Agent-based Solutions for Natural Language Generation Tasks

Raquel Hervás and Pablo Gervás

Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid, Spain
raquelhb@fdi.ucm.es,pgervas@sip.ucm.es

Abstract. When building natural language generation applications it is desireable to have the possibility of assembling modules that use different techniques for each one of the specific generation tasks. This paper presents an agent-based module for referring expression generation and aggregation, implemented within the framework of a generic architecture for implementing multi-agent systems: Open Agent Architecture.

1 Introduction

Natural language generation (NLG) is subdivided into several specific subtasks [1], and each one of them operates at a different level of linguistic representation - discourse, semantics, lexicon, syntax... Natural language generation can be applied to domains where the communication goals and the characteristics of the texts to be generated can be very different, from the transcription into natural language of numerical contents [2] to the generation of literary texts [3].

Each type of NLG application may need a different way of organising the system into modules [4]. Even given a specific organization (or *architecture*) of the system, different types of applications may require different solutions for each of the specific tasks involved in the generation process. In this context, a Multi-Agent System (MAS) implementation [5] may be highly benefitial. Each agent may be assigned a specific task to solve, and agents may negotiate between to reach a final solution, with no need to define an explicit architecture beyond the society of agents. Defining the architecture of the system would be limited to establishing the communication interfaces between the agents.

To put this idea to the test, two subtasks of the process of automatic generation of language texts have been selected: referring expresion generation and aggregation.

2 Referring Expression Generation and Aggregation

The appropriate use of referring expresions to compete with human-generated texts involves a certain difficulty. Simple algorithms to decide when to use a pronoun or a full noun produce very poor results. According to Reiter and Dale

[6], a referring expresion must communicate enough information to identify univocally the intendent referent within the context of the current discourse, but always avoiding unnecessary or redundant modifiers. Reiter and Dale propose an algorithm for generating noun phrases to identify objects in the sphere of attention of the reader. Kibble and Power [7] propose a system for planning coherent texts and choosing the referring expresions. They claim that textual and syntactic planning mut be partially directed by the goal of maintaining referential continuity, increasing the opportunities for the unambiguous use of pronouns. Palomar et al. [8] also present an algorithm to identify noun phrases which are intended referents for personal, demostrative and reflexive pronouns or which have been ommitted in Spanish texts. They define a list of restrictions and preferences for the different types of pronominal expresions and they stress the importance of each type of knowledge - lexical, morphological, syntactic, and statistical - in the resolutoin fo anaphoric references.

Aggregation is the task of deciding how to compact the representation of the information in a given text. However, there is no accepted definition in the literature of what it is [9]. This task operates at several different linguistic levels, but in this paper we are only considering its application to concepts and their attributes. For instance, the system must decide whether to generate *"The princess is blonde. She sleeps."* or to generate *"The blonde princess sleeps."*. One should take care not to produce texts with too many adjectives when the information to be processed is dense in terms of attributes, as in *"The pretty blonde princess lived in a strong fancy castle with her stern rich parents."* Aggregation is generally desireable, but a balance must be found between conciseness and an acceptable style.

3 Open Agent Architecture

The Open Agent Architecture (OAA) [10] is a framework for developing multiagent systems intended to enable more flexible interactions among a dynamic community of heterogeneous software agents. The operation of the architecture is based on the idea of delegation: agents do not hard-code their interactions (method calls or messages) in a way that fixed how and whom they will interact with, instead the interactions between OAA agents are expressed in terms of needs delegated to a Facilitator agent. This Facilitator agent coordinates the agent community so that it can achieve its task. It does this by providing services such as parallelism, failure handling, and conflict detection, which relieves each client agent from having to worry about these issues itself. OAA's Distributed Agents are simply programs - or wrappers around programs - that share several common functionalities, and which are possibly distributed across various machines.

In OAA, control of how interaction and communication occurs among agents arises from cooperation between 4 distinct knowledge sources:

– the requester which specifies a goal to the Facilitator,

- providers who know what services they can provide and register their capabilities with the Facilitator,
- the Facilitator which receives requests from requesters, maintains a list of available provider agents, and has a set of general strategies for meeting goals, and
- meta-agents that contain domain- or goal-specific knowledge and strategies which are used as an aid by the Facilitator.

