Blabbeur - An Accessible Text Generation Authoring System for Unity

No Author Given

No Institute Given

Abstract. We present Blabbeur, a generative, context-aware, text generation system for Unity. It provides a simple, accessible context-free grammar inspired syntax allowing conditional generation and the surfacing of variables. Content requests are easily invoked in Unity scripts with relevant variables passed either as property dictionaries or through class interfaces. A persistent testing environment allows authors to quickly test their grammars against different contexts.

Keywords: Text generation · Context-free grammars · Authoring · Unity

1 Another Text Generation System?

In this demo paper, we present *Blabbeur*, a generative, context-aware, text generation system for the Unity game engine [1]. Blabbeur was designed for a multi-member game development team making an emergent narrative game in Unity. For the game, it is necessary to generate text communicating greatly varying world states and events to players, the scale of which prevents hand-authoring. As a result, we sought to use a text generation tool, which would meet the following requirements:
1. Easy to author – Team members without programming background should be able to quickly learn to create content with it.
2. Context-aware – Authors should be able to set conditions qualifying or disqualifying text fragments. They should also be able to surface specific values in the text.
3. Generative – The tool should easily allow for variations in the generated text.
4. Easy to troubleshoot – Authors should be able to quickly test context scenarios without having to wait for them to be naturally occurring in the system.
5. Compatible – It should be easy to communicate system states to the tool and request text generations.
6. Supports collaboration – Multiple authors should be able to contribute simultaneously.

Requirements 1, 2 and 3 are commonly found in research focused text generation systems, namely Expressionist [6], Tracery [3], and STEP [4]. We considered

[1] By the time of publication, the system will be made publicly available with the MIT license

each system as a possibility for our requirements, but each system typically fell short of one or more of requirements. Out of the three, Expressionist was the initial clear choice, being heavily inspired by the existing functionality of Tracery, but also provided the functionality of requirement 2, whereas Tracery does not provide any built in conditional checks, or value surfacing.

The major limitation which was found with Expressionist, is that it is implemented in Python, and would require a code port to Unity's native coding language of C#. This proved non-trivial namely due to Expressionist's use of the EVAL() function to evaluate conditions and effects. EVAL essentially allows the execution of an arbitrary string as though it was a line of Python code, but this function is not available in C# and is non-trivial to implement. STEP, being already implemented in Unity, is an obvious alternative, but the PROLOG-like syntax was found to be overly hard for the non-technical authors to learn.While re-implementing may seem a trivial issue, it nonetheless highlights the different challenges required in the actual use of text generation systems in real world projects. Elements such as cost, differently skilled team members and integration limit the acceptance of research systems in a broader context, eg. while Tracery is considered a successful system, it only became common use with the development of Twitter [2] and Twine [1] integrations. We therefore, in addition to the contribution of the tool itself, present the specifics of the Blabbeur design, in hope that it can help other researchers or practitioners understand the more practical side of tool development.

2 Blabbeur

In this section we present the syntax, system communication and debugging features of Blabbeur. As stated before, the system is not particularly novel for text generation, but rather the focus is on implementation, and the needs of a game development team.

```
// Wiki output for accidents
// Requires: Damage (int), TypeOfAccident (string), victim (human)

Wiki_Accident:  [description] [damage];

damage: and I lost <Damage> health points, leaving me with <victim.healthpoints>.;

description:    {TypeOfAccident == ""} [generic_accident];
                {TypeOfAccident == "agriculture"} [agriculture];
                {TypeOfAccident == "fishing"} [fishing];
                {TypeOfAccident == "hunting"} [hunting];

generic_accident:   I had a freak accident;
                    I messed up big time;
                    I was minding my business, when suddenly [misfortune];

agriculture:        I hurt my [bodypart] with a [farmingtool];
                    I suffered heatstroke from working all day under the sun;
                    I grabbed what I thought was my [farmingtool], but it was a snake. It bit me;
                    I sprained my [bodypart] pulling out weeds;
```

Fig. 1. Excerpt from a Blabbeur grammar file used to generate accident descriptions.

