Self-Comprehension for More Coherent Language Generation*

George A. Wright[1][0000−0002−2036−7737] and Matthew Purver[1,2][0000−0003−2297−1273]

[1] School of Electronic Engineering and Computer Science, Queen Mary University of London, United Kingdom
george.a.wright@qmul.ac.uk
[2] Department of Knowledge Technologies, Jožef Stefan Institute, Ljubljana, Slovenia
m.purver@qmul.ac.uk

Abstract. Linguoplotter is a distributed and chaotic architecture where an entanglement of different processes interact to generate a text describing a raw data input. This paper describes recent additions to the architecture whereby a greater degree of language comprehension is used to improve the coherence of generated text. Some examples of the architecture operating are considered, including where it performs well and generates a good quality text; and instances where it gets trapped in loops that either prevent an output from being generated or cause a lower quality output to be produced before there is a chance to find a better alternative. Finally, ideas from the program Metacat are considered which could allow the program to observe its own processes and become a more human-like intelligence.

Keywords: NLG · NLU · Distributed Architecture.

1 Introduction

Evidence from linguistics, psychology, and neuroscience shows that language production and comprehension are intertwined with and influence one another in humans [5]. Moreover, when people write, they re-read what they have written and may make adjustments or stop and consider what to write next in a *cycle of engagement and reflection* [7]. It makes sense, not only that people do interweave comprehension with production, but that they *need* to do so, for it is important to check that what one says or writes can be understood by the intended recipient. This will be just as true for an artificial person.

The architecture of Linguoplotter allows such an intermingling of different processes including: the perception of an input; the generation of text that describes it; the comprehension and evaluation of text; and the decision to

* Partially supported by the UK EPSRC under grants EP/R513106/1 (Wright), EP/S033564/1 and EP/W001632/1 (Purver); the Slovenian Research Agency via research core funding for the programme Knowledge Technologies (P2-0103) and the projects CANDAS (J6-2581) and SOVRAG (J5-3102).

output a text. These are not separate processes running in parallel, but entangled processes that inform one another: just as humans use the projection of narrative frames to understand new situations [8], LINGUOPLOTTER's narrative frames guide how it conceptualizes its input. Furthermore, recent changes to the architecture increase the degree to which language comprehension affects its generation of text, with the recognition of patterns in text influencing how it combines and arranges sentences into a more cohesive whole.

This paper discusses these recent changes and also considers how a greater capacity for introspection in the architecture could both give it a more human-like intelligence and help overcome problems that it faces when generating text.

2 A Simple Problem Domain

LINGUOPLOTTER is developed and tested with toy examples from a simple domain: sequences of maps showing temperatures at different times on a fictional island (cf figure 2). This domain is ostensibly very plain and simple, but human-written examples describing the same maps demonstrate a variety of phenomena including conceptual metaphor (*a spike in warmer temperatures* — 1c); anthropomorphism (*everywhere will enjoy temperatures in the 20s* — 1b; connections drawn between the end and the beginning (*falling back on Sunday* – 1b); and self-referential text (*it's a tale of two halves* — 1c); as well as more mundane, matter-of-fact language (*cool throughout the weekend* — 1a).

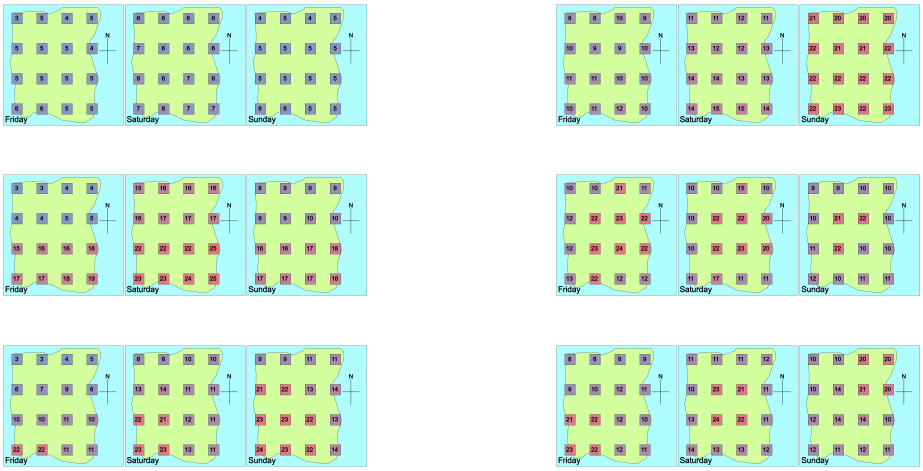

Fig. 2: Some sequences of maps used to test the architecture.

