DIALOGIC: A Toolkit for Generative Interactive Dialog

Daniel C. Howe
School of Creative Media
City University of Hong Kong, daniel@rednoise.org

ABSTRACT

Open-ended narrative games present vast surfaces that require large amounts of compelling writing. Automated solutions have made little progress in this domain and talented human writers who code are few and far between. Thus the question of how to augment writers with digital tools—without requiring them to become programmers, or to need continual assistance from programmers—is a crucial one for the field. To address this question, we present *Dialogic*[1], a scripting language, execution environment, and set of online tools designed to support skilled human authors in creating engaging interactive writing for games, leveraging generative strategies atop a simple syntax familiar to writers. We describe the system's goals and motivations, architecture, and technical details, and evaluate its use in two production-quality titles.

KEYWORDS

Natural language, generative language, dialog systems, interactive narrative, digital literature

ACM Reference format:

Daniel C. Howe. 2020. DIALOGIC: A Toolkit for Generative Interactive Dialog. In *International Conference on the Foundations of Digital Games (FDG '20), September 15-18, 2020, Bugibba, Malta.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3402942.3402993

1 Introduction

As natural language-based interfaces for computing (e.g., Alexa, Siri, etc.) have become cheap and ubiquitous, computational dialog systems have enjoyed new popularity among researchers. Yet most focus on the accomplishment of specific tasks by users: selecting music, ordering coffee, or answering a technical question. Less attention has been paid to subjective aspects of human communication, and still less to literary outputs. One might argue that we are hard at work training systems to enact the least interesting elements of human communication. One explanation for this concerns the magnitude of the writing surface that interactive contexts present. Even simple interactive applications can include an overwhelming number of paths for which writing is required. In complex, open-ended experiences, the space can be truly vast; a fact that makes these contexts exciting to engage with, but difficult to write for.

Contrary to media reports detailing how 'deep-learning' will replace all sorts of human labor, there is no easy computational solution here. Even with significant research in the area, creative writing remains one of a number of human activities at which AI has made little progress. The human writing process, though highly resource-intensive, far exceeds the best AI efforts thus far. It may be this very tension, between the compelling vastness of open-ended, interactive spaces on the one hand, and the need for highly-skilled, resource-intensive human labor to populate them with writing on the other, that is the central problem facing would-be creators. For tool-makers then, the challenge of how to engage computation, not as a substitute for, but as an amplification of, human writerly power, becomes critically important. As Ryan et al. say,

> Skilled writers who can produce stylistically rich and evocative text should be the ones writing videogame dialogue, but people who are both skilled writers and NLG (natural language generation) experts are rare. As such, we require approaches in which authors without procedural backgrounds can still be largely responsible for the production of dialogue... while somehow harnessing a nontrivial degree of computer generativity (Ryan et al, 2015)

As noted, generativity is an important potential strategy for resolving the tension described. Yet the question of how to integrate even simple generative procedures in a context that is friendly and engaging to traditional writers is an open one. In this paper, we present our efforts at addressing this question through *Dialogic*, a free, open-source scripting language and integrated set of support tools for generative interactive dialog. *Dialogic* is designed for human writers, rather than for programmers or generative artists, to craft compelling dialog that responds organically to user prompts and events in the environment. The system supports arbitrarily complex interactions between multiple actors, but makes no assumptions about how text is displayed, or how users will choose their responses. Below we describe the system's design goals and motivations, architecture and technical details, and analyze its use in two award-winning titles, *TendAR* and *Spectre*.

FDG '20, September 15-18, 2020, Bugibba, Malta

ACM ISBN 978-1-4503-8807-8/20/09...$15.00
https://doi.org/10.1145/3402942.3402993
[1] Dialogic can be found at https://github.com/dhowe/dialogic

2 Motivations

It has been noted that interactive games with natural language form a particularly challenging context for those interested in literary language. As Mateas stated over a decade ago,

> Current authoring practice for videogame dialogue production, in which individuals or even teams of writers tirelessly produce huge amounts of content by hand, is largely seen as both untenable and constraining of the form. (Mateas, 2007)

The problem is magnified when the experience for which dialog is required spans a very large space of possible paths:

> [A]uthorial burden grows monotonically with a system's state space... the more states a system can get into, the more states that must be expressible by the system, and thus the more content that must be authored for it. So while, for instance, games with huge state spaces are often lauded for this very enormity, that same property yields a serious authoring challenge... This issue is best alleviated by more content, but for games with particularly massive state spaces, it is not feasible to author by hand as much content as would be needed. (Ryan et al, 2016)

