A Self-Evaluating Architecture for Describing Data*

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. This paper introduces Linguoplotter, a workspace-based architecture for generating short natural language descriptions. All processes within Linguoplotter are carried out by *codelets*, small pieces of code each responsible for making incremental changes to the program's state, the idea of which is borrowed from Hofstadter *et al* [6]. Codelets in Linguoplotter gradually transform a representation of temperatures on a map into a description which can be output. Many processes emerge in the program out of the actions of many codelets, including language generation, self-evaluation, and higher-level decisions such as when to stop a given process, and when to end all processing and publish a final text. The program outputs a piece of text along with a *satisfaction* score indicating how good the program judges the text to be. The iteration of the program described in this paper is capable of linguistically more diverse outputs than a previous version; human judges rate the outputs of this version more highly than those of the last; and there is some correlation between rankings by human judges and the program's own satisfaction score. But, the program still publishes disappointingly short and simple texts (despite being capable of longer, more complete descriptions). This paper describes: the workings of the program; a recent evaluation of its performance; and possible improvements for a future iteration.

Keywords: Language generation · Self-evaluation · Workspace · Codelet

1 Introduction

Work on language generation and language understanding are often kept separate, but in humans the two processes are intertwined [9]: for example, simultaneous use of production and comprehension allow people to interweave contributions in dialogue [2]. This paper introduces Linguoplotter, an attempt

* The authors were partially supported by the UK EPSRC under grants EP/R513106/1 (Wright) and EP/S033564/1 (Sodestream: Streamlining Social Decision Making for Improved Internet Standards).

at a cognitively plausible model of language generation in which constant self-evaluation and selection between competing ideas are integral to the generative process. The model is tested in a toy domain of temperatures on a fictional map.

Many of the core ideas of the program are borrowed from the Fluid Analogies Research Group, whose programs model high-level perceptual processes involved in analogy making [6]. Their programs such as COPYCAT [8] and TABLETOP [4] operate in different toy domains, but in essence do the same thing: search for a compact representation of their two inputs which allows for a satisfying mapping and an inference that solves a problem. Unlike other contemporary models of analogy-making such as the Structure Mapping Engine [3], the programs are not provided with a ready-made high-level representation of the input, but generate their own representation as part of the analogy making process.

LINGUOPLOTTER is not directly concerned with analogy making, but does create mappings from data into natural language descriptions and generates high-level representations of its input as part of the process. It avoids a pipeline architecture typical of many language generating programs which keep the analysis of data separate from the conversion to linguistic form (For example Reiter [11], Leppnen [7]). The program is thus more concordant with work such as by Turner [13] suggesting that language and narrative frames can influence the way we perceive the world.

This work is open to the charge of being old-fashioned and restricted to toy domains, but its aim is to produce an architecture which, like a neural network, displays high-level behaviour emergent from the interactions of small parts, while, like symbolic programs, is self-explanatory and easy to interpret.

2 How Linguoplotter Works

LINGUOPLOTTER[3] centers on a bubble chamber which contains a number of spaces representing the program's long- and short-term memory. Structures are built and connected to each other in these spaces and the best, most relevant structures bubble to the surface of the program's attention by receiving boosts in activation and spreading activation to related structures.

Long-term memory includes *concepts* (nodes located at prototypical points in conceptual spaces), *frames* (recursive structures which map between semantics and text), and *letter-chunks* (chunks of text ranging from morphemes to sentences) in a network of spreading activation.

Short-term structures built as the program runs include *chunks* which group together similar data points; *labels* which indicate that a node is an instance of a particular concept; *relations* which link items to create instances of relational concepts such as MORE or LESS); *correspondences* which map between items in different spaces (usually for the purpose of filling a slot in a frame); and *views* which collect together a consistent group of correspondences. Views have an output space where a piece of text made of *letter-chunks* is built.

[3] Source code is available at https://github.com/georgeawright/linguoplotter

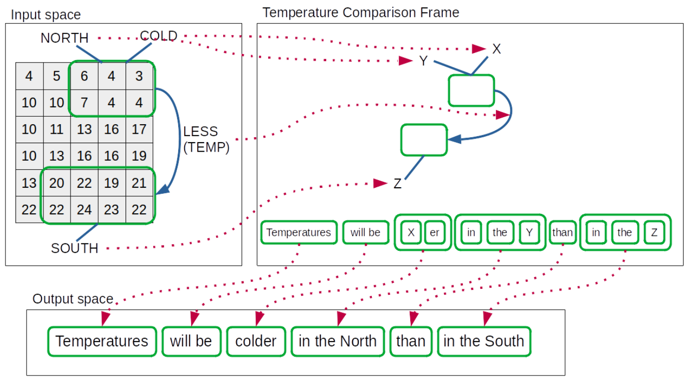


Fig. 1: Structures built by codelets inside a single view. Solid blue lines are labels and relations; dotted red lines are correspondences; green boxes are chunks.

