the kinds of services one would like to have provided (e.g. tracing, state saving, overall control). However, they take radically different views of the activeness of the server. The RAGSOCKS server is essentially a passive telephone exchange, whereas the OASYS server has key control over the data available to the modules (which means that it is constantly consulted) and can to some extent impose global control strategies.

A key decision in choosing an architecture seems to be how active the central server should be. Having an active server lessens the amount the programmer needs to worry about control within the modules. The modules then become more like declarative rules, whose operation is separately scheduled.

### *6.4 Roles of Data*

The OASYS use of a shared, monotonic, blackboard is very similar in spirit to the central database in GATE. Assuming that only a single blackboard is used, an OASYS module is then accessing *the* current description of the state of the generation process. XSLT pipelines take a similar view, although the current state is passed from module to module, rather than being shared. The RAGSOCKS approach, however, allows new information to be created and discarded on the fly. It takes a more anarchic view that data structures exist purely for the convenience of allowing pairs of modules to communicate (hence eclecticism) and that they need have no further significance.

In the end, the choice of an architecture then implies a view about how "organised" the NLG process is. Are the modules indeed cooperating on fleshing out the true picture (OASYS), or does generation emerge as a result of more complex interactions or even competition?

### *6.5 "Plug and play"*

A crucial role of a generic software architecture is to support the plugging in of alternative versions of a module. "Plug and play" will however only happen significantly if in practice different researchers end up defining compatible versions of a particular module, even when they perhaps disagree about other modules or about the order of execution of modules. To support this, the architecture must:

- have a sufficiently general notion of module that modules can be implemented independently of the exact context in which they are to be used,
- minimise the extent to which the programmer of a module needs to worry about context beyond the immediate focus of the module.

Whether this can be achieved depends to a large extent on the underlying data model, which we do not consider here. There may also be a tradeoff between these two goals – an architecture with a very general notion of module may be harder to program modules for.

Let us consider first the issue of module generality. An XSLT pipeline module is designed to be run at a particular point in a pipeline. It must be aware of all the information produced previously in order to preserve it or alter it as necessary for subsequent stages of the pipeline. Its definition may thus be very specific to its position in the pipeline, which means that someone else would have to be working with a near identical pipeline in order to come up with an alternative version of the module. A RAGSOCKS module, on the other hand, is defined in terms of the inputs it is given and the outputs it produces. It only needs to consider information that is part of these, which makes it independent of manipulations of data that are not its primary business. This should increase the chance that someone else may want to define a module with exactly the same specification. But how can one be sure that other modules will indeed package up exactly the right data (all at once) to provide its inputs and that other modules will exist that can make sense of the particular combinations of structures that are in its outputs? A module will only be reusable if other systems provide these contexts correctly. Finally, an OASYS module does not rely on its input all at once and can produce its output incrementally as its input appears. If written well, it can act effectively in many situations, regardless of exactly how or when the necessary information appears. There are some constraints on the context. For instance there should be no deadlocks possible in the blackboard and the scheduling of modules should be fair. These are, however, probably relatively frequently satisfied and cheaply checked.

To what extent does the module programmer have to worry about context in the different architectures? In an XSLT pipeline, a module definition must necessarily be concerned with the preservation/updating of the relevant context, and the programmer must deal with this explicitly. In RAGSOCKS, however, a module is only responsible for producing its specified output. Irrelevant contextual information can be excluded from its input or can simply be ignored by the module. Thus the module programmer hardly needs to be concerned about what else is going on in the generation process. The situation in OASYS is more variable. If a module produces output in batches, when complete inputs are available, then the situation is similar to that in RAGSOCKS. However, if the output is to be produced incrementally then the module programmer must explicitly cater for variations in the order in which the input may become available. This could potentially be complex. Thus there is a programming overhead associated with the extra generality that OASYS brings (if it is used).

The above discussion suggests that OASYS may be the most promising of the architectures discussed in terms of allowing for the reuse and comparison of modules in a way that is compatible with NLG practice. Paradoxically, however, OASYS is also the architecture that deviates most obviously from mainstream software engineering paradigms used in current NLG. And the extra generality over RAGSOCKS is bought at the expense of some programming and data management complexity. Just as RAGS offers a functional architecture for NLG but only through the effort of abstracting away from some current practice, OASYS may offer a similar tradeoff of implementation benefits if implementors make the effort to work in a more general setting.

## 7 Conclusions and further work

In this paper we have sought to make a distinction between functional and implementation aspects of generic software architecture proposals, in particular in the context of NLG. We have done so in part as an attempt to take the RAGS proposal for an NLG architecture one step closer to a complete software architecture, and in part as a way of thinking about software architectures more generally. The NLG scenario is challenging for any notion of a generic software architecture, because of the complexity of the task and the corresponding range of existing work to take into consideration. The partial functional characterisation of NLG offered by RAGS provides a foundation to think about implementation issues somewhat independently and we contend that this is useful: explicit separation of the functional and implementation structure of architectures is a valuable aid to developing and understanding complex tasks such as NLG. By comparing different implementation approaches we can gain a better understanding of the tradeoffs between simplicity and re-usability, and gain insight into the best ways to conceive of functional module definition and its relationship to implementation in general.