These knowledge sourcers interact in the following way to make cooperation possible among a set of OAA agents. The Facilitator matches a request to an agent or agents providing that service, delegates the task to them, coordinates their efforts, and delivers the results to the requester. This manner of cooperation among agents is suitable both for straightforward and compound, multi-step tasks. In addition to delegation, OAA also provides the ability to make direct calls to a specific agent, and to broadcast requests.

The OAA's Interagent Communication Language (ICL) is the interface and communication language shared by all agents, no matter what machine they are running on or what computer language they are programmed in. The ICL has been designed as an extension of the Prolog programming language, to take advantage of unification and other features of Prolog. A number of key declarations and other program elements are specified using ICL expressions. These include declarations of capabilities, events (communications between agents), requests for services, responses to requests, trigger specifications, and shared data elements.

4 Multiagent Module for Referring Expression Generation and Aggregation

The work presented here is a multi-agent module for the tasks of Referring Expression Generation and Aggregation, implemented within the OAA architecture. Each agent is responsible for a specific subtask, and the conjunction of all the subtasks gives rise to a full implementation of the required tasks. This module differs from a blackboard architecture in the ability to negotiate that OAA agents have. When an agent requests an action from the rest of the system, the system may gather the solutions offered by different agents and use the one that best fits its purpose, or a combination of various solutions. The agents in OAA negotiate between them, rather than simply sharing data.

The multiagent system is made up of five specific agents and one auxiliary agent, coordinated by the Facilitator provided by the OAA architecture. A brief description of these agents and their functionality follows:

- **FileAgent.** Auxiliary agent capable of reading and writing text files.
- **ReferringAgent.** Main agent for the the two tasks addressed by the module. It deals with interpreting the input data and ensuring they are available for the rest of the agents.

– **PronounAgent.** Agent in charge of deciding if the reference to a given concept is made using its full name or only a pronoun.
– **KnowledgeAgent.** Agent that enriches some of the concepts with the attributes associated to them in the knowledge base.
– **AggregateAgent.** Agent that aggregates some of the descriptions of concepts in the text - in term of attributes - with the occurrences of the concepts in the text.
– **DisaggregateAgent.** Agent that performs the opposite task: it separates a description of a concept in terms of its attributes from a given occurrence of the concept in the text.

4.1 Data Representation

The notation described here corresponds to the internal representation within the multiagent system. Messages to be communicated in the final text are organised into paragraphs, and they are represented in the ICL language of the OAA architecture.

Characters, locations, and attributes are represented as simple facts that contain an identifier - which distinguishes each character or location from the others - and its name.

```
character(ch26,princess)
location(l14,castle)
attribute(ch26,blonde)
```

The identifier occurring in attribute facts corresponds to the identifier of the concept that has that attribute, and the name corresponds to the attribute itself.

The current prototype operates over simple linguistic constructions: the description of a concept using an attribute or the relation between two concepts.

```
[character(ch26,princess),
    isa(),
    attribute(ch26,pretty)]

[character(ch26,princess),
    relation(ch26,l14,live),
    location(l14,castle)]
```

Pronominal references are indicated by adding a ‘pron’ element to the corresponding fact, as in

```
character(ch26,princess,pron)
```

Finally, concepts may be accompanied by attributes that will precede the name of the concept, as in “the pretty blonde princess”. This list of attributes is added to the corresponding concept.

```
character(ch26,princess,
          [attribute(ch26,pretty),
          attribute(ch26,blonde)])
```

4.2 Specific Funcionality of the Agents

Each one of the agents that make up the module is responsible for a very specific task. Thanks to the flexibility of multi-agent systems, not all agents will be required every time to act on a text draft. Each agent carries out different modifications over the text draft, and only some of them may be desireable at a given stage.