All processing on structures is performed by *codelets*. Each codelet has a small, specific task and after running, spawns another codelet to carry out follow-up work. Codelets have an urgency representing the importance of their work which the *coderack* uses when stochastically selecting the next codelet to be run.

Most codelets belong to a cycle which begins with a *suggester* codelet. A suggester finds target structures and suggests a chunk or link that can be built with them. For example, a *label suggester* classifies the proximity of a chunk to a concept and spawns a *label builder* with an urgency reflecting its confidence that the chunk belongs to that concept. A *builder* codelet builds structures and spawns an *evaluator* codelet. An *evaluator* decides how good a structure is and assigns it a quality score. It then spawns a *selector* codelet that chooses between alternative structures, for example two different labels belonging to the same chunk, and probabilistically boosts the activation of the best one. The *selector* codelet spawns a *suggester* codelet to continue building related structures and an *evaluator* so as to maintain the process of selecting the best structure.

Left unchecked, this forking of codelet cycles would cause an explosion in the population of the coderack, hence *coderack cleaners* remove codelets that are no longer contributing to an increase in the program's satisfaction. In order to avoid the depletion of the coderack population and to make sure that active concepts and frames have structures suggested for them, *factory* codelets spawn suggesters to inititate new cycles of codelets.

The running of codelets and selection of structures all happens with some degree of randomness which is determined by *satisfaction*. This is a measure of the overall quality of structures in the bubble chamber. When the program lacks active high quality structures, it is more random and pursues more options in the search for a solution, but once good quality structures have been built and activated, it proceeds more deterministically towards a finished piece of text.

Earlier iterations of the program were prone to too much randomness and failed to narrow down on a single pathway towards a solution. This iteration of

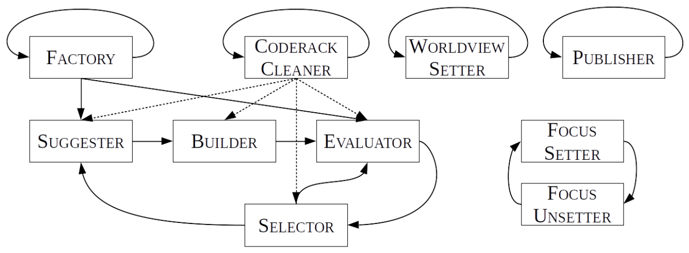


Fig. 2: Codelet types that run in the program. Solid arrows show each type's follow-up type. Dashed arrows show which types a *coderack cleaner* can remove.

the program has a *focus setter* codelet which chooses a single view at a time as *focus*. Once a view is set as focus, codelets are targeted towards it and fill in the slots in its frame. Having a *focus* gives the program a short-term sub-goal and a narrower search space. A lack of progress or completion of the view causes a *focus unsetter* to remove the view from focus. The satisfaction of the program then falls; randomness increases; and the search space broadens.

Linguoplotter also has a longer-term *Worldview* (an idea borrowed from Tabletop [4]) which represents the best view completed so far. Its output space contains a candidate piece of text for publication. Every time a *worldview setter* codelet runs it searches for a better alternative to the current worldview. If it fails, it sends activation to the publish concept. Eventually the program fails to improve on itself and if a *publisher* codelet runs when the publish concept is fully activated, the program halts and the text in the worldview is output.

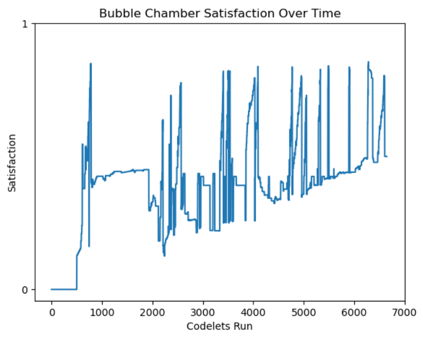


Fig. 3: Bubble chamber satisfaction over a run of the program. Satisfaction spikes when the focus is set and its slots filled in. Satisfaction dips when the focus is unset. Satisfaction increases over time as the worldview is set or improved upon.