The architecture should be seen, not as an "expert system", but as an early prototype which could in future be applied more widely. Little knowledge engi-

neering is required in this domain, thus the focus is on the fundamental processes involved in perception and language, not on domain-specific details.

3 The Architecture

The Architecture of LINGUOPLOTTER borrows much from the architecture of COPYCAT [4] and related models of analogy-making [2], in which micro-agents called *codelets* run stochastically, making incremental changes to structures in a shared workspace. In LINGUOPLOTTER, codelets selected from the *coderack* make incremental changes to networks across many spaces in the *bubble chamber*.

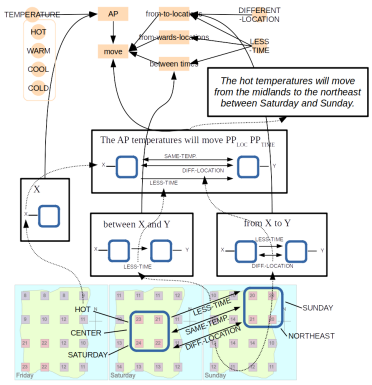

Fig. 3: Some of the structures involved in describing an input. Orange nodes at the top of the diagram belong to the concept network. Solid arrows connecting them are links that spread activation. Each box is either a frame or an output space. Dotted lines connecting items in different spaces are correspondences between items. Not all structures are shown in full detail for sake of clarity.

3.1 The Bubble Chamber

As the program runs, structures are incrementally added to the bubble chamber which identify patterns in the input; match them with abstract representations in long-term conceptual knowledge; and generate a text which describes the input. The structures built in the bubble chamber are based on the *simplex networks* of Fauconnier and Turner [1]. A simplex network matches elements in an unstructured input space with slots in a structured frame and has an output space containing a blend of content from the input space and structure from the frame. In LINGUOPLOTTER, frames provide the syntax and morphology of sentences describing the input. Figure 3 illustrates an example where frames are matched to part of the input to generate a sentence that describes it. Each simplex network is contained inside a *view*, which must only contain non-contradictory structures.

Each structure, be it a *chunk* grouping together similar nodes, a *label* or *relation* assigning a property to a chunk, or a *correspondence* matching a structure in one space to a structure in another is first suggested by a suggester codelet and then built by a builder. Evaluators assign a quality score to a structure based on how representative it is of a concept and how much it contributes to the network around it. Selectors choose stochastically between competing structures.

3.2 The Worldview and Focus

As sentences and longer pieces of text are built they can be promoted into the worldview. The text in the worldview is a candidate for publication. Worldview setting codelets are responsible for deciding between candidate texts and publisher codelets decide whether or not to output a text.

The focus is a temporary sub-goal selected from among these simplex networks that encourages activity towards a single network, so that processing can at times be more direct and less broad. Focus setting and un-setting codelets determine which simplex network should be in focus and view-driven factory codelets spawn codelets that suggest structures to fill in the network.

3.3 The Coderack

Codelets are chosen from the coderack, a stochastic priority queue where a codelet's urgency determines the likelihood it is selected. Once a codelet has run, it spawns a follow-up codelet which replaces it on the coderack. This results in self-sustaining streams of codelets that continue performing operations in the bubble chamber until a text is output.

3.4 Satisfaction and Randomness

The architecture is stochastic and distributed with no centralized decision maker: competitions between alternative structures in the bubble chamber and codelets on the coderack decide which texts are promoted into the worldview and published. There is a degree of randomness in codelet and structure selection which is determined by the program's satisfaction score. This score is based on the *temperature* mechanism of Copycat but an alternative name is used to avoid confusion with the temperatures on the maps the program describes.

When satisfaction is high (there are good quality and coherent structures matching the input to a piece of text) the program becomes more deterministic so that existing simplex networks can be completed and the resulting text output. When satisfaction is low, the program becomes more random so that a wider range of possibilities can be explored.

While there is a certain degree of randomness when it comes to micro-level selection between individual structures or codelets, the program tends to converge upon a narrow range of macro-level behaviours and textual outputs. The satisfaction score S is calculated:

$$S = max(G, F) \tag{1}$$
$$G = aI + bV + cW \tag{2}$$
$$W = dC_1 + eC_2 + fC_3 \tag{3}$$

- G: a measure of the general quality of all structures in the bubble chamber.
- F: the quality of the view in focus.
- I: the quality of the structures are built on the raw input.
- V: the average quality of all views in the bubble chamber.
- W: the quality of the worldview.
- C_1, correctness: the quality of input structures in the worldview.
- C_2, completeness: the proportion of the raw input described in the worldview.
- C_3, cohesiveness: the quality of relations connecting worldview sentences.

