The AIIDE-17 Workshop on
Intelligent Narrative Technologies
WS-17-20

# Analyzing Expressionist Grammars by Reduction to Symbolic Visibly Pushdown Automata

**Joseph C. Osborn, James Ryan, Michael Mateas**
Computational Media
University of California, Santa Cruz
1156 High St
Santa Cruz, CA 95064

## Abstract

We extend the Expressionist project, and thereby the re-emerging area of grammar-based text generation, by applying a technique from software verification to a critical search problem related to content generation from grammars. In Expressionist, authors attach tags (corresponding to pertinent meanings) to nonterminal symbols in a context-free grammar, which enables the targeted generation of content that expresses requested meanings (i.e., has the requested tags). While previous work has demonstrated methods for requesting content with a single required tag, requests for multiple tags yields a search task over domains that may realistically span quintillions or more elements. In this paper, we reduce Expressionist grammars to symbolic visibly pushdown automata, which allows us to locate in massive search spaces generable outputs that satisfy moderately complex criteria related to tags. While the satisficing of more complex tag criteria is still not feasible using this technique, we forecast a number of opportunities for future directions.

## Introduction

Grammar-based text generation is enjoying a small renaissance. The mostly context-free formalism Tracery has spurred the creation of new communities of grammar authors (Compton, Kybartas, and Mateas 2015), and Expressionist and its tagged-grammars approach is being used in a variety of videogame projects (Ryan et al. 2016b; Lessard et al. 2017) and in the middleware technology of Spirit AI (Spirit AI 2017). Beyond applications of Tracery and Expressionist, others are employing grammar-based approaches in games and interactive storytelling, too (Horswill 2014; Mawhorter 2016; Dias 2016b; Togelius, Shaker, and Dormans 2016; Lewis 2017). As an alternative to conventional natural language generation pipelines—the dominant approach in mainstream text generation—grammars are appealing because they are easier to author (especially for writers who do not code) while still being very generative (due to an inherent combinatorial explosion).[1] The core appeal of Expressionist in particular, the approach we focus on in this paper, is that its tagging mechanism enables *targeted generation* (at runtime, content can be requested by specifying the meanings it should express) and *content understanding* (the meanings expressed by generated content are known, which means a larger system can understand and act on generated content). These features are combined in an easy-to-use authoring environment: Expressionist has been used by a number of non-programmer authors in intensive videogame development projects (Ryan et al. 2016b; Lessard et al. 2017). While combining these features with ease of authoring provides significant expressive power, an unfortunate hazard has emerged, ironically, from the very generativity of the grammar underpinnings. Due to the rapid combinatorial growth of generable spaces yielded by grammar authoring, where a day of authoring can yield billions or more outputs (Ryan et al. 2016b), targeted generation becomes a search challenge: even when the meanings expressed by each and every generable output are known, a space of quintillions or more may have to be searched to retrieve a meaning that is requested at runtime. Moreover, things are further exacerbated at authoring time, where one may seek to analyze an authored grammar for higher-order considerations beyond the meanings of individual outputs. This requires more complex queries over huge content spaces, which means this kind of design support currently manifests as an even more significant search challenge.

Thus, the idea that we can analyze a tagged grammar and make claims about the sorts of outputs it is likely to generate—or whether it can produce outputs that have certain meanings—is important both for expressively constrainable content generation at runtime *and* to support grammar authors at design time. As grammars become more complex and as we hope to use them to address more dynamic situations, it may not be obvious whether a grammar could produce, for example, a line of dialogue where a character says their name twice in the course of one introduction. We may also want to know how probable a given output is in the context of the grammar's total generable space, or to visualize a histogram of the possible lengths of all outputs, as in Fig. 1. At runtime, we will want to furnish generated content with the right meanings at the right time. These design questions and runtime considerations may not come up for small grammars, where exhaustive search is feasible, but we have discovered in this renaissance of grammar-based text generation that typical grammars authored for use in real ex-

[1]Elsewhere, we more extensively situate the grammar-based approach against conventional techniques (Ryan, Mateas, and Wardrip-Fruin 2016).