Syntax A Blabbeur grammar is written as a basic text file. This has many advantages, such as not being tied to any particular editor, and being particularly easy to track on source control systems. Syntax was designed to be legible and intuitive, and contains the following components:
Symbols – Grammar symbols are defined as labels followed by colons, the first one being the point of entry (in Fig. 1,"Wiki_Accident"). Non-terminal symbols that need to be resolved are expressed within brackets. In Fig. 1, the generation will first resolve [description] and then [damage]. Multiple outputs of a symbol can be defined by separating them with semicolons. In this case, the [generic_accident] symbol can randomly resolve either as "I had a freak accident", "I messed up big time", or "I was minding…". Symbols can be nested ad infinitum.
Conditional Expressions – Authors can make a symbol conditional by preceding it with an expression in curly braces. Only the symbols whose conditions are met will be considered as possible outputs at the moment of generation. In Fig. 1, for example, the author redirects the resolution of [description] by checking what kind of accident they are dealing with. If the "TypeOfAccident" variable is set to "agriculture", only the second symbol will be considered valid, thus leading to the resolution of [agriculture]. Currently, the following operators are supported: !=, ==, <, >, &&, ||
Variables – Variables that have been passed along the generation request can be used within conditional expressions or surfaced directly within the text. This is done by placing a condition within angle brackets. In Fig. 1, <Damage> and <victim.healthpoints> will be replaced with the variables' values. Values can be numerical, strings, enumerators, or conditionals.
Comments –Any line preceded with "//" are disregarded as comments.
System Communication – The Blabbeur system is implemented as a singleton and is accessible at any point of the project's code. Blabbeur communication is done through requests, passing the name of the top-level symbol of the desired grammar, as well as a custom Blabbeur Object containing an arbitrary set of variables describing the state of the request. The generated text is returned as a string.

```
Blabbeur.Objects.PropertyDictionary blabvars = new Blabbeur.Objects.PropertyDictionary("blabvars");
blabvars.Add("Damage", Damage);
blabvars.Add("TypeOfAccident", TypeOf);
blabvars.Add("victim", actor);
return Blabbeur.TextGen.Request("Wiki_Accident", blabvars);
```

Fig. 2. A Blabbeur content request in C#

Blabbeur Object – A Blabbeur object is a <string,value> dictionary which matches a variable string from the grammar to its appropriate value. It is further possible to "nest" blabbeur objects, eg. the "victim" shown in Fig. 2, is assigned to the actor object, which then contains its own dictionary, eg. name, age, etc.
Blabbeur Interface – Blabbeur objects can also be implemented through a C# interface, where a class can be treated as a blabbeur object by implementing the

required functions of the interface. In Fig. 2, the actor is a "Human" class which implements the Blabbeur interface and is therefore treated as a nested Blabbeur object.

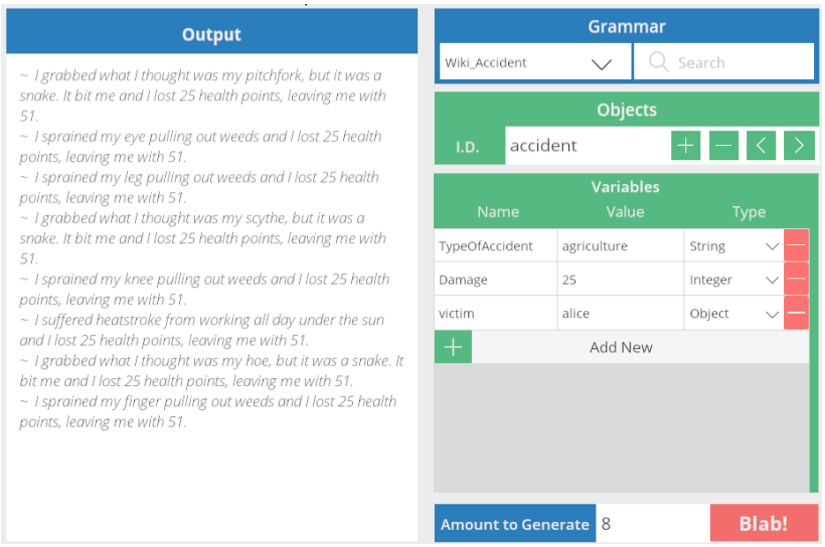

Fig. 3. The Blabbeur test environment

Testing Environment – As underlined in Lessard et al. [5] productivity with generative text tools depends on the availability of a robust testing environment. We have designed for Blabbeur a Unity interface allowing authors to test their grammars against different states. Testing involves selecting a grammar and creating or modifying custom Blabbeur objects to create a number of possible game scenarios.

3 Conclusion

We have been using this system for months now and it has proven easy to use and efficient. With less than an hour of training, authors can immediately begin producing new content. We already have more than 40 grammars in our system and they are a key component in communicating with players. As we stress test Blabbeur, we are also noting requests and comments from authors to increase usability, focusing on elements such as error feedback, and accessing another grammar from within a grammar.

References

1. Balousek, M.R.F.: Twincery. Online, https://github.com/mrfb/twinecery, last accessed 2021/06/07
2. Buckenham, G.: Cheap bots done quick. online, last accessed 2021/06/07
3. Compton, K., Kybartas, B., Mateas, M.: Interactive Storytelling, chap. Tracery: An Author-Focused Generative Text Tool, pp. 154–161. Springer International Publishing, Cham (2015)
4. Horswill, I.: Generative text using classical nondeterminism. In: Joint Proceedings of the AIIDE 2020 Workshops. Worcester, MA (2020)
5. Lessard, J., Brunelle-Leclerc, E., Gottschalk, T., Jetté-Léger, M.A., Prouveur, O., Tan, C.: Striving for author-friendly procedural dialogue generation. In: Proceedings of the International Conference on the Foundations of Digital Games - FDG '17. ACM Press (2017). https://doi.org/10.1145/3102071.3116219
6. Ryan, J., Seither, E., Mateas, M., Wardrip-Fruin, N.: Interactive Storytelling, chap. Expressionist: An Authoring Tool for In-Game Text Generation, pp. 221–233. Springer International Publishing, Cham (2016)