2.1 Macro-level Processes in Linguoplotter

Macro-level processes emerge from Linguoplotter's interacting components such as: cycles that reinforce certain concepts and frames; alternation between more engaged and more reflective behaviour; and the gradual decision to publish.

Self-Reinforcing Concepts When a selector codelet chooses from competing alternatives, it spawns a follow up suggester to suggest another related structure, for example another link with the same concept or another view with the same frame. This results in a snowballing of the use of certain conceptual structures. For example, having created a sentence with a certain frame, the program would be likely to start another sentence with the same frame.

Cycles of Engagement and Reflection As the program runs, the changing satisfaction score affects the breadth of processing. When the focus is set and slots in its frame are filled in, satisfaction spikes and processing becomes more deterministic and dedicated to that set of structures. But when there is no focus, or when little progress is being made on the focus, processing is more random and lacks a clear aim. This is analogous to Sharples' cycle of engagement and reflection [12], the idea that humans alternate between bursts of purposeful behaviour and periods of greater reflection and exploration. Unlike Mexica [10], a model of narrative generation based on Sharples' idea, this program's cycle of engagement and reflection is not explicitly coded, but results from a feedback loop between codelets altering the bubble chamber and bubble chamber satisfaction adjusting the randomness of codelet selection.

Publishing The decision to publish is the responsibility of *worldview setter* and *publisher* codelets, but can be affected by the intervention of other codelets. Repeated failure of *worldview setters* results in the boosting of the publish concept. This increases the likelihood of publication. But, if other codelets run and maintain other processes, the publish concept will decay. The interactions of different processes thus result in a period of indecision. Only a sustained stream of failed worldview-setting can lead to publication.

2.2 Developing the Program

Linguoplotter's many interleaved processes give it great potential, but also make it difficult to optimize. Its behaviour can be altered by tweaking a number of parameters, such as the method for calculating each codelet's urgency; the method used to define the quality of each structure; and the formula for calculating satisfaction. There are also a number of hyper-parameters, such as the rate at which activation spreads between structures, the decay rate of activation, and the method for determining how random the program should be.

As shown by the outputs of the program in table 1, the program in its current form is stuck in a local optimum where it is capable of producing outputs which are, for the most part, true descriptions of the input, but which lack detail and leave some of the input undescribed, even though it is capable of linking sentences with connectives such as *and* to produce a more full description.

Calculating the Satisfaction Score Satisfaction has three components: general satisfaction G, focus satisfaction F, and worldview satisfaction W. The overall satisfaction of the program S is given by:

$$S = max(F, mean(G, W))$$

If there is a view in the focus, otherwise:

$$S = mean(G, W)$$

This limits satisfaction to 0.5 when there is no worldview and no focus and prevents satisfaction from dropping to 0 when an empty view is placed in the focus (this would result in highly random and unfocused behaviour).

General Satisfaction G is the mean quality of all the input and output spaces of the bubble chamber's views. A space's quality is determined by:

$$space\_quality = \frac{\sum_{a \in A} quality(a) \times activation(a)}{|A|}$$

Where A is the set of structures in the space with an activation greater than 0.5. A high quality space must contain active high quality structures.

Worldview Satisfaction W is calculated as:

$$W = AND(OR(Q, \frac{D}{10}), OR(P, \frac{1}{T}))$$

Where

$$AND(X, Y) = X \times Y$$

$$OR(X, Y) = X + Y - X \times Y$$

Q is the quality of the worldview as determined by evaluator codelets; D is the depth of the view's parent frame (depth is a number between 1 and 10, with lower numbers given to frames for simple phrases and higher numbers given to sentences and conjunctions); P is the proportion of the input involved in the view; T is the total number of frame types used by the view (i.e. types of phrase constituting the sentence). This satisfaction metric encourages the program to output sentences which are correct (high Q), complete (high P), grammatically complete (high D), and succinct (low T).