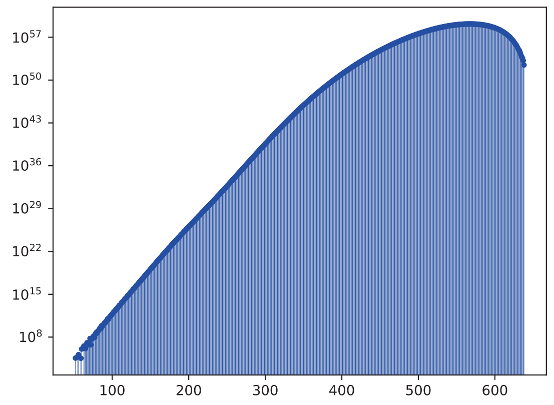

Figure 1: A histogram of grammar output count (log-scale) by output length. This grammar has $10^{64}$ possible outputs.

pressive applications will rarely or never be small (Ryan et al. 2016b).

The problem, again, is a search problem that emerges from the naturally massive generable spaces yielded by the grammar-based approach to text authoring. To solve this problem, we need to reduce these huge spaces to smaller ones that can more efficiently be operated over.

Fortunately, our community is not alone in using formalisms like grammars to describe objects of interest. The area of *software verification* uses equivalent flavors of automata to great effect in formulating and answering questions including whether a regular expression might fail to catch a dangerous class of user input. As it turns out, we can borrow many of their techniques and algorithms by considering grammars as something like a program with function calls and returns, which lets us directly analyze their structures rather than their huge generable spaces.

In this paper, we investigate and compare a family of techniques for controlling and analyzing Expressionist grammars, using actual authored examples as test cases. First, we *reduce* these grammars to *symbolic visibly pushdown automata* (D'Antoni and Alur 2014), and then we use off-the-shelf algorithms to analyze those automata. We evaluate this approach on small, medium, and large grammars, with test cases of varying difficulty. While this paper represents significant steps toward unrestricted targeted generation from tagged grammars, providing a set of explicit performance baselines, some persisting performance issues illuminate opportunities for future improvements.

## Related Work

Beyond applications of Tracery and Expressionist, recent projects by Horswill, Mawhorter, and Dias, respectively, have also utilized generative grammars for expressive text generation (Horswill 2014; Mawhorter 2016; Dias 2016b). By employing generative grammars, these systems harness the power of *templated dialogue*, a pattern used in *Prom Week*, Versu, the LabLabLab trilogy, *Event[0]*, and various works of interactive fiction (McCoy and others 2014; Evans and Short 2014; Lessard 2016; Mohov 2015; Short 2014). One account of these approaches is that they forego the power of conventional natural language generation pipelines to emphasize authorability and expressivity above other concerns. In turn, our current project aims to further improve authorability by providing novel design support at authoring time (authors form offline queries about what meanings generable content can express) and to further improve expressivity by providing novel generation support at runtime (a system requests generated content in terms of the meanings it ought to express). There unfortunately has not been much work on design support for procedural text authors, but Garbe et al.'s work on visualizing the combinatorial content space of the interactive narrative *Ice-Bound* is a touchstone in this area (Garbe et al. 2014). Emily Short has also written about approaches to and prospects for visualizing the combinatorial text spaces yielded by generative grammars (Short 2016b; 2016c). Our approach, which supports queries about the array of meanings that generable content can express, can be thought of as a method for authors to investigate the *expressive range* of their text generators. The notion of expressive range is due to Smith and Whitehead (Smith and Whitehead 2010), and recent work by Cook et al. has explored design support via visualizing expressive range (Cook, Gow, and Colton 2016). With regard to runtime concerns for procedural text, Short and Dias have individually articulated the need for generative text that expresses salient meanings, rather than content yielded by random sampling from a generable space (Short 2016a; Dias 2016a). In procedural generation more broadly, this relates to the *10,000 Bowls of Oatmeal* problem articulated by Compton (Compton 2016) and expanded on by Cook (Cook 2016) and Cardona-Rivera (Cardona-Rivera 2017), in which a generative system can fall prey to producing an astronomical number of content variations that do not provide salient or meaningful differences to a human observer. The design of Expressionist inherently confronts this issue by allowing authors to efficiently index their generable spaces according to prominent authorial concerns, which means distinctions in the meaning of generable content will be reified as distinctions in the tags attached to such content.

