Towards a Programmable Instrumented Generator

Chris Mellish
Computing Science
University of Aberdeen
AB24 3UE, UK
`c.mellish@abdn.ac.uk`

Abstract

In this paper, we propose a general way of constructing an NLG system that permits the systematic exploration of the effects of particular system choices on output quality. We call a system developed according to this model a *Programmable Instrumented Generator* (PIG). Although a PIG could be designed and implemented from scratch, it is likely that researchers would also want to create PIGs based on existing systems. We therefore propose an approach to "instrumenting" an NLG system so as to make it PIG-like. To experiment with the idea, we have produced code to support the "instrumenting" of any NLG system written in Java. We report on initial experiments with "instrumenting" two existing systems and attempting to "tune" them to produce text satisfying complex stylistic constraints.

1 Introduction

Existing NLG systems are often fairly impenetrable pieces of code. It is hard to see what an NLG system is doing and usually impossible to drive it in any way other than what was originally envisaged. This is particularly unfortunate if the system is supposed to produce text satisfying complex stylistic requirements. Even when an NLG system actually performs very well, it is hard to see why this is or how particular generator decisions produce the overall effects. We propose a way of building systems that will permit more systematic exploration of decisions and their consequences, as well as better exploitation of machine learning to make these decisions better. We call a system built in this way a Programmable Instrumented Generator (PIG). As an initial exploration of the PIG idea, we have developed a general way of partially instrumenting any NLG system written in Java and have carried out two short experiments with existing NLG systems.

2 Controlling an NLG System: Examples

NLG systems are frequently required to produce output that conforms to particular stylistic guidelines. Often conformance can only be tested at the end of the NLG pipeline, when a whole number of complex strategic and tactical decisions have been made, resulting in a complete text. A number of recent pieces of work have begun to address the question of how to tune systems in order to make the decisions that lead to the most stylistically preferred outputs.

Paiva and Evans (2005) (henceforth PE) investigate controlling generator decisions for achieving stylistic goals, e.g. choices between:

> The patient takes the two gram dose of the patient's medicine twice a day.

and

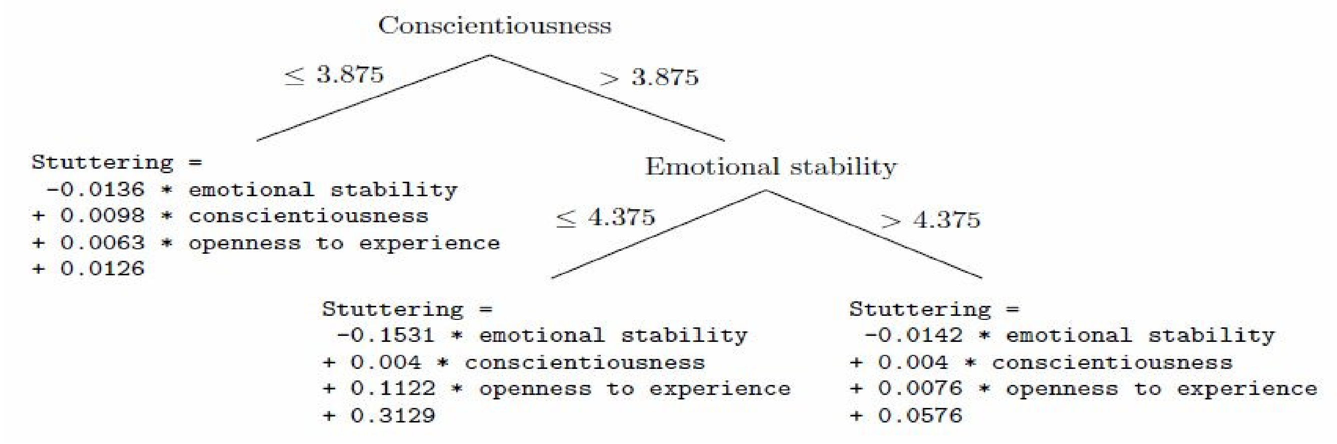
> The dose of the patient's medicine is taken twice a day. It is two grams.