How then to realize generative variation across interactive experiences with large state spaces so that dialog is both engaging and contextually specific? The answer to this question, we believe, involves three distinct elements. First, we must support a wide range of writers, including those without a technical background, to easily create dialog for the system. Second, writers should be able to organically intermix generative elements--from simple variation, to context-free-grammars, to more advanced techniques—if/as they see fit, without changing tools. Third, we must provide a mechanism by which the appropriate fragments of dialog are triggered at the correct time, customized for the current state of the system. The following sections describe how each of these elements were realized in *Dialogic*.

To begin answering this question we designed the *Dial* scripting language to allow a range of writers, including those without a technical background, to easily create dialog for the system. They can write static text and/or organically integrate generative elements; from simple variation, to context-free-grammars, to more advanced techniques. To locate and trigger appropriate dialog at the appropriate moment, customized for the context, we developed a fuzzy-search algorithm that sits at the core of *Dialogic's* scheduler. Below we describe our implementation of these components, as well as the tools and functionalities developed to support them.

3 Scripting Dial

In order to enable use by a diverse range of writers with varying degrees of technical skills, we created a scripting language called *Dial*. In *Dial* strings of text are first-class objects. This means that, unlike many languages, strings are not delimited by quotation marks, but can simply be typed in the manner with which writers are familiar. Beyond convenience, this decision foregrounds natural language as the core material of the system. *Dial* also mimics the way dialog is traditionally written in screenplays. For example, the following is perfectly correct *Dial* script:

```
Lance: If you are all right, then say something.
Mia: Something!
```

3.1 Dial Commands

In *Dial*, each section of text is known as a CHAT. Each chat has a unique name and contains a number of commands. When a chat is run, each command is executed in order, until all have been run, or the system branches to a new chat. The simplest command is SAY which simply echoes the given output:

```
SAY Welcome to your first Dial script!
```

As mentioned, quotations are not needed around strings. In fact, the SAY keyword itself is even optional. So the simplest *Dial* script is just a single line of text:

```
Welcome to your first Dial script!
```

Commands generally begin a line. Common commands include SAY, DO, ASK, OPT, FIND, SET, and others. Here is a longer example:

```
CHAT Start
SAY Welcome to my world
WAIT 1.2
SAY Thanks for Visiting
ASK Do you want to play a game?
OPT Sure
OPT No Thanks
```

This chat is called "Start" and performs a few simple functions; welcoming a visitor, pausing, then prompting for a response.

Of course in most cases we would want to do something with this response. In the code below we branch, based on the user's response:

```
ASK Do you want to play a game?
OPT Sure #Game1
OPT No Thanks
```

If the user selects the first option, the system jumps to the chat named "Game1". If not, the current chat continues.

3.2 Generative Variation

Dialogic is designed to smoothly blend scripted and generated text to create the type of variation found in natural-sounding dialog. The simplest way to include generative elements in a response is via the OR operator (the 'pipe' character), grouped with parentheses as follows:

```
SAY You look (sad | gloomy | depressed).
```

Elements between the | operators are randomly selected, so the line above will generate each of the following 3 outputs with equal probability:

```
You look sad.
You look gloomy.
You look depressed.
```

Writers may also specify weightings for various choices, as well as favoring choices that have not been recently selected. Another example, demonstrating nested OR constructions:

```
SAY I'm (very | super | really) glad to ((meet | know) you | learn
about you).
```

One can also save the results of an expansion for later use. For example, if one wanted to pick a character name that would then be reused several times in a paragraph:

```
SAY Once there was a girl called [hero=(Janice | Mary)].
SAY $hero lived in [home=(Neverland | Nowhereland)].
SAY $hero liked living in $home.
```

Possible outputs would include:

```
Once there was a girl called Janice.
Janice lived in Neverland.
Janice liked living in Neverland.
```

and

```
Once there was a girl called Mary.
Mary lived in Nowhereland.
Mary liked living in Nowhereland.
```

One could also use the SET command to similar effect:

```
SET hero = (Janice | Mary)
SET home = (Neverland | Nowhereland)
SAY Once there was a girl called $hero
SAY $hero lived in $home.
SAY $hero liked living in $home.
```

Dial script also supports the use of system variables which are expanded at runtime, based on their current value:

```
Mia: What a nice $user.shirt.color top you have on.
```