## Targeted Generation

Expressionist is a tool for text generation that associates units of generable content with their meanings, by a scheme in which tags may be attached to the nonterminal symbols in a context-free grammar (Ryan et al. 2016b). When content is generated, it comes bundled with all the tags attached to all the nonterminals that were expanded to produce it. By 'meanings', we mean any semantic, pragmatic, or other information (e.g., a character personality trait) that content may express in the application context. This scheme critically enables two things:

- **Content understanding.** Because generated content comes bundled with its meanings as structured metadata, the larger system may understand it and execute any associated effects. For example, a generated insult might carry tags specifying that the recipient of the line should lower their affinity for the speaker by some amount.

- **Targeted generation.** A system requests content by requesting the meanings that it should express. For instance, the system might request that an insult, or something more specific, as in the example given below.

Targeted generation can naturally be thought of as a search task: find in the space of generable outputs one that has the requested meanings (i.e., requested tags). Due to the combinatorial explosion of generative grammars, where a few hours of authoring may produce trillions or more possible outputs (Ryan et al. 2016b), this search task becomes a major challenge. In response to this, previous work introduced a search method called *middle-out expansion*, where a nonterminal with a desired tag is targeted and the system traverses the grammar in both directions (Ryan, Mateas, and Wardrip-Fruin 2016)—avoiding undesirable tags—until a completed path has been formed. This method only works for requests to target a *single* desired tag, however: unless there is an individual nonterminal with all the desired tags, targeting multiple tags requires finding a path that connects multiple nonterminals that together have all the desired tags, which could equate to search across a space of trillions or more paths. To approximate solutions to cases where multiple tags are desired, a follow-up method, *heuristic expansion*, carries out *greedy* middle-out expansion by preferring production rules whose expansions have nonterminals with desired tags (Ryan et al. 2016a). This method has its own drawbacks, though: it does not guarantee that all desired tags will be collected, and it is susceptible to local maxima.

The overarching goal of this line of work, which the above methods have not satisfied, is to support content requests that contain the following: any number of tags the content must have, any number of tags that it must *not* have, and optionally a scoring metric specifying the desirability of all other tags. Moreover, these kinds of content requests should be fulfillable at both runtime and authoring time, as a form of design support taking the form of queries about the range of content requests that *can* be satisfied at runtime. While these two kinds of support have different practical considerations (relating to the purposes they serve, what is then done with the generated content, etc.), they have the same technical requirement: a mechanism for satisfying content requests of the structure we have just given. We note that “content request” (runtime) and “query” (authoring time) are essentially interchangeable terms: each is a solicitation of content satisfying a provided set of criteria.

## Example

Here, we will illustrate this idea using actual examples from the system that generates text in *Juke Joint* (Ryan et al. 2016a). In this game, procedurally generated characters have visited a small-town bar to mull over personal dilemmas, and the player is a ghost who haunts a jukebox in the bar, selecting which of its songs will play next. As the lyrics of a song emanate across the bar, each stanza elicits *thoughts* in the minds of the characters—expressed in generated natural language—that work to gently guide their streams of consciousness toward prospective resolutions to their dilemmas. When a lyric plays, its *themes* become activated in the minds of the characters, and this may cause other concepts to also become activated. For instance, if the current stanza has the theme *commitment*, the concepts *my job* and *my family* may also become activated for a character who is committed to their job and their family. Meanwhile, our authored Expressionist grammar includes tags that correspond to all of the concepts that can become activated. After a stanza plays, all of the concepts that have become activated in a character's mind are collected to form a content request for an elicited thought to be generated for the character. In the above example, the content request would look something like this:

```
present: ‘commitment’
absent: N/A
metric: (‘my job’, 3), (‘my family’, 1)
```

This request specifies that the generated content must express *commitment*, the lyrical theme, which the generator knows can be expressed by producing text that is associated with a tag called ‘commitment’ (which we as authors have included in our Expressionist grammar). While the request does not include any tags that must *not* be collected, it specifies the relative desirability of ‘my job’ and ‘my family’, with the weights in the tuples defining the former to be three times as desirable. To satisfy this request, the generator collects all the generable thoughts that have the ‘commitment’ tag, and then uses the scoring metric to determine which ones best express ‘my job’ and/or ‘my family’. While this is an example of targeted generation at runtime, we might also want to carry this out at authoring time. For example, we may want to check that a thought *can* be generated from our grammar that expresses *commitment*, *my job*, and *my family*. Or we may want to quickly generate dozens of these to check for content quality.