In this case, a stylistic goal of the system is expressed as goal values for features SS_i, where each SS_i expresses something that can be measured in the output text, e.g. counting the number of pronouns or passives. The system learns to control the number of times specific binary generator decisions are made (GD_j), where these decisions involve things like whether to split the input into 2 sentences or whether to generate an N PP clause. A process of *offline training* is first used to establish correspondences between counts of generator decisions and the values of the stylistic features. This works by running the system with multiple outputs (making decisions in many possible ways) and keeping track of both the counts of the decisions and also the values of the stylistic features achieved. From this data the system then learns correlations between these:

$$SS_i \cong SS_i^{est} = x_0 + \sum_j x_j.GD_j$$

To actually generate a text given stylistic goals SS_i, the system then uses an *online control* regime. At each choice point, it considers making GD_j versus not GD_j. For each of these two, it estimates all the SS_i that will be obtained for the complete text, using the learned equations. It prefers the choice that minimises the sum of absolute differences between these and the goal SS_i, but is prepared to backtrack if necessary (best-first search).

Mairesse and Walker (2008) (henceforth MW) use a different method for tuning their NLG system ("PERSONAGE"), whose objective is to produce texts in the styles of writers with different personality types. In this case, the system performance depends on 67 parameters, e.g. REPETITIONS (whether to repeat existing propositions), PERIOD (leave two sentences connected just with ".", rather than any other connective) and NEGATION (negate a verb and replace it by its antonym). For MW, *offline training* involves having the program generate a set of outputs with random values for all the parameters. Human judges estimate values for the "big five" personality traits (e.g. extroversion, neuroticism) for each output. Machine learning is then used to generate rules to predict how the parameter values depend on the big five numbers. For instance, Figure 1 shows the rule predicting the STUTTERING parameter.



Conscientiousness
≤ 3.875
> 3.875
Stuttering = -0.0136 * emotional stability + 0.0098 * conscientiousness + 0.0063 * openness to experience + 0.0126
Emotional stability
≤ 4.375
> 4.375
Stuttering = -0.1531 * emotional stability + 0.004 * conscientiousness + 0.1122 * openness to experience + 0.3129
Stuttering = -0.0142 * emotional stability + 0.004 * conscientiousness + 0.0076 * openness to experience + 0.0576

Figure 1: Example PERSONAGE rule

Once these rules are learned, *online control* to produce text according to a given personality (specified by numerical values for the big five traits) uses the learned models to set the parameters, which then determine NLG system behaviour. Human judges indeed recognise these personalities in the texts.

3 Towards a PIG

Looking at the previous two examples, one can detect some common features which could be used in other situations:

- An NLG system able to generate random (or all possible) outputs
- Outputs which can be evaluated (by human or machine)
- The logging of key NLG parameters/choices
- Learning of connections between parameters and output quality

This then being used to drive the system to achieve specific goals more efficiently than before.

PE and MW both constructed special NLG systems for their work. One reason for this was that both wanted to ensure that the underlying NLG system allowed the kinds of stylistic variation that would be relevant for their applications. But also, in order to be able to track the choices made by a generator, Paiva and Evans had to implement a new system that kept an explicit record of choices made. This new system also had to be able to organise the search through choices according to a best-first search (it was possibly the first NLG system to be driven in this way). The only possibility for them was to implement a new special purpose generator for their domain with the desired control characteristics.

NLG systems are not usually immediately suitable for tuning of this kind because they make choices that are not exposed for external inspection. Also the way in which choices are made and the overall search strategy is usually hardwired in a way that prevents easy changing. It seems plausible that the approaches of PE and MW would work to some extent for *any* NLG system that can tell you about its choices/ parameter settings, and for *any* stylistic goal whose success can be measured in the text. Morever, these two are not the only ways one might train/guide an NLG system from such information (for instance, Hovy's (1990) notion of "monitoring" would be an alternative way of using learned rules to drive the choices of an NLG system). It would be revealing if one could easily compare different control regimes in a single application (e.g. monitoring for PE's task or best-first search for MW's), but this is currently difficult because the different systems already have particular control built in.

This discussion motivates the idea of developing a general methodology for the development of NLG systems that permits the systematic exploration of learning and control possibilities. We call a system built in such a way a *Programmable Instrumented Generator* (PIG).[1] A PIG would be an NLG system that implements standard NLG algorithms and competences but which is organised in a way that permits inspection and reuse. It would be *instrumented*, in that one would be able to track the choices made in generating a text or texts, in order to tune the performance. It would also be *programmable* in that it would be possible to drive the system in different ways according to a learned (or otherwise determined) "policy", e.g. to:

- Generate all solutions (overgeneration)
- Generate solutions with some choices fixed/constrained
- Generate solutions with user control of some decisions
- Generate solutions using an in-built choice mechanism
- Generate solutions according to some global search strategy (e.g. monitoring, best-first search)

4 Using a PIG

A general way of using a PIG is shown in Figure 2. A PIG interacts with a (conceptually) separate processing component, which we call the "oracle". This applies a *policy* to make choices for the generator and receives evaluations of generated texts. It logs the choices made and (using machine learning) can use this information to influence the policy.