ReferringAgent The ReferringAgent interprets the input data obtained from a file and makes them available for the rest of the agents to work on them. It sends a request to the agent system for reading from a text file the input to the referring expresion generation module. The agent reads this string, transforms it into the internal representation of the agent system, and makes it public to the rest of the agents by specifying to the Facilitator that it has those data and that any other agent may read them or modify them. Another task available from this agent is that of saving the processed information to file. An example of input text is:

[character(ch26,princess);relation(ch26,l14,live);location(l14,castle)]
[character(ch26,princess);isa();attribute(ch26,pretty)]

For this text, the following facts are generated as messages:

```
message(0,0,[character(ch26,princess),
             relation(ch26,l14,live),
             location(l14,castle)])
message(0,1,[character(ch26,princess),
             isa(),
             attribute(ch26,pretty)])
```

Messages are numbered indicating to which paragraph they belong, and their position with respect to other messages within that paragraph. The rest of the agents will eliminate or add messages, and it is necessary to retain the order between them for the generated text to keep the desired structure.

PronounAgent The Pronoun Agent decides whether the reference to a given concept is carried out using its full name, for instance *"the princess"*, or the corresponding pronoun, in this case, *"she"*. The algorithm employed by the agent is based on two ideas. When writing a text one cannot use a pronoun to refer to something that has not been mentioned before, or readers will be confused. An example might be:

<u>She</u> lived in a castle. <u>A princess</u> was the daughter of parents.

Also, if the full name reference and the pronominal reference are too far apart, the reader will be confused and he will be unable to relate the two occurrences of the same concept. An example is given in the following text:

A princess lived in a castle. She was the daughter of parents. She loved a knight. She was pretty. She was blonde. They lived in it.

The heuristic used by the agent relies on using a pronoun if the concept in question is one of the last two concepts mentioned by their full name. A possible result would be:

```
message(0,0,[character(ch26,princess),
             relation(ch26,l14,live),
             location(l14,castle)])
message(0,1,[character(ch26,princess,pron),
             relation(ch26,ch25,love),
             character(ch25,knight)])
message(0,2,[character(ch26,princess),
             isa(),
             attribute(ch26,pretty)])
```

KnowledgeAgent The Knowledge Agent enriches some of the concepts using the attributes that they have associated in the knowledge base. This agent has an associated probability value, and it decides whether to enrich a concept or not based on that probability. This value must be chosen with care. If it is too small, the text will have very few adjectives, and if it is too high the text will have redundant mentions of certain attributes.

An example of the operation of this agent is given below. In this case, attributes have been added to the reference to the castle, but not to the reference to the princess.

```
message(0,0,[character(ch26,princess),
             relation(ch26,l14,live),
             location(l14,castle,[attribute(strong)])])
```

AggregateAgent The Aggregate Agent aggregates some of the descriptions in the text to other occurrences of the concepts that they correpond to. In order to achieve this, it searches in each paragraph for messages of the type “X is Y”, and it aggregates the attribute appearing in the message with an earlier occurrence of the given concept in the same paragraph. This is done in some cases and not in others according to a probability associated with the agent. If the aggregation is performed, the message containing the description is eliminated from the paragraph.

An example of the results that this agent may generate from the initial messages read is given below. The probability used in this case is 0.5.

```
message(0,0,[character(ch26,princess,[attribute(pretty)]),
             relation(ch26,l14,live),
             location(l14,castle)])
```

DisaggregateAgent The Disaggregate Agent carries out a task complementary to that of the `AggregateAgent`. It disaggregates some of the attributes of a concept and it inserts into the text a separate message describing them of the type "X is Y". In each paragraph it checks all the concepts that have an associated list of attributes, and - again according to a predetermined probability - it decides to disaggregate some of them. If it decides to disaggregate a given attribute, it eliminates it from the list of attributes associated with the concept and it adds a descriptive message following the message where the concept occurred.

An example of the operation of this agent - with a probability of 0.5 - and starting from the example presented in the description on the `KnowledgeAgent`, would be:

```
message(0,0,[character(ch26,princess),
             relation(ch26,l14,live),
             location(l14,castle)])
message(0,1,[location(l14,castle),
             isa(),
             attribute(l14,strong)])
```

4.3 Experiments and Results

The OAA architecture provides mechanisms for monitoring and refining the implemented agents. Each agent can be manually switched on or off. These mechanisms have been used to perform some experiments over the implemented multi-agent system.