Future directions for this work can be considered at both the specific and general levels. The RAGSOCKS implementation for RAGS is available on the RAGS website, together with a range of support interfaces. Public release of OASYS is anticipated soon, together with additional support to promote take-up in the context of RAGS. Together, these resources will help take the RAGS initiative forward to a stage where a range of re-usable modules, created by a range of practioners, are available for the NLG community, and the fine details of a generic architecture become more evident. More generally, we believe that the overall framework we have described here has wider applicability to other NLP tasks, perhaps based broadly on the RAGS analysis, and more generally to other complex and 'intelligent' systems.

## 8 Acknowledgements

This work was partly supported by two research grants from the EPSRC (RAGS — *Reference Architecture for Generation Systems*: grant GR/L77102 to University of Brighton, grant GR/L77041 to University of Edinburgh). We also acknowledge the contribution of colleagues on the RAGS project: Donia Scott, Lynne Cahill, Mike Reape, Daniel Paiva, Christy Doran, Neil Tipper and Rodger Kibble, as well as the support and interest of other colleagues at Brighton, Edinburgh and Aberdeen.

## References

Barrutieta, G., J. Abaitua and J. Diaz. (2002) Cascading XSL Filters for Content Selection in Multilingual Document Generation. *Procs of the Second Workshop on NLP and XML*, Taipei.

Bateman, J. (1997) Enabling Technology for Multilingual Natural Language Generation: The KPML Development Environment. *Natural Language Engineering*, 3(1):15–55.

Bayer, S., C. Doran and B. George. (2001) Dialogue Interaction with the DARPA Communicator Infrastructure: The Development of Useful Software. Poster presentation, *HLT 2001.*

Beale, S., S. Nirenburg, E. Viegas and L. Wanner. (1998) De-Constraining Text Generation. *Procs of the Ninth International Workshop on Natural Language Generation*, Niagara-on-the-Lake.

Brew, C., D. McKelvie, R. Tobin, H. Thompson and A. Mikheev. (2000) The XML Library LT XML version 1.2. Obtainable at `http://www.ltg.ed.ac.uk/software/xml/xmldoc/xmldoc.html`.

Cahill, L. (1999) Lexicalisation in Applied NLG Systems. Technical Report ITRI-99-04, ITRI, University of Brighton. obtainable at `http://www.itri.brighton.ac.uk/rags/`.

Cahill, L. and M. Reape. (1999) Component tasks in applied NLG systems. Technical Report ITRI-99-05, ITRI, University of Brighton. obtainable at `http://www.itri.brighton.ac.uk/rags/`.

Cahill, L., J. Carroll, R. Evans, D. Paiva, R. Power, D. Scott, and K. van Deemter. (2001) From RAGS to RICHES: exploiting the potential of a flexible generation architecture. *Procs of the 39th Annual Meeting of the Association for Computational Linguistics (ACL-01)*, pages 98–105, Toulouse, France.

Cahill, L., R. Evans, C. Mellish, D. Paiva, M. Reape, and D. Scott. (2001) The RAGS Reference Manual . Technical Report ITRI-01-08, ITRI, University of Brighton. Available at `http://www.itri.brighton.ac.uk/rags`.

Carroll, J., A. Copestake, D. Flickinger and V. Poznanski. (1999) An efficient chart generator for (semi-)lexicalist grammars. *Procs of the 7th European Workshop on Natural Language Generation (EWNLG'99)*, pages 86–95, Toulouse.

Cheng, H., M. Poesio, R. Henschel and C. Mellish. (2001) Corpus-based NP Modifier Generation. *Procs of the Second Meeting of the North American Chapter of the Association for Computational Linguistics (NAACL-01)*, Pittsburgh.

Cunningham, H., Y. Wilks, and R. Gaizauskas. (1996) GATE - a general architecture for text engineering. *Procs of the 16th International Conference on Computational Linguistics (COLING'96)*, volume 2, pages 1057–1060, Copenhagen.

Danlos, L. (1984) Conceptual and Linguistic Decisions in Generation. *Procs of the 10th International Conference on Computational Linguistics (COLING'84)*, Stanford.

De Smedt, K. (1994) Parallelism in Incremental Sentence Generation. In G. Adriaens and U. Hahn (Eds) *Parallel Natural Language Processing*, Ablex, pages 421–447.

De Smedt, K., H. Horacek, and M. Zock. (1996) Architectures for Natural Language Generation: Problems and Perspectives. In G. Adorni and M. Zock, (Eds) *Trends in Natural Language Generation*, Springer Verlag, pages 17–46.

Elhadad, M. and J. Robin. (1992) Controlling content realization with functional unification grammars. In R. Dale, E. Hovy, D. Roesner and O. Stock (Eds) *Aspects of Automated Natural Language Generation*, Lecture Notes in Artificial Intelligence, 587. Springer-Verlag, pages 89–104.

Evans R., P. Piwek and L. Cahill. (2002) What is NLG? In *Procs of the Second International Conference on Natural Language Generation (INLG-02)*, New York, pages 144–151.