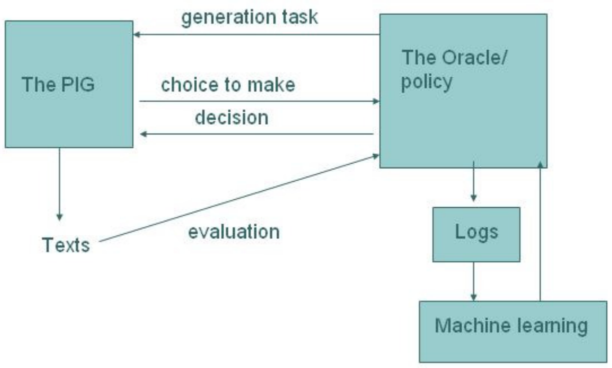


Figure 2: Using a PIG

There are two main modes in which the PIG can be run, though mixtures are also possible. In (offline) *training* mode, the system is run on multiple inputs and uses random or exhaustive search to sample the space of generatable texts. The choices made

[1] If one had a sufficiently expressive PIG then perhaps one could train it for *any* testable stylistic goals – a kind of "universal" NLG system?

are logged, as is the quality of the outputs generated. In (online) *execution* mode, the PIG is run as a normal generator, running on a single input and making choices according to a learned policy.

To support this, the PIG itself needs minimally to support provide external access to the following function:

```
generate(input:InputSpec) returns text:String
```

which produces a text, from a given input specification. On the other hand, the Oracle needs to provide external access to at least the following (used by the PIG):

```
choice(question:String, suggestion:int,
    possibilities:ListOfString, state:String)
returns decision:int  or RESTART

outcome(state:String, value:Float) (no return value)
```

where question represents a choice to be made (with possible answers possibilities), suggestion is the index of a suggested choice and decision is the index of the choice made. state is a representation of generator state, in some standard format (e.g. ARFF (Hall et al 2009)) and outcome (giving the final state and the text quality) is called as the last action of generating a text. RESTART is a special value that by convention causes the system to return to a state where it can be asked to generate another text.

To support the above, the PIG needs to maintain some representation of program state. Also the oracle needs to implement a training/testing algorithm that involves providing the PIG with example inputs, restarting the PIG on the current or a new example, implementing a policy, logging results and possibly interacting with a user.

The above model of how to use a PIG is partly motivated by existing approaches to monitoring and testing complex electronic equipment. Testing is often carried out by attaching “automatic test equipment” to the unit under test. This automatic test equipment is akin to our “oracle” in that it drives the unit through special test sequences and automatically records what is going on.

5 The PIG panel

There is a practical question of how best to build PIGs and what resources there might be to support this. Given their concern with explicit representation of choices, NLG models based on Systemic Grammar (Bateman 1997) might well be promising as a general framework here. But in reality, NLG systems are built using many different theoretical approaches, and most decisions are hard-coded in a conventional programming language. In order to investigate the PIG concept further, therefore, we have developed a general way of “instrumenting” in a limited way any NLG system written in Java (giving rise to a *PIGlet*). We have also implemented a general enough oracle for some initial experiments to be made with a couple of PIGlets. This experimental work is in line with the API given above but implemented in a way specific to the Java language.

In order to instrument the client generator, one has to identify places where interesting choices are made. This is obviously best done by someone with knowledge of the system. There are a number of ways to do this, but the simplest basically replaces a construct of the form:

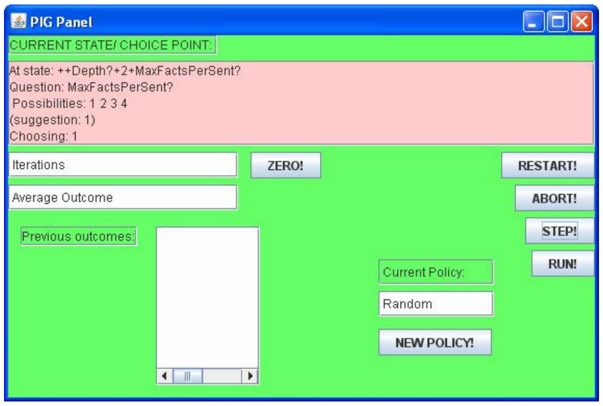
```
if (<condition>) <action>
```

by

```
if (Oracle.condRec(<name>,<condition>)) <action>
```

where <name> is a string naming this particular choice. This allows the oracle to intervene when the choice is made, but possibly taking into account the suggested answer (<condition>).