Focus Satisfaction F is calculated as:

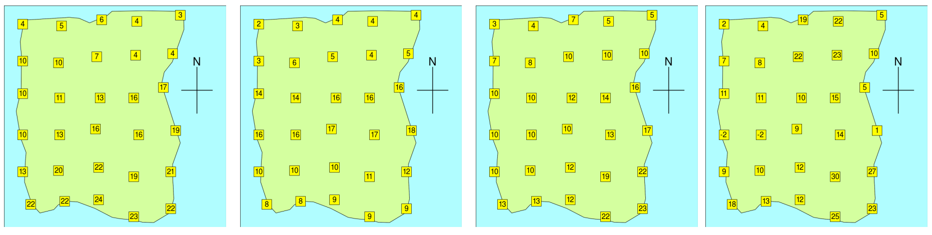
$$F = mean(mean\_correspondence\_quality, \frac{|FilledSlots|}{|Slots|})$$

Where correspondence qualities are determined by codelets according to the quality of their arguments and how well they fill in frame slots.

Optimizing the Satisfaction Score Changes to the satisfaction score affect the program's randomness and the texts that it prefers to publish. For example, calculating worldview satisfaction as the product of Q, $\frac{D}{10}$, P, and $\frac{1}{T}$ generally lowers overall satisfaction and makes the program unlikely to terminate, even after finding a more complete description than the current iteration. Alternatively, using the mean of the four components to calculate worldview satisfaction can result in very high satisfaction scores and causes the program to publish an output much earlier than the current iteration before any significant traversal of the search space. Further investigation is required in the search for a satisfaction formula which allows the program to distinguish between good and bad outputs and to estimate a good time to stop working and publish.

3 Performance of Linguoplotter

Linguoplotter was tested on the four inputs shown in figure 5. Three of these are variations of a similar input. The fourth is a more challenging map which has little or no pattern in the temperatures. The program was run 30 times for each map. Table 1 shows the mean and standard deviation for the number of codelets that were run before the program published a description of each input. It also shows the output that gave the highest mean satisfaction score, the output that was most frequently output (if there were two or more equally frequent outputs, the one one with highest mean satisfaction is shown), and the output with the lowest mean satisfaction score. Also shown for each input are two hand-selected human generated texts (one a detailed description and one written in note-form), and a text output by a previous version (judged by humans to be more correct than its other outputs.)



Input 1 Input 2 Input 3 Input 4

Fig. 5: The four maps described by the program.

3.1 Method for Evaluation by Human Judges

The program's outputs are evaluated through pairwise comparison. This avoids the difficulty of requesting a numeric score for a subjective judgement (describing a text as 100% interesting would be nonsensical) and results in less variance between respondents [1]. It also allows for direct comparison between different iterations of the program and humans. This iteration of the program was evaluated using the 24 texts in table 1.

	Input 1	Input 2
	Mean run length: 10425 (σ: 6197)	Mean run length: 12857 (σ: 6353)
D	The temperature is cold in the north but progressively warm moving south, reaching 24 degrees.	It is generally warmer in the south than the north but warmest in the central regions.
N	Cool in the north, warm in the south.	Cold in the north, milder in the centre. cooler in the south.
B	Temperatures will be higher in the east than in the north. (0.675)	Temperatures will be colder in the east than in the west. (0.670)
F	Temperatures will be colder in the north than in the south. (0.644)	Temperatures will be warm in the east. (0.405)
W	Temperatures will be better in the southwest than in the northwest. (0.326)	Temperatures will be cool in the west. (0.290)
O	It is hot in the southeast.	It is mild in the south.
	Input 3	Input 4
	Mean run length: 10560 (σ: 7824)	Mean run length: 13583 (σ: 9660)
D	The temperature gets much warmer as you go from northwest to southeast. it's very chilly in the northwest and warm in the southeast.	The temperature is inconsistent across the region with isolated pockets of high and low temperatures in various places.
N	It is cold in the north, a little warmer elsewhere, but warm in the south east.	The temperatures are very erratic.
B	Temperatures will be higher in the southeast than in the northwest. (0.674)	Temperatures will be cooler in the west than in the northeast. (0.673)
F	Temperatures will be cool in the north. (0.604)	Temperatures will be hot in the north. (0.540)
W	Temperatures will be higher in the south than in the northwest. (0.350)	Temperatures will be lower in the southwest than in the southeast. (0.338)
O	The north is cold.	The southeast is hot.

Table 1: Texts for inputs 1-4 (D: human (detailed); N: human (note-form); B: best satisfaction; F: most frequent; W: worst satisfaction; O: old version). Mean satisfaction scores are given in parentheses after machine-generated texts.

For each possible pair of texts, human judges were asked five questions in order to evaluate the texts along different dimensions:

1. Which text is easier to understand?

2. Which text is written more fluently?
3. Which text do you find more interesting?
4. Which text is more factually correct with regard to the temperatures on the map?
5. Which text provides a more complete description of the map?