Although the OAA architecture is intended to make the agents independent from one another, there are some restrictions that must be taken into account. The first agent that should be called is the `ReferringAgent`, for without the input data the rest of the agents can do nothing. However, as this agent sends a request to the system for reading or writing a file, the `FileAgent` must have been called beforehand, so that it can reply to the requests presented by the `ReferringAgent`.

The other four agents of the system can be called in any order, and even called repeatedly. Given the probabilistic factors involved, and the fact that messages are added or eliminated in some cases, the result of system operation will vary depending on the order in which the agents are called. A clear example is the possible interaction between the `DisaggregateAgent` and the `KnowledgeAgent`. If the first one is called before the second one, very few disgregations will be performed on the text, since the concepts initially have no associated list of attributes. On the other hand, it the `KnowledgeAgent` has already been called, there will be more concepts with associated attributes, and the action of the `DisaggregateAgent` will be more visible.

Another issue to be taken into account is that of pronouns. Each of the agents operates over the concepts without considering whether they will be finally realised as pronominal references, so in theory messages such as *"Pretty* *she loved a knight"* are possible. Since this sort of sentence is ungrammatical, in such situations the system will ignore any attributes associated with the concept described by a pronoun. This gives rise to correct sentences such as *"She loved a knight"*.

5 Discussion

The results obtained from the operation of the multi-agent system have been compared with those of an existing application, ProtoPropp [11], and it evolutionary version EvoProtoPropp [12]. ProtoPropp is a system for the automatic generation of stories that reuses a case base of previous stories to produce a new one that matches the user's requests. The system operates over a knowledge base organised as a taxonomy. This knowledge base or ontology includes the characteristics of the concepts and the relations that exist between them. The data structure that the system outputs is plot plan, in which a skeleton of the plot is described in terms of the elements of the ontology that the system handles - characters of the story, locations for the action, events that take place... From this, the textual representation of the story is obtained.

In the generation module of ProtoPropp the referring expresions to be used for each concept are determined using a very simple heuristic: the first time that a concept appears in a paragraph, the generator uses its full name, and for all subsequent occurrences in that paragraph it uses a pronoun. The problem with this method is that two appearances of the same concept may be quite far apart within the same paragraph, so the reader is confused when reading the text. The generation module of ProtoPropp does not aggregate concepts and attributes. A fragment of text generated by ProtoPropp is the following:

> *A princess lived in a castle. She loved a knight. She was pretty. She was blonde. It had towers. It was strong.*

EvoProtoPropp is an implementation of ProtoPropp in which the tasks of referring expresion generation and aggregation are carried out using Evolutionary Algorithms. The same fragment given above, if generated by EvoProtoPropp would come out as:

> *A pretty princess lived in a strong castle. She loved a brave knight. The princess was blonde. The castle had towers.*

Using the multi-agent system described in this paper, that same fragment of text is generated as:

> *A princess lived in a strong castle. She loved a brave knight. The princess was pretty. She was blonde. The castle had towers.*

Both EvoProtoPropp and the multi-agent system improve the results of ProtoPropp, resulting in a better use of adjectives and more referential coherence of the final text.

A possible further improvement would be to introduce an additional agent responsible for checking that the text fulfills a set of restrictions before accepting it as valid. The current implementation may result in redundant uses of attributes, like in *"The pretty princess was pretty"*. Problems of coherence may also arise in cases where two occurrences of the same concept appear associated with different attributes, as in *"The pretty princess lived in a castle. The blonde princess loved a knight"*. This situation might erroneously suggest that there are two different princesses in the generated story, even if the underlying data imply a pretty blonde princess.

Another issues to be tested is the concurrent operation of the agents. The current implementation relies on the agents being called in any order, but no two at the same time. The OAA architecture allows the definition of triggers over the data, so that an agent may learn when data over which it has already worked are modified by another agent. This could lead that agent to revise the data in order to apply its functionality again.

6 Conclusions and Future Work

The first experiments concerning the optimization of Referring Expresion Generation and Aggregation tasks have given promising results. Each agent is responsible for very specific issues within these tasks. This results in a very flexible solution, so that users may call all or any of these tasks separately with great ease.