The qualities of individual structures within the spaces and the quality of views is calculated by individual evaluator codelets. Full details are available in the Python implementation of LINGUOPLOTTER available on GitHub[3].

The coefficients a to f are real numbers. The outputs discussed in this paper were generated using the values ($a = 0.4$, $b = 0.2$, $c = 0.4$, $d = 0.3$, $e = 0.2$, $f = 0.5$). A discussion of tests using different weights and the effect they have on the program's behaviour is provided in Wright and Purver [9].

4 Pattern Recognition on Many Levels

Earlier versions of this program [9] have focused on describing states (single maps) and events on sequences of maps. This latest version of the architecture attempts to recognize patterns between events and sentences so that it can build more coherent narratives describing a larger portion of the input.

This involves a greater deal of self-comprehension than earlier versions of the program, which only built and evaluated structures and sentences representing patterns discovered on the input maps. The latest version of LINGUOPLOTTER also uses its frames to recognize patterns between sentences so that a cohesive text can be built out of them. These new frames recognize patterns such as parallelism, an ordering along a particular dimension, or disanalogy. These frames serve to classify pairs of sentences so that they can be written in an appropriate order and connected with a relevant conjunction. For example:

- *Temperatures will be cold in the country between friday and saturday* ***then*** *temperatures will be cool in the country between saturday and sunday.* (Ordering in time).
- *Temperatures will increase in the south between friday and saturday* ***and*** *temperatures will increase in the north between friday and saturday.* (Parallel times).

[3] https://github.com/georgeawright/linguoplotter

– *Temperatures will be cool in the country between saturday and sunday* ***but*** *temperatures will be cold in the country between friday and saturday.* (Disanalogy — same verb describing different temperatures).

In recognizing patterns between sentences in order to further develop them as texts, LINGUOPLOTTER intertwines more fully the processes of language production and language comprehension.

5 The program's behaviour

Tables 1 and 2 show the range of outputs that the program generates when run multiple times with the sequences in figures 1a and 1c. The tables show the average satisfaction score for each text, the average time taken to generate the text (in codelets run) and the frequency with which it produces that text. The program's symbolic nature allows us to look inside and understand how these texts were generated and why it sometimes fails to produce a good output.

Text	Satisf	Time	Freq
Temperatures will be cold in the country between friday and saturday **then** temperatures will be cool in the country between saturday and sunday.	0.842	7141	4
Temperatures will be cool in the country between saturday and sunday **but** temperatures will be cold in the country between friday and saturday.	0.839	6950	2
Temperatures will be cold in the country between friday and saturday **then** temperatures will be cold in the country between saturday and sunday.	0.786	4833	1
Temperatures will be cold in the country between friday and saturday **and** temperatures will be cool in the country between saturday and sunday.	0.691	8209	2
Temperatures will be cool in the country between saturday and sunday **and** temperatures will be cold in the country between friday and saturday.	0.578	6381	3
Temperatures will be bad in the country between friday and saturday **and** temperatures will be cool in the country between saturday and sunday.	0.537	9821	1
Temperatures will be cool in the country between saturday and sunday.	0.4	6483	14
Temperatures will be cold in the country between friday and saturday.	0.4	5827	22
Temperatures will be cold in the country between saturday and sunday.	0.304	1603	1

Table 1: Outputs for sequence 1a. Conjunctions in bold for clarity.

Text	Satisf	Time	Freq
Temperatures will increase in the north between friday and saturday **and** temperatures will decrease in the south between saturday and sunday.	0.709	11726	2
Temperatures will increase in the south between friday and saturday **and** temperatures will decrease in the south between saturday and sunday.	0.684	14331	1
Temperatures will increase in the south between friday and saturday **and** temperatures will increase in the north between friday and saturday.	0.644	17751	1
Temperatures will increase in the north between friday and saturday.	0.35	10737	4
The warm temperatures will move from the south northwards between friday and saturday.	0.35	12952	3
The warm temperatures will move from the north southwards between saturday and sunday.	0.347	11163	1
Temperatures will decrease in the north between saturday and sunday.	0.26	9712	2
Temperatures will decrease in the south between saturday and sunday.	0.233	9387	3
None	0.109	20000	33

Table 2: Outputs for sequence 1c. Conjunctions in bold for clarity.

5.1 An example of the program running

This is a sample of the events that took place inside LINGUOPLOTTER[4] when it was given the input from figure 1a and the random seed 0. Numbers indicate the time measured by the number of codelets run.

0-400 Processing is dominated by chunk building on the input resulting in 3 chunks, each covering the entire island at a different point in time. Similar temperatures across the island allow high quality chunks of that size.