3.3 Transformations

Further adjustment of utterances is often required based on context. Examples include conjugation, pluralization, pronoun realization, and a range of other cases in which the exact form of a textual element depends on the surrounding language. To address this, we provide "transforms", simple functions that work on elements of *Dial* script before they are output. Built-in transforms include pluralize(), articlize(), capitalize() and others, which can be called from *Dial* scripts as follows:

```
Lance: How many (tooth | menu | child).pluralize() do you have?
```

The output would be one of the following:

```
How many teeth do you have?
How many menus do you have?
How many children do you have?
```

Or

```
Mia: Are you (dog | cat | ant).articlize()?
```

which gives:

```
Are you a dog?
Are you a cat?
Are you an ant?
```

Transforms can also be applied to variables created with the SET keyword:

```
SET choices = (tooth | menu | child)
How many $choices.pluralize() do you have?
```

Or to parenthesized words or phrases

```
How many (octopus).pluralize() do you have?
```

String functions from the host language (C# or JavaScript, for example) can also be included:

```
SET choices = (tooth | menu | child)
How many $choices.toUpperCase() do you have?
```

And one can arbitrarily chain multiple transforms:

```
SET choices = (tooth | menu | child)
How many $choices.pluralize().toUpperCase () do you have?
```

To extend the system, custom transforms can be added programmatically (and then later called from *Dial* scripts).

Dial's SET command supports commonly used formalisms such as context-free grammars (CFG), generative algorithms used in popular text frameworks like Tracery (Compton et al, 2015), and RiTa (Howe, 2009). The following CFG from Tracery,

```
{
color: ["red", "green", "indigo", "ecru", "violet"],
animal : ["panda", "ocelot", "meerkat", "platypus"],
mood: ["joyful", "morose", "alert", "sleepy", "pensive"],
pet: ["puppy", "#mood# kitty", "#mood# #color# #animal#"],
tale: ["This is the story of a #pet#..."]
}
```

can be written in *Dial* as

```
SET color = red | green | indigo | ecru | violet
SET animal = panda | ocelot | meerkat | platypus
SET mood = joyful | morose | alert | sleepy | pensive
SET pet = puppy | $mood kitty | $mood $color $animal
SET tale = This is the story of a $pet
```

4 The Dialogic Workbench

The second core component of *Dialogic* is the online editor interface, which allows creators to easily author with the system, either working bottom-up (editing, validating, and testing individual *Dial* scripts, with syntax highlighting, linting, and intuitive error messages), or top-down (crafting larger narrative arcs through a drag-and-drop environment). Figure 1 shows the online workbench environment, which consists of the network view (to left) and the editor view (to right). In the network view, authors craft dialog by connecting and sequencing existing chats. In the editor window, authors write, edit, and verify chats. Once a chat is written, it can be interpreted and executed directly in the online interface, either alone or in conjunction with other chats, in order to view the range of possible outputs.

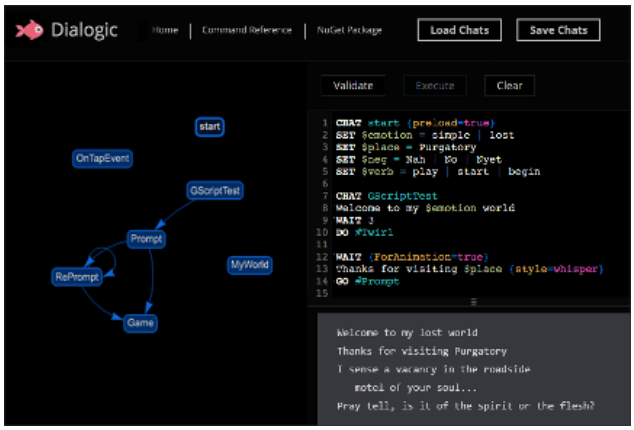

Figure 1: The Dialogic Workbench

The workbench is designed to maximize ease-of-use for authors, allowing authors to intuitively (1) enter new content; (2) predict and check runtime behavior; (3) identify, and fix errors, (4) distinguish easily between language elements (commands, text, metadata) and (5) to obtain a high-level view of the connections between elements of dialog.