Grishman, R. (1995) TIPSTER Phase II Architecture Design Document Version 1.52. Technical Report, Dept of Computer Science, New York University.

Herzog, G., A. Ndiaye, S. Merten, H. Kirchmann, T. Becker and P. Poller. (2004) Large-Scale Software Integration for Spoken Language and Multimodal Dialog Systems. *Natural Language Engineering*, this issue.

Hirschman, L. and R. Gaizauskas. (2001) Natural Language Question Answering: the view from here. *Natural Language Engineering*, 7(4):275–300.

Hobbs, J. (1993) The generic information extraction system. *Procs of the fifth Message Understanding Conference (MUC-5)*, Morgan Kaufman.

Ide, N. and L. Romary. (2004) International Standard for a Linguistic Annotation Framework. *Natural Language Engineering*, this issue.

Langkilde, I. and K. Knight. (1998) Generation that Exploits Corpus-based Statistical Knowledge. *Procs of the Conference of the Association for Computational Linguistics (COLING/ACL-98)*.

Langkilde-Geary, I. (2002) An Empirical Verification of Coverage and Correctness for a General-Purpose Sentence Generator. *Procs of the Second International Conference on Natural Language Generation (INLG-02)*, New York, pages 17–24.

Laprun, C., J. Fiscus, J. Garafolo and S. Pajot. (2002) A Practical Introduction to ATLAS. *Procs of the Third International Conference on Language Resources and Evaluation (LREC-02)*, Las Palmas.

Lavoie, B. and O. Rambow. (1997) A fast and portable realiser for text generation. *Procs of the Fifth Conference on Applied Natural Language Processing (ANLP-97)*, Washington, pages 265-268.

McDonald, D. (1981) MUMBLE: A Flexible System for Language Production. *Procs of the Seventh International Joint Conference on Artificial Intelligence (IJCAI-81)*, Vancouver, page 1062.

Mellish, C., A. Knott, J. Oberlander and M. O'Donnell. (1998) Experiments using Stochastic Search for Text Planning. *Procs of the Ninth International Workshop on Natural Language Generation (INLG-98)*, Niagara-on-the-Lake, pages 98–107.

Mellish, C., R. Evans, L. Cahill, C. Doran, D. Paiva, M. Reape, D. Scott, and N. Tipper. (2000) A representation for complex and evolving data dependencies in generation. *Procs of the Language Technology Joint Conference, ANLP-NAACL2000*, Seattle.

Mellish, C., D. Scott, L. Cahill, R. Evans, D. Paiva and M. Reape. (2004) A Reference Architecture for Generation Systems. *Natural Language Engineering*, this issue.

Nirenburg, S., V. Lesser and E. Nyberg. (1989) Controlling a Language Generation Planner. *Procs of the 11th International Joint Conference on Artificial Intelligence (IJCAI-89)*, Detroit, pages 1524–1530.

O'Donnell, M., A. Knott, C. Mellish, and J. Oberlander. (2001) *ILEX*: The architecture of a dynamic hypertext generation system. *Natural Language Engineering*, 7:225–250.

Power, R. (2000) Planning texts by constraint satisfaction. *Procs of the 18th International Conference on Computational Linguistics (COLING-2000)*, Saabruecken, pages 642-648.

Radev, D., N. Kambhatla, Y. Ye, C. Wolf and Z. Wlodek. (1999) DSML: A Proposal for XML Standards for Messaging Between Components of a Natural Language Dialogue System. *Procs of the AISB'99 Workshop on Reference Architectures and Data Standards for NLP*, Edinburgh.

Reiter, E. (1994) Has a consensus NL generation architecture appeared and is it psycholinguistically plausible? *Procs of the 8th International Workshop on Natural Language Generation (INLG-94)*, Kennebunkport, pages 163–170.

Reiter, E. (2001) Pipelines and Size Constraints. *Computational Linguistics* 26:251-259.

Robin, J. and K. McKeown. (1996) Empirically designing and evaluating a new revision-based model for summary generation. *Artificial Intelligence*, 85(1-2).

Rubinoff, R. (1992) Integrating Text Planning and Linguistic Choice by Annotating Linguistic Structures. In R. Dale, E. Hovy, D. Roesner and O. Stock (Eds) *Aspects of Automated Natural Language Generation*, Springer Verlag, pages 45–56.

Seki, Y. (2001) XML Transformation-based three-stage pipelined Natural Language Generation System. *Procs of the Sixth Natural Language Processing Pacific Rim Symposium (NLPRS 2001)*, Tokyo, pages 767–768.

Text Encoding Initiative. See `http://www.tei-c.org/`.

Wahlster, W. (2001) Robust Translation of Spontaneous Speech: A Multi-Engine Approach. *Procs of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-2001)*, Seattle.

Wanner, L. and E. Hovy. (1996) The HealthDoc Sentence Planner *Procs of the Eighth International Natural Language Generation Workshop (INLG-96)*, Herstmonceux, pages 1–10.

Wilcock, G. (2001) Pipelines, Templates and Transformations: XML for Natural Language Generation. *Procs of the first NLP and XML Workshop*, Tokyo, pages 1–8.