## Symbolic Visibly-Pushdown Automata

Finite automata are a fundamental tool of thought in computer science; here we briefly explore a variety of increasingly sophisticated automata to obtain important support tools for expressive text generation, which we discuss below. The role of automata in grammar-based text synthesis is undisputed, given the importance of the Chomsky hierarchy and its equivalences between (some) important classes of grammars and increasingly more powerful automata: regular grammars and the finite automata, context-free grammars and pushdown automata, and context-sensitive grammars and linearly-bounded Turing machines. Because Expressionist deploys a context-free grammar, we limit our attention here to the first two classes.

Finite automata can be seen as simple machines (defined as labeled directed graphs) that recognize whether a given finite input sequence belongs or does not belong to a language. Such an automaton has a set of *states*, of which we denote (without loss of generality) an *initial* state and some *terminal* states, along with a set of *edges* between pairs of states (self-edges are allowed) where each edge has an associated *symbol*. We say that an automaton *accepts* a string if and only if it induces a sequence of *moves* along the edges starting from the initial state and ending exactly on a terminal state, without skipping any characters in the string or making any additional moves. The string is processed one character at a time, and a move is only allowed if the current character is exactly the same as the symbol of the prospective edge. While this is phrased as a decision problem, it is clear that we could also use a finite automaton to construct strings by traversing it as a directed graph, accumulating symbols as we go, and terminating when we reach a terminal state; any complete search algorithm would suffice.

Finite automata have some wonderfully useful properties: they admit efficient emptiness checking (it is easy to see if an automaton accepts no strings), and they are *closed* under intersection, union, concatenation, and complementation. Closure here means, for example, that the intersection of two finite automata (an automaton which only accepts the strings which *both* of the component automata would accept) is also a finite automaton. This means we can easily ask for all strings which are valid phone numbers but also have no repeating digits (intersection), or we can ask for words that do not end in "-ing" (complementation) with the exception of "fling" (union). In effect, this allows us to manipulate the potentially infinite-magnitude *languages*—the sets of recognized strings—in terms of operations on the finite automata. Unfortunately, the performance of many of these algorithms scales poorly with increasing numbers of symbols; ASCII is a challenge and full Unicode is unusably large.

The issue here is that an automaton needs one unique edge between states per distinct symbol. But all edges that go from one state to the same target state are in some sense equivalent—the predicate is not really a single character equality, but a set membership problem. *Symbolic* finite automata (SFAs) address and generalize this concern (Bès 2008). In a symbolic automaton, we are still processing a finite string of inputs, but each input comes from a potentially infinite set; to account for this, edges have associated *guards* which are predicates from a given Boolean algebra (a structure admitting disjunction, conjunction, and negation with designated true and false elements). The classic algorithms on finite automata generalize and maintain their closure properties, and because the number of edges remain small they can be extremely efficient. Symbolic finite automata have seen great success in the safety analysis of regular expressions, string sanitizers, and string-manipulating programs (Yu, Alkhalaf, and Bultan 2010; D'Antoni and Veanes 2013; Aydin, Bang, and Bultan 2015).

*Pushdown* automata (PDAs) generalize finite automata in a different direction: by adding some memory in the form of a *stack* and allowing edges to *push* or *pop* symbols or *peek* at the stack's topmost symbol. In a pushdown automata, we split the alphabet into input symbols (as for finite automata) and *stack symbols*, and give edges an associated action (which might be the null action); moreover edges can push a stack symbol or pop a specific symbol off the top of the stack. This allows them to recognize context-free languages involving for example matched parentheses or other counted pairs. Sadly, pushdown automata are not closed under intersection or complementation, although they are closed under intersection with regular languages. For the purposes of validating properties about grammars, intersection is extremely important (the above applications of symbolic automata relied on them extensively).