124 The first label-builder codelet runs attaching the label SUNDAY to a small chunk. Labels built at this early stage are attached to chunks that will eventually be superseded and removed from the bubble chamber, but their construction leads to the activation of relevant concepts.

752-784 An adjectival phrase frame is set as focus and as its slots are filled in, the program experiences one of its first spikes in satisfaction.

1000-2000 Simplex networks with ADJECTIVAL, IN-LOCATION, and BETWEEN-TIMES frames are completed and those frames become fully active. They spread activation to frames, for which they can be a component such as BE.

2658 A worldview setter sets a recently completed BE sentence as worldview. Now that the worldview has been set, satisfaction is permanently higher.

2881 A view suggester runs. Since the BE frame has a high activation and only one instance, it suggests another simplex network based on the BE frame.

[4] Using the version at https://github.com/georgeawright/linguoplotter/tree/v2.0.0

2904-2949 A focus setter sets the newly built view as focus. Codelets matching sub frames and input structures to its frame increase its quality and cause a spike in satisfaction which subsides when the focus is unset.
3326 A view with a DISANALOGY frame is built.
3367 A garbage collector codelet runs and deletes the DISANALOGY view which had a low quality score because it was empty.
3000-5000 Worldview setters occasionally run causing the worldview to alternate between the two BE sentences.
4903 A view with a TEMPORAL-ORDER frame is built.
4930 A publisher codelet runs but does not publish because the focus is occupied by the TEMPORAL-ORDER view.
5411 The word *then* is built in the TEMPORAL-ORDER view's output space.
5573 A worldview setter selects the recently completed TEMPORAL-ORDER view.
5743 A publisher codelet runs and publishes the worldview.

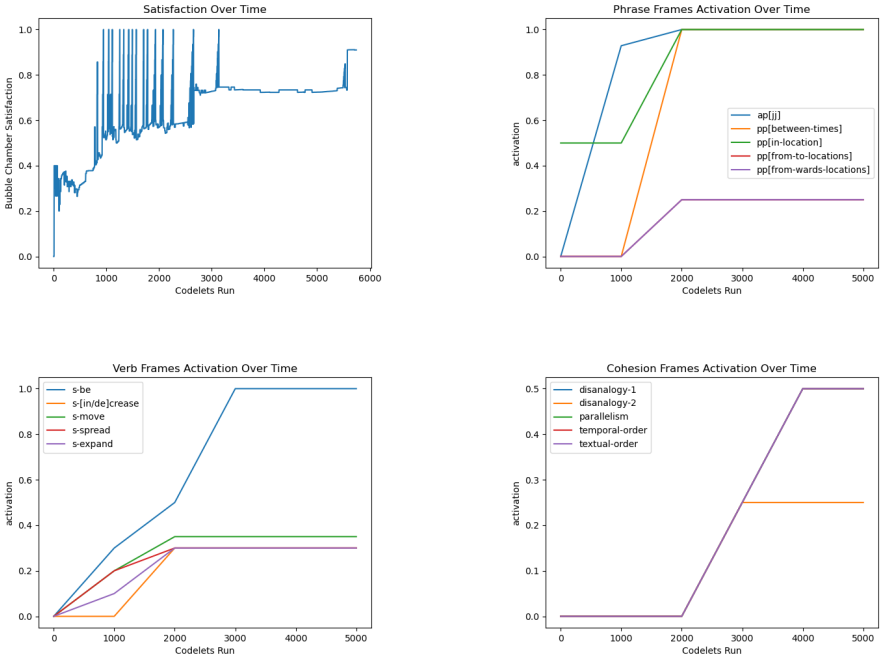

Fig. 5: Linguoplotter's satisfaction and the activation of frame types over time.

5.2 Emergent Pipelines

Although the architecture is not hard-coded to follow a modular data-to-text pipeline, "pipelines" do to some extent emerge out of the knock-on effects of

codelets and the spreading of activation between concepts. A bottom-up pipeline begins with codelets searching for disconnected structures in the bubble chamber. These suggest structures which, when built, spread activation to relevant concepts. This triggers top-down processing whereby codelets search for instances of active concepts or for structures to fill in the slots of unfinished simplex networks.

Overall, a pipeline also emerges which begins with lower-level pattern recognition and is followed by increasingly high-level structures from the level of phrases, to that of sentences, and on to cohesive texts. This is demonstrated by the changing activation of concepts and frames shown in figure 5. This is not dissimilar to the data-to-text pipeline of Reiter [6], but it is less rigid and can be interrupted by top-down processes which encourage reversion to an earlier stage. The collective behaviour of codelets thus results in a more autonomous and flexible alternative to programs following a pre-specified algorithm.