5 The Dialogic Runtime

The third core component of *Dialogic* is the runtime environment, which handles the interpretation scheduling of *Dial* scripts in real-time interactive contexts. In addition to parsing *Dial* scripts into code to be run in the host language, the

runtime passes messages between the various components (the script-parser, state-manager, dialog-scheduler, etc.) so that dialog can be composed and output as needed in response to game events. The system includes components that deal with chat selection, symbol and choice resolution, as well as snapshots, repetition, and interruption/smoothing.

5.1 Constraint-based Fuzzy Search

To facilitate efficient location and triggering of context-appropriate dialog, *Dialogic* includes a mechanism we call Constraint-based Fuzzy Search (CFS). CFS allows the system to avoid the deeply nested conditionals associated with large branching narratives, especially as content is added over time[2]. Consider a typical use-case where a greeting is to be chosen based on the runtime 'strength' property of the user. In a conditionally-oriented language we might expect something like the following:

```
if ($user.strength >= .5) {
    nextChat = "strong-char-greeting";
}
else {
    nextChat = "weak-char-greeting";
}
scheduleNextChat(nextChat);
```

This approach seems fine enough until we consider that the context may be a large-scale game under development with many thousands of branches for which dialog is being written.

Imagine a scenario in which, initially, we have only the two bits of dialog, one for strong characters, and one for weak characters, as above. What happens when later writers wish to add further nuance, so that the greeting depends on the character's hunger as well? Programmers would now likely need to go back into the game and add more conditionals for when the character is strong and hungry, strong and not-hungry, weak and hungry, and weak and not-hungry. For each new property that we wish to consider, the number of conditionals (at least) doubles.

To avoid such conditional explosion, CFS can be employed to search for the chats most closely matching a set of constraints[3]. When written, chats can be annotated with metadata tags that list the properties on which the chat is conditioned. These properties are then evaluated at runtime. Instead of changing program code as new dialog is added, we simply add appropriate metadata tags to the dialog itself.

Imagine that when we start we have only one generic greeting, as follows:

```
CHAT greeting
SAY Hello $user.name
```

To transition to that chat, we use the Find command to specify that the runtime should jump to the specified chat before continuing:

```
FIND {label=greeting}
```

When writers later add a variable greeting based on the "strength" property, something like:

```
CHAT strong-greeting { $user.strength >= .5 }
SAY Hello $user.name, you look strong today!
CHAT weak-greeting { $user.strength < .5 }
SAY Hello $user.name, not feeling well?
```

We can simply change our call to the following (specifying that the name of the chat's label must contain the string 'greeting'):

```
FIND {label*=greeting}
```

Each of the three listed chats will match this constraint, but one of the other two will also match its user strength constraint (depending on that property's value when the search is executed), therefore it will score higher and be returned first.

Now let us say that we want to add a new variant for when the user is weak and also hungry. Here we simply add the new chat, and it will be automatically triggered in the correct cases, with no further code changes:

```
CHAT weak-hungry-greeting { $user.strength <.5, $user.hunger >.7 }
SAY $user.name, you're looking a bit pale, better find some food!
```

If 'user.hunger' isn't above .7, then we automatically fall back to our 'weak-greeting' chat. In this way we can gradually layer more customized (and more specifically-targeted) dialog as it is created by writers. In fact, new dialog can even be injected at runtime (via a game update or upgrade, for example) and it will organically integrate into the game, in most cases without requiring any code changes at all.

More abstractly, the FIND command accepts any number of metadata key-value pairs specifying constraints to match. The highest scoring CHAT that does not violate any of the constraints is located, and the system then branches to this CHAT.

Normally CHATs will match a constraint if it is present in their own metadata and its value matches, or if they do not have the given key in their metadata . Prepending the strict operator (!) to a key signifies that the key MUST be both present and matching. To allow more precise control over the fuzzy search, FIND allows comparison operators in its metadata, including: >, <, >=, <=, != (not equal), ^= (startsWith), $= (endsWith), and *= (matches).

Consider the following examples:

```
FIND {day=monday,level=10,emotion=happy}
FIND {day!=monday,level>=10,emotion^=h}
FIND {day!=monday,level<10,emotion*=(happy|excited)}
// if 'emotion' exists, it must be 'happy'
FIND {emotion=happy}
// 'emotion' MUST exist and must be 'happy'
FIND {!emotion=happy}
```

Note that a CFS will not return an empty set. Instead it will relax the given constraints until a match is found. However, if this is not the desired behavior, prepending the double-strict operator (!!) to a key will ensure that the constraint will never be relaxed.

```
// 'emotion' MUST exist, must be 'happy' and will never be relaxed
FIND {!!emotion=happy}
```

5.2 Interruptions / Smoothing

Dialogic also responds naturally to user interaction and/or interruption. This is enabled via a stack mechanism that stores recently executed chats. When an event or other interruption occurs, the response chat is pushed atop the stack and the

[2] A similar idea was proposed by Elan Ruskin, an engineer at Valve games. For more information, see Ruskin, 2012.