The implemented oracle (the “PIG panel”) supports a kind of “single stepping” of the generator (between successive choices), manual control of choices and restarting. It has built in policies which include random generation, following the choices suggested by the PIGlet, systematic generation of all possibilities (depth-first) and SARSA, a kind of reinforcement learning (Sutton and Barto 1998). It provides simple statistics about the evaluations of the texts generated using the current policy and a user interface (Figure 3).



```
CURRENT STATE/ CHOICE POINT:
At state: ++Depth?+2+MaxFactsPerSent?
Question: MaxFactsPerSent?
 Possibilities: 1 2 3 4
(suggestion: 1)
Choosing: 1
Iterations
Average Outcome
Previous outcomes:
Current Policy:
Random
```

Figure 3: PIG Panel interface

For the oracle to be able to control the PIGlet, it needs to be provided with a "connector" which represents it through a standard API (specifying how to generate a text, how to evaluate a text, what examples can be used, etc.). This also includes a specification of how to derive the "state" information about the generator which is logged for machine learning process. State information can include the number of times particular choices are made (as in PE), the most recent choices made and other generator-specific parameters which are communicated to the oracle (as in MW).

Finally the PIGlet and oracle are linked via a "harness" which specifies the basic mode of operation (essentially training vs execution).

In the following sections, we describe two tentative experiments which produced PIGlets from existing NLG systems and investigated the use of the PIG panel to support training of the system. It is important to note that for these systems the instrumenting was done by someone (the author) with limited knowledge of the underlying NLG system and with a notion of text quality different from that used by the original system. Also, in both cases the limited availability of example data meant that testing had to be performed on the training data (and so any positive results may be partly due to overfitting).

6 Experiment 1: Matching human texts

For this experiment, we took an NLG system that produces pollen forecasts and was written by Ross Turner (Turner et al 2006). Turner collected 68 examples of pollen prediction data for Scotland (each consisting of 6 small integers and a characterisation of the previous trend) with human-written forecasts, which we took as both our training and test data. We evaluated text quality by similarity to the human text, as measured by the Meteor metric (Lavie and Denkowski 2009). Note that the human forecasters had access to more background knowledge than the system, and so this is not a task that the system would be expected to do particularly well on.

The notion of program "state" that the oracle logged took the form of the 6 input values, together with the values of 7 choices made by the system (relating to the inclusion of trend information, thresholds for the words "high" and "low", whether to segment the data and whether to include hay fever information).

The system was trained by generating about 10000 random texts (making random decisions for randomly selected examples). For each, the numerical outcome (Meteor score) and state information was recorded. The half of the resulting data with highest outcomes was extracted and used to predict rules for the 7 choices, given the 6 input parameters (we used Weka (Hall et al 2009) with the JRip algorithm). The resulting rules were transcribed into a specific "policy" (Java class) for the oracle.

Applied to the 68 examples, trying random generation for 3 times on each, the system obtained an average Meteor score of 0.265. Following the original system's suggestions produced an average score of 0.279. Following the learned policy, the system also obtained an average of 0.279. The difference between the learned behaviour and random generation is significant (p =0.002) according to a t test.

7 Experiment 2: Text length control

A challenging stylistic requirement for NLG is that of producing a text satisfying precise length requirements (Reiter 2000). For this experiment, we took the EleonPlus NLG system developed by Hien Nguyen. This combines the existing Eleon user interface for domain authoring (Bilidas et al 2007) with a new NLG system that incorporates the SimpleNLG realiser (Gatt and Reiter 2009).

The system was used for a simple domain of texts about university buildings. The data used was the authored information about 7 university buildings and associated objects. We evaluated texts using a simple (character) length criterion, where the ideal text was 250 characters, with a steeply increasing penalty for texts longer than this and a slowly increasing penalty for texts that are shorter.

The notion of "state" that was logged took account of the depth of the traversal of the domain data, the maximum number of facts per sentence and an aggregation decision.