The positive results of this experiment opens the doors to considering the application of similar solutions to other aspects of NLG, such as the organisation of a generation system in modules, or the definition of its control flow. There are many ways of organising a natural language generation system, and their advantages and disadvantages are still subject to discusion [4, 13]. One can find singnificantly different architectures according to their division into modules or the topology of the connections between them. In [4] several of these architectures are discussed in detail, and all of them show advantages and disadvantages. The developer of an NLG application must consider a wide range of architectural solutions, for each one of them may be relevant for some particular aspect of his problem.

The cFROGS architecture [14] aims to provide a possible NLG application developer with the necessary infrastructure to facilitate his task as much as possible. This is achieved by providing generic architectural configurations for the most commonly used configurations used in this type of system. cFROGS identifies three basic design decisions when building a generation system: the set of modules to use, the flow of control that handles those modules, and the data structures that pass from one module to another.

The idea of a multi-agent system may be adapted to an architecture such as cFROGS from two different points of view. On one hand, the agents may be used as wrappers for modules of an NLG system implemented in cFROGS, so that the flow of control that governs them is the OAA architecture itself. The

NLG systems implemented according to this structure would have similarities with a blackboard architecture. On the other hand, a single module within a cFROGS architecture might be a wrapper for an OAA architecture, responsible for starting both the Facilitator and the rest of the agents as necessary.

References

1. Reiter, E., Dale, R.: Building Natural Language Generation Systems. Cambridge University Press (2000)
2. Goldberg, E., Driedger, N., Kittredge, R.: Using natural-language processing to produce weather forecasts. IEEE Expert: Intelligent Systems and Their Applications **9** (1994) 45–53
3. Callaway, C., Lester, J.: Narrative prose generation. In: Proceedings of the 17th IJCAI, Seattle, WA (2001) 1241–1248
4. DeSmedt, K., Horacek, H., Zock, M.: Architectures for natural language generation: Problems and perspectives. In Ardoni, G., Zock, M., eds.: Trends in natural language generation: an artificial intelligence perspective. LNAI 1036. Springer Verlag (1995) 17–46
5. Ferber, J.: Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)
6. Reiter, E., Dale, R.: A fast algorithm for the generation of referring expressions. In: Proceedings of the 14th conference on Computational linguistics, Nantes, France (1992)
7. Kibble, R., Power, R.: An integrated framework for text planning and pronominalization. In: Proc. of the International Conference on Natural Language Generation (INLG), Israel (2000)
8. Palomar, M., Ferrández, A., Moreno, L., Martínez-Barco, P., Peral, J., Saiz-Noeda, M., Muñoz, R.: An algorithm for anaphora resolution in spanish text. Computational Linguistics **27** (2001) 545–567
9. Reape, M., Mellish, C.: Just what is aggregation anyway? In: Proceedings of the 7th EWNLG, Toulouse, France (1999)
10. Martin, D.L., Cheyer, A.J., Moran, D.B.: The open agent architecture: A framework for building distributed software systems. Applied Artificial Intelligence **13** (1999) 91–128 OAA.
11. Gervás, P., Díaz-Agudo, B., Peinado, F., Hervás, R.: Story plot generation based on CBR. In Macintosh, A., Ellis, R., Allen, T., eds.: 12th Conference on Applications and Innovations in Intelligent Systems, Cambridge, UK, Springer, WICS series (2004)
12. Hervás, R., Gervás, P.: Uso flexible de soluciones evolutivas para tareas de generación de lenguaje natural. Procesamiento de Lenguaje Natural **35** (2005) 187–194
13. Reiter, E.: Has a consensus NL generation architecture appeared, and is it psychologically plausible? In McDonald, D., Meteer, M., eds.: Proceedings of the 7th. IWNLG '94, Kennebunkport, Maine (1994) 163–170
14. García, C., Hervás, R., Gervás, P.: Una arquitectura software para el desarrollo de aplicaciones de generación de lenguaje natural. Procesamiento de Lenguaje Natural **33** (2004) 111–118