[3] Following reasoning often advanced by proponents of object-oriented programming (OOP), *Dial* does not include any conditional statement. Rather than via search, the conditional is often avoided in OOP via *polymorphism,* the dynamic dispatch of method calls to objects based on their runtime type.

current chat marked as 'interrupted'. When the response chat is finished, control moves to the next interrupted chat on the stack. Smoothing sections can be added in order to make transitions more natural, i.e., 'so as I was saying'.

To add smoothing to a chat, writers use the 'onResume' metadata tag, specifying either the label of the smoothing chat, or a set of FIND constraints to use in locating it. In the example below, each time the 'Long' chat is interrupted, 'Smooth2' will be triggered before it resumes once again.

```
CHAT Long {onResume=Smooth2}
SAY Oh, it's you...
SAY It's been a long time. How have you been?
...
SAY You get what I'm saying?

CHAT Smooth2 {noStart=true}
SAY Where was I? Oh, yes
```

5.3 Repetition Detection

For generative interactive systems to be compelling they must be able to recognize and avoid exact and inexact repetitions. *Dialogic* handles repetition avoidance at multiple levels. At the micro-level individual substitutions will never repeat. In the following example,

```
Lance: If you're ok, then (say|shout|yell) something.
```

Lance will never repeat any of the three options. At a higher level, each selected utterance is compared, via a Minimum-Edit Distance metric, to recently spoken utterances within the runtime's chat history. If the selected resolution is too close to one in the recent history, its score will be discounted and other chats may be used instead, assuming they do not violate any of the strict constraints of the search.

5.4 Snapshots

As interactive systems must be able to gracefully pause, restart, and update, *Dialogic* includes a sophisticated mechanism for snapshots that stores the state of all chats and the system itself across multiple suspends and restores, as well as allowing for the merging of multiple snapshots.

6 Integrating Dialogic

Dialogic can be run alone or with a game engine such as Unity3D. In the C# example below, a runtime is created that reads in chat descriptions from a plain-text file (or folder) and parses them into a list of chats to be executed. The runtime's Run() function is called to start execution, specifying the name of the chat with which to begin.

The application calls the runtime's Update() function each frame, passing the current world-state (a dictionary of key-value pairs) and any events that have occurred since the last frame.

```
public RealtimeGame()
{
    dialogic = new ChatRuntime();
    dialogic.ParseFile(fileOrFolder);
    dialogic.Run("#StartChat");
}
public IUpdateEvent Update() // Game Loop
{
    // Call the dialogic interface
    evt = dialogic.Update(worldState, ref gameEvent);
    // Handle the event received
    if (evt != null) HandleEvent(evt);
    ...
}
```

6.1 Extensibility

When designing dialog systems it is important to allow for functionalities that designers have not yet imagined. Toward this end we have included transforms (as discussed above), as well as custom commands and validators, with which *Dialogic* can be extended. To create a custom command, one must first create a subclass of Dialogic.Command that implements the functionality needed. Then the name and class definition are registered as part of the configuration object, as follows (in C#):

```
myConfig = new MyAppConfig();
myConfig.AddCommand("CUST", typeof(myNamespace.MyCustomCommand));
runtime = new ChatRuntime(myConfig);
```

Custom commands are treated exactly as the primitives (CHAT, SAY, FIND, SET, DO, ASK, OPT, WAIT, GO[4]) included with *Dial*, parsed via the interpreter and executed by the scheduler. Validators, also optionally registered with custom commands, perform any necessary verification of the data passed to these commands.