Respondents could answer by either stating that one text was better than the other or that they were both the same. At no point were the respondents told that texts were machine-generated or human-generated.

Previous evaluation of the program only sought judgements on *easiness*, *fluency*, *correctness*, and *completeness*. These are typical characteristics considered in the evaluation of the quality and accuracy of machine generated language [5]. This time *interestingness* is also considered so as to measure the extent to which the computer program can match humans' creative flair when writing.

3.2 Results of Human Evaluation

Table 2 shows aggregate rankings of the different texts for each map calculated using the pairwise preferences given by human judges answering a survey on Amazon Mechanical Turk. Rankings were calculated by giving a text a score where it gained 1 point each time it was preferred over another text and lost 1 point each time another text was preferred over it. Overall, the outputs of the program are judged better than outputs of the previous version, but still lag behind the best texts generated by humans.

It should be noted that there was little agreement between human judges. Fleiss' Kappa was -0.016 for *easiness*, -0.018 for *fluency*, -0.018 for *interestingness*, -0.014 for *correctness*, and -0.007 for *completeness*. Low agreement is inevitable with subjective judgements, but this is also partly due to a large number of annotators – 36: only 2 annotators answered all questions in the survey. Agreement was greater between those two annotators, especially along the more objective dimensions of *correctness* (0.266) and *completeness* (0.589). It might be better in future for each annotator to provide rankings for all texts instead of just pairwise preferences so that a greater number of judgements per person can be obtained.

The latest iteration of the program consistently performs better than the previous evaluated version along the dimensions of *interestingness*, *correctness*, and *completeness*, but fails to match the best human performance.

As found when evaluating the previous version, humans tend to perform worse in terms of *easiness*, most likely because the computer-generated texts are simpler. Sometimes the previous iteration of the program (which produced shorter sentences) also outperformed the latest iteration along this dimension.

Interestingly, the latest iteration's best output ranked highest for all inputs in terms of fluency, usually followed by the detailed human text. The mean Spearman's rank correlation coefficient for each input between the rankings of the program's satisfaction scores and the aggregated human judge rankings for fluency is 0.875. Correlation is lower for the other dimensions (0.625 for interestingness, 0.25 for completeness, 0 for correctness, and 0 for easiness). Since the number of texts being compared is so small, little weight should be given to these correlation scores. Nevertheless this does suggest that more work is needed to improve the program's satisfaction score, not only to optimize the running of the program, but also to improve its judgement.

Input 1	Easy	Fluent	Interesting	Correct	Complete
1.	F	B	D	D	D
2.	N	D	B	F	F
3.	B	F	F	N	N
4.	O	O	N	B	B
5.	D	W	W	O	W
6.	W	N	O	W	O
Input 2	Easy	Fluent	Interesting	Correct	Complete
1.	F	B	D	D	D
2.	W	D	N	N	N
3.	B	W	B	F	F
4.	O	F	W	W	W
5.	D	O	F	O	O
6.	N	N	O	B	B
Input 3	Easy	Fluent	Interesting	Correct	Complete
1.	O	B	D	D	D
2.	F	D	N	W	N
3.	B	F	B	B	W
4.	W	W	W	N	B
5.	D	N	F	F	F
6.	N	O	O	O	O
Input 4	Easy	Fluent	Interesting	Correct	Complete
1.	O	B	D	O	D
2.	F	F	B	W	B
3.	B	W	W	D	W
4.	W	D	N	N	N
5.	N	O	F	B	O
6.	D	N	O	F	F

Table 2: Average rankings according to pairwise preferences for inputs 1-4.

3.3 Discussion

Overall, the program produces better outputs than its earlier iteration, but still falls short of human performance. It has some wherewithal to recognize poor performance in the form of its satisfaction score, but this also needs improving. The current configuration of the program takes on the order of 10^4 codelets to run and normally outputs a single sentence comparing two areas of the input map, though it sometimes produces a sentence describing a single area.

Its descriptions either make use of the TEMPERATURE space of the original input (*temperatures will be colder in the north than in the south*) or the HEIGHT or GOODNESS spaces (*temperatures will be higher in the east that in the north*, *temperatures will be better in the southwest than in the northwest*). There is little variation in the program's language but more than in the previous version which only used TEMPERATURE adjectives and simpler sentences. The program's outputs still fall far short of the richer human produced texts, but this is partly due to a lack of sentence frames available to the program.