Following the previous successful demonstration of reinforcement learning for NLG decisions (Rieser and Lemon 2006), we decided to use the SARSA approach (though without function approximation) for the training. This involves rewarding individual states for their (direct or indirect) influence on outcome quality as the system actually performs. The policy is a mixture of random exploration and the choosing of the currently most promising states, according to the value of a numerical parameter ε.

Running the system on the 7 examples with 3 random generations for each produced an average text quality of -2514. We tried a SARSA training regime with 3000 random examples at $\varepsilon=0.1$, followed by 2000 random examples at $\varepsilon=0.001$. Following this, we looked at performance on the 7 examples with $\varepsilon=0$. The average text quality was -149. This was exactly the same quality as that achieved by following the original NLG system's policy. Even though there is a large difference in average quality between random generation and the learned policy, this is, however, not statistically significant ($p = 0.12$) because of the small number of examples and large variation between text qualities.

8 Conclusions and Further Work

Each of our initial experiments was carried out by a single person in less than a week of work, (which included some concurrent development of the PIG panel software and some initial exploration of the underlying NLG system). This shows that it is relatively quick (even with limited knowledge of the original NLG system) for someone to instrument an existing NLG system and to begin to investigate ways of optimizing its performance (perhaps with different goals than it was originally built for). This result is probably more important than the particular results achieved (though it is promising that some are statistically significant).

Further work on the general software could focus on the issue of the visualization of choices. Here it might be interesting to impose a Systemic network description on the interdependencies between choices, even when the underlying system is built with quite a different methodology.

More important, however, is to develop a better understanding of what sorts of behaviour in an NLG system can be exposed to machine learning to optimize the satisfaction of what kinds of stylistic goals. Also we need to develop methodologies for systematically exploring the possibilities, in terms of the characterization of NLG system state and the types of learning that are attempted. It is to be hoped that software of the kind we have developed here will help to make these tasks easier.

Finally, this paper has described the development and use of PIGs mainly from the point of view of making the best of NLG systems rather like what we already have. The separation of logic and control supported by the PIG architecture could change the way we think about NLG systems in the first place. For instance, a PIG could easily be made to overgenerate (in the manner, for instance, of HALOGEN (Langkilde-Geary 2003)), in the confidence that an oracle could later be devised that appropriately weeded out non-productive paths.

Acknowledgments

This work was supported by EPSRC grant EP/E011764/1. The ideas here have benefited particularly from discussions with Graeme Ritchie and Roger Evans. We also acknowledge the helpful comments of two anonymous reviewers.

References

John Bateman. 1997. Enabling technology for multilingual natural language generation: the KPML development environment. *Natural Language Engineering* 3(1):15-55.

Dimitris Bilidas, MariaTheologou and Vangelis Karkaletsis. 2007. Enriching OWL Ontologies with Linguistic and User-Related Annotations: The ELEON System. *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence* (ICTAI), Patra, Greece.

Albert Gatt and Ehud Reiter. 2009. SimpleNLG: A realisation engine for practical applications. *Proceedings of ENLG-2009.*

Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann and Ian H. Witten. 2009. The WEKA Data Mining Software: An Update; SIGKDD Explorations, Volume 11, Issue 1.

Eduard H. Hovy. 1990. Pragmatics and Natural Language Generation. Artificial Intelligence 43(2), pp. 153–198.

Alon Lavie and Michael Denkowski. 2009. The METEOR Metric for Automatic Evaluation of Machine Translation. Machine Translation, published online 1st November 2009.

Irene Langkilde-Geary. 2003. A foundation for general-purpose natural language generation: sentence realization using probabilistic models of language. PhD thesis, University of Southern California, Los Angeles, USA.

François Mairesse and Marilyn Walker. 2008. Trainable Generation of Big-Five Personality Styles through Data-driven Parameter Estimation. In *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics (ACL)*, Columbus.

Daniel Paiva and Roger Evans. 2005. Empirically-based control of natural language generation. *Proceedings of the 43rd Annual Meeting of the ACL*, pages 58-65.

Ehud Reiter. 2000. Pipelines and Size Constraints. *Computational Linguistics*. **26**:251-259.

Verena Rieser and Oliver Lemon. 2006. Using Machine Learning to Explore Human Multimodal Clarification Strategies. *Procs of ACL 2006.*

Richard S. Sutton and Andrew G. Barto. 1998. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA.

Ross Turner, Yaji Sripada, Ehud Reiter and Ian Davy. 2006. Generating Spatio-Temporal Descriptions in Pollen Forecasts. *Proceedings of EACL06 Companion Volume.*