7 Evaluation

While there are evaluation metrics that might be applied to *Dialogic* itself, we subscribe to the position that new dialog systems cannot be truly appraised except through implemented experiences that are built atop them (Ryan et al, 2015, pg 3). *Dialogic* has been used in two production-quality projects thus far. The first is *TendAR*, released for Android devices in November 2018 on the Google Play Store. To develop the Unity-based, augmented-reality game, ten writers, with widely varying degrees of technical skill, used the *Dialogic* system to realize the game's goal of "applying long form interactive storytelling to the emerging field of augmented reality". As *Dialogic* was developed and refined in conjunction with the development of *TendAR*, writers and engineers on the project were instrumental in developing the system, specifically the fuzzy search functionality for which the implementation owes a great deal to Tender Claws founder, Danny Canizzaro. In *TendAR*, users engage with a virtual fish that feeds, via realtime emotion detection, on laughter, surprise, sadness and other emotions. The AI fish can recognize and respond to hundreds of objects, building up knowledge as it interacts with the user and, through the camera, the physical world. Users help to bring the fish from a non-verbal AI to a unique creature with its own personality. As Canizzaro says[5],

> Dialogic enabled our game to achieve the possibility of having three weeks of written content that could be executed generatively. This amount of content is very rare for existing AR apps. Many games writers are still stuck on a branching model; Dialogic has the potential to encourage a model of writing that provides new choices to players by opening up generative and responsive dialog.

[4] For a full description of each command, see https://github.com/dhowe/dialogic/wiki/Command-Reference
[5] From email correspondence

In the months following its release *TendAR* garnered a number of awards and accolades, including the IndieCade "Innovation in Interaction Award" and selection in the Sundance Festival's New Frontiers category, which "spotlights work at the dynamic crossroads of film, art and technology," and was an Official Selection of the IDFA Doc Lab.

The second test case for *Dialogic* was *Spectre*, by the author in collaboration with artist Bill Posters. Winner of the 2019 Alternate Realities Commission, *Spectre* is an interactive installation that reveals the secrets of the Digital Influence Industry as visitors "pray to Dataism and the Gods of Silicon Valley". The artwork engages users in a personalised journey that tells a cautionary tale of computational propaganda, technology and democracy, curated by an algorithm, and powered by visitor data. In *Spectre*, *Dialogic* was used to generate descriptive text passages of visitors based on their OCEAN behavioural profile, a series of metrics used in behavioural advertising and micro-targeted political campaigns. The following text was generated by the system:

> Jeanine wasn't always how she was; the kind of girl who could make the worst type of decision without even a hint of hesitation. And she was certainly not afraid to point out when someone went off, no matter if the others didn't know it yet or otherwise. In her view, life was hard enough without, as her uncle used to say, asking the devil in for hotcakes. She might overreact from time to time, sure... but was that worse than the consequences? Not under the present circumstances.

Spectre was shortlisted for the Aesthetica Art Prize and the Digital Dozen Breakthroughs in Storytelling Awards, and was selected as the first AI-based work to be included in the British Film Archive. It is scheduled to tour Europe and North America later in 2020, with venues including the Baltic, ZKM Karlsruhe, and SXSW.

8 Prior Work

A large number of projects have taken steps toward the integration of natural language generation into games, but few have resulted in commercially-released titles with generative dialog (Ryan et al, 2016). Walker and collaborators have explored the integration into games of traditional NLG pipelines proceeding from character models (Walker et al. 2013) as well as symbolic content representations (Lukin et al, 2014; Antoun et al. 2015). Such systems have resulted in the ambitious *SpyFeet* prototype (Reed et al, 2011). The commercially-released *Bot Colony* (Joseph 2012) employs a fully-realized NLG pipeline—a first for a commercial title. Like Ryan (2015), we have avoided the complexity of traditional NLG and instead utilize a generative grammar-based approach. As argued elsewhere, NLG systems incur an authorial burden that can outweigh the benefits of their generativity. Furthermore, such systems may not be reliable enough in terms of content quality to be used in commercial releases. Perhaps most importantly, such complex systems are unlikely to be approachable by writers without technical backgrounds.

Like *Dialogic*, a number of previous systems have featured templated dialogue annotated with metadata, such as *Curveship* (Montfort 2009), *Prom Week* (McCoy and others 2013), *Versu* (Evans and Short 2014), *Event[0]* (Mohov 2015) *Expressionist* (Ryan et al, 2016), and *LabLabLab* (Lessard 2016). This approach allows authors to include variables in lines of dialogue that the system resolves only as they are displayed. Like *Expressionist*, *Dialogic* supports variables that may resolve to expressions that themselves have variables in them (and so forth recursively). Finally, *Dialogic* is influenced both by the grammar-based story-authoring tool *Tracery* (Compton et al, 2014), and by *RiTa* (Howe, 2009), a toolkit for generative natural language, which includes grammars, Markov chains, and other means of language generation.