5.3 Problems the program encounters

As shown in tables 1 and 2, the program does not always perform as well as in section 5.1. Certain problems recur: often the program struggles to zero in on a good representation of the input; other times, it gets dominated by publisher codelets and outputs a text before allowing itself to find better alternatives.

Fruitless loops In 33 out of 50 runs when describing sequence 1c, the program fails to publish an output before timing out after 2×10^4 codelets.

When the program runs with sequence 1c and random seed 0, it performs well for approximately the first 5000 codelets, generating phrases and ultimately promoting a sentence (*temperatures will increase in the north between Friday and Saturday*) into the worldview. Unfortunately, it concurrently generates an identical sentence. After this point, codelets are more likely to instantiate frames for combining sentences, but are unable to complete the slots in the frames, because it is not possible to conjoin a sentence with itself. The program can identify networks that cannot be completed and tends to delete them, but it shortly after tries to recreate similar networks unaware that it is repeating itself. The amount of attention paid to an impossible task prevents the program from finding other sentences before it times out.

Premature Publication Sometimes the program makes the decision to publish a text even while it is half-way through generating a potentially better text.

When the program runs with sequence 1a and random seed 16, after a sentence (*temperatures will be cold in the country between saturday and sunday*) has been added to the worldview at time 1436, a publisher happens to run at 1536 and because the focus is empty it spawns another publisher with a slightly higher urgency. This triggers a stream of publishers that run intermittently with ever higher urgency until one at 1603 finally publishes the worldview. Had any of these publishers run when the focus was not empty, their urgency would have been lower and the program may have been able to build a fuller text.

That this can happen is a downside in terms of performance but also provides some degree of psychological realism: were the program to continue running beyond the point at which it makes the publication decision and therefore finish generating a better text, this could be seen as an example of the French concept of *l'espirit de l'escalier* or *staircase wit* — thinking of the perfect thing to say after it is too late — an entirely human behaviour!

6 Future Work: Meta-level pattern recognition

Whereas LINGUOPLOTTER used only to recognize patterns in the maps it described, recent improvements allow it also to recognize patterns in its own texts. But, the program can still struggle to find its way through a large search space and sometimes gets stuck repeatedly trying to build uncompletable networks.

Codelet activity in the bubble chamber can be narrated by an observer. If the program were able to do this itself, it could recognize patterns of futile behaviour such as those in section 5.3 and take action to avoid them.

METACAT, an extension of COPYCAT, holds a store of recent activity in a *trace* and uses codelets to recognize problematic behaviour. Other codelets can then alter processing by "clamping" patterns of structures that lead to failure so as to prevent them from re-occurring. This allows METACAT to "jump out of the system" and stop wasting time on fruitless loops to which COPYCAT was prone [3]. A similar extension to LINGUOPLOTTER could improve its performance at narrating weather patterns and would also lay the foundation for a program that can introspect and narrate itself.

7 Conclusion

Recent additions to LINGUOPLOTTER allow it to produce fuller texts by classifying intermediate texts in terms of cohesion relations built by codelets within its bubble chamber. This constitutes a step towards a greater entanglement of language production and comprehension. But the necessarily chaotic nature of the architecture means it is difficult to optimize and often does not work as well as would be hoped. Future work on the architecture should expand the range of patterns that the program can recognize in input and text so that it can replicate a wider range of human abilities; and should use the recognition of patterns in its own processes to avoid loops and other futile behaviour.

References

1. Fauconnier, G and Turner M. The Way We Think. Basic Books (2002)
2. Hofstadter, D and FARG: Fluid Concepts and Creative Analogies. Basic Books (1995)
3. Marshall, J.B. Metacat: A Self-Watching Cognitive Architecture for Analogy-Making and High-Level Perception. PhD Thesis, Indiana University (1999)

4. Mitchell, M. Analogy-Making as Perception. MIT Press (1993)
5. Pickering, M.J. and Garrod, S. An integrated theory of language production and comprehension. Behavioural and Brain Sciences **36**, 329-392 (2013)
6. Reiter, E: An Architecture for Data-to-Text Systems. In: Proceedings of the Eleventh European Workshop on Natural Language Generation, pp. 97-104 (2007)
7. Sharples, M: How We Write. Routledge (1998)
8. Turner, M: The Literary Mind. Oxford University Press (1996)
9. Wright, G.A. and Purver, M. A self-evaluating architecture for describing data. In: Text, Speech, and Dialogue: 25th International Conference, pp. 187âĂŞ198 (2022)