The program's comparisons do not always seem sensible to the human ear, although they may be acceptable in terms of being truthful. For example, the program's "best" output for input 1 is a comparison between the *east* and the *north*. This description is odd, partly because it neglects a large part of the map, but also because it is ambiguous about the northeast. The program's more frequent output comparing the *north* and the *south* is ranked higher by human judges in terms of correctness, completeness, and easiness. It is strange that the "best" output is ranked higher in terms of fluency and interestingness, but there is especially low agreement for these dimensions.

On average the program took longest to describe input 4, but the difference is not significant and it shows a similar range of satisfaction scores for the outputs. Since input 4 ought to have been more difficult for the program to describe, a longer run time was to be expected, but the similar outputs and similar spread of satisfaction scores was not. Inspections of the structures built inside the program for each input indicate that relatively small chunks are being used to fill in frame slots and generate descriptions. The program therefore judges its descriptions to represent similar proportions of the map even though the descriptions for inputs 1-3 should be interpreted as describing larger areas. The program either builds large chunks and neglects to use them or fails to build larger chunks before generating text.

4 Future Improvements

It is clear from analysis of both the program's behaviour and its outputs, that further work is required to improve the program's satisfaction score and its ability to judge between good or bad outputs. This includes making sure that the program recognizes how correct and complete its descriptions are.

The program must remedy its description's lack of completeness by both building chunks which cover a wider area of the input when possible so as to gauge more accurately how much of the input a sentence can be interpreted as describing, but also to continue processing and generate more sentences when the text produced so far does not provide a full description.

Knowledge represented in the program and taken into account by self-evaluation ought also to include more aesthetic considerations. For example, the program ought to prefer comparison between opposite locations. Odd statements should not be forbidden in case a different context makes them relevant, but the program should generally avoid them.

Future work should also include a more thorough search for a set of hyper-parameters that will encourage good quality and complete descriptions of the input. This will not necessarily require large-scale changes to the program, but remains a challenge considering the complexity of the search space.

Ultimately, the program must also be put to work in a more complex domain in order to test its general applicability to the task of generating descriptions.

5 Conclusion

This paper presents an advance towards an explainable model of language generation which can evaluate its own work and is directly affected by its own self-evaluation. The idea has great cognitive plausibility due to the intertwining of different processes, but the implementation thus far still lags behind humans in terms of both language generation and language evaluation. Future work must focus on improving the program's in-built aesthetic measures so that it can more reliably predict human judgements; produce better descriptions; and know when to publish them.

References

1. Belz, A. and Kow, E.: Comparing Rating Scales and Preference Judgements in Language Evaluation. In: Proc. 6th International Conference on Natural Language Generation, pp. 7-15 (2010)
2. Clark, H.H.: Using Language. Cambridge University Press (1996)
3. Falkenhainer, B., Forbus, K.D., and Gentner, D.: The Structure Mapping Engine: Algorithm and Examples. Artificial Intelligence **41**, 1-63 (1989)
4. French, R.M: The Subtlety of Sameness: A Theory and Computer Model of Analogy-Making. MIT Press (1995)
5. Gatt, A. and Krahmer, E.: Survey of the State of the Art in Natural Language Generation: Core tasks, applications, and evaluation. Journal of Artificial Intelligence Research **61**, 65-170 (2018)
6. Hofstadter, D. and FARG: Fluid Concepts and Creative Analogies. Basic Books (1995)
7. Leppnen, L., Munezero, M., and Granroth-Wilding, M., and Toivonen, H.: Data-Driven News Generation for Automated Journalism. In: Proc. 10th International Natural Language Generation Conference, pp. 188-197 (2017)
8. Mitchell, M. Analogy-Making as Perception: A Computer Model. MIT Press (1993)
9. Pickering, M.J. and Garrod, S. An integrated theory of language production and comprehension. Behavioural and Brain Sciences **36**, 329-392 (2013)
10. Prez y Prez, R. and Sharples, M.: Mexica: A Computer Model of a Cognitive Account of Creative Writing. Journal of Experimental and Theoretical Artificial Intelligence **13**(2), 119-139 (2001)
11. Reiter, E: An Architecture for Data-to-Text Systems. In: Proceedings of the Eleventh European Workshop on Natural Language Generation, pp. 97-104 (2007)
12. Sharples, M: How We Write: Writing as Creative Design. Routledge (1998)
13. Turner, M: The Literary Mind: The Origins of Thought and Language. Oxford University Press (1996)