9 Conclusions & Future Work

Our experiences using *Dialogic* as the authoring system for *TendAR* and *Spectre* suggest that, with appropriate tool support, traditionally-oriented writers can in fact leverage simple generative techniques to better meet the demands of large interactive writing surfaces. As we imagine that the popularity and demand for such systems will continue to grow, tools focused toward this user group may prove particularly useful in advancing the field. Additionally we have seen that writers, once accustomed with a system employing a familiar syntax, are quite capable of experimenting with simple generative mechanisms. In fact, one traditionally-trained writer on the *TendAR* team actively explored the full set of generative possibilities offered by the system, even suggesting new extensions to the *Dial* language, all with no prior coding experience. Such anecdotal evidence also supports the argument that generative writing in the context of games and other creative contexts may be a productive avenue for education in computational thinking and procedural literacy (Howe, 2009).

In the coming months we will complete our port of the system from C# to JavaScript and release a formal *Dial* grammar specification so that it can be easily implemented in additional languages. As both *TendAR* and *Spectre* are applications that do not fully test the multi-actor capabilities of the system, we are seeking partners for a long-form game or interactive experience featuring multiple NPCs. Additionally we imagine that *Dialogic* could integrate smoothly into creative applications targeting so-called 'Home Assistants' such as Amazon's Alexa or Google's Home, another fertile area for exploration.

ACKNOWLEDGMENTS

The author would like to extend special thanks to Danny, Sam, Tanya, Jacob and the rest of the brilliant team of writers and coders at Tender Claws.

REFERENCES

Antoun, C.; Antoun, M.; Ryan, J. O.; Samuel, B.; Swanson, R.; and Walker, M. A. 2015. “Generating natural language retellings from Prom Week play traces”. Proc. PCG.

Compton, K., Kybartas, B., and Mateas, M. 2015. "Tracery: an author-focused generative text tool." In International Conference on Interactive Digital Storytelling, pp. 154-161. Springer, Cham.

Evans, R., and Short, E. 2014. “Versu—a simulationist storytelling system.” In Computational Intelligence and AI in Games.

Howe, D. C. 2009. RiTa: creativity support for computational literature. In Proceedings of the seventh ACM conference on Creativity and cognition (C&C '09). ACM, New York, NY, USA, 205-210.

Joseph, E. 2012. “Bot colony—a video game featuring intelligent language-based interaction with the characters.” In Proc. GAMNLP.

Lessard, J. 2016. “Designing natural-language game conversations.” In Proc. DiGRA-FDG.

Lukin, S. M.; Ryan, J. O.; and Walker, M. A. 2014. “Automating direct speech variations in stories and games.” In Proc. GAMNLP.

Mateas, M. 2007. "The authoring bottleneck in creating AI-based interactive stories [panel]." In Proc. INT. 2007.

McCoy, J., et al. 2014. “Social story worlds with Comme il Faut”. In Computational Intelligence and AI in Games.

Mohov, S. 2015. “Turning a chatbot into a narrative game: Language interaction in Event[0]”. In nucl.ai.

Montfort, N. 2009. “Curveship: An interactive fiction system for interactive narrating.” In Proc. CALC.

Reed, A. A., et al. 2011. “A step towards the future of role- playing games: The SpyFeet mobile RPG project.” In Proc. AIIDE.

Ruskin, E. 2012. “AI-driven Dynamic Dialog,” presented at Game Developers Conference (GDC) 2012, Moscone Center, San Francisco, CA.

Ryan, J. O., Mateas, M., and Wardrip-Fruin, N. 2015. “Open design challenges for interactive emergent narrative.” In International Conference on Interactive Digital Storytelling, pp. 14-26. Springer, Cham.

Ryan, J. O., Fisher, A. M., Owen-Milner, T., Mateas, M., and Wardrip-Fruin, N. 2015. “Toward natural language generation by humans.” In Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference.

Ryan, J. O., Mateas, M., and Wardrip-Fruin, N. 2016. “Characters who speak their minds: Dialogue generation in Talk of the Town.” In Twelfth Artificial Intelligence and Interactive Digital Entertainment Conference..

Walker, M. A.; Sawyer, J.; Jimenez, C.; Rishes, E.; Lin, G. I.; Hu, Z.; Pinckard, J.; and Wardrip-Fruin, N. 2013. “Using expressive language generation to increase authorial leverage.” In Proc. INT.