It is important to note that this lack of closure applies to context-free languages *in general*, but *some* pairs of languages can be intersected *without* leaving the realm of PDAs. Regular languages are one such subset; as it turns out, there is another set in between the regular languages and fully context-free languages called *visibly context-free*, recognized by *visibly* pushdown automata (VPAs), which is extremely useful for our purposes (Alur and Madhusudan 2004). *Visibly* here means that all stack operations (pushes and pops) happen because of specific, recognizable symbols in the input string (we call these respectively *call* and *return* symbols by analogy to function calls). To be available for a transition, a return symbol must match the stack symbol pushed by the corresponding call. Accordingly, these are also called *input-driven* PDAs. Formally, a VPA has a set of states, initial states, terminal states, edges, input symbols, and stack symbols (like a PDA), but the input symbols are further partitioned into calls, returns, and *internal* symbols.

Not every context-free grammar designates its nonterminals with such opening and closing "parentheses," but any context-free grammar (CFG) can be *augmented* to output these as well (becoming a *visibly CFG*); it is straightforward to transform a visibly CFG into a VPA. We can imagine that every nonterminal begins with a call and ends with a return, while every generated terminal is an internal symbol. VPAs are closed under intersection and complementation. We end our tour of automata with *symbolic* VPAs, which, like SFAs, generalize edge conditions to arbitrary Boolean algebras (D'Antoni and Alur 2014). These enjoy all the closure properties of VPAs but remain compact after intersections and other operations, and have recently been used to determine what control flows through programs might have elicited a particular error log (Ohmann et al. 2017).

## Reducing Expressionist Grammars to Automata

Because of the expressive power of symbolic automata, and their successful application in many domains that seem similar to what we need for Expressionist, we wanted to apply an automata-based approach to Expressionist grammars as a solution to the problem we have formulated above. We use the off-the-shelf library SVPAlib (D'Antoni 2017) because it comes with an implementation of SVPAs and several useful Boolean algebras. In this section, we explore how we reduce Expressionist grammars to SVPAs and how queries about generable content with targeted meanings, *i.e.*, desired tags, can be answered through an automata-theoretic lens. We aim for a *reduction* in the computer-science sense: properties of interest should hold on the SVPA if and only if they also hold on the original grammar. As an aside, most of the algorithms we describe work for grammars and SVPAs which can contain loops, which are not allowed in Expressionist grammars.

Our goal is to create, for a given Expressionist grammar, a *deterministic* automaton. Many algorithms of interest require deterministic automata, and while every nondetermin-

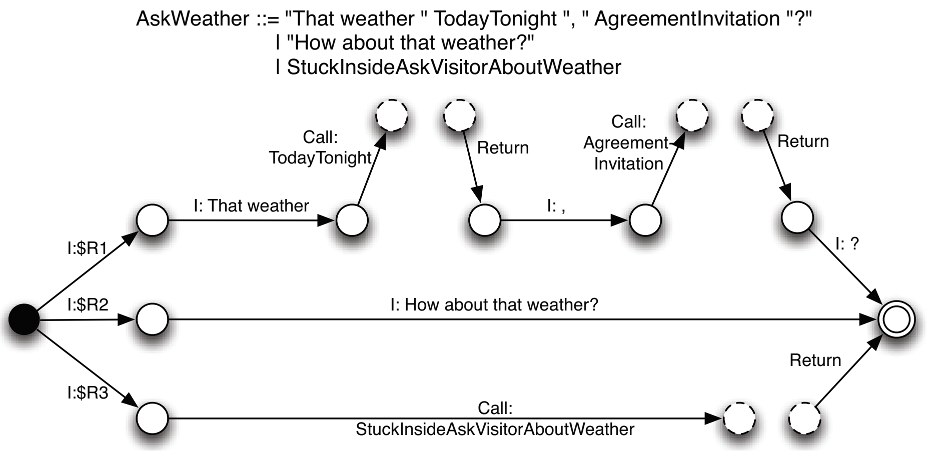
AskWeather ::= "That weather " TodayTonight ", " AgreementInvitation "?"
| "How about that weather?"
| StuckInsideAskVisitorAboutWeather
Call: TodayTonight
Return
Call: Agreement Invitation
Return
I:$R1
I: That weather
I: ,
I: ?
I:$R2
I: How about that weather?
Return
I:$R3
Call: StuckInsideAskVisitorAboutWeather

Figure 2: An Expressionist rule and corresponding SVPA fragment. Edges are labeled (I)nternal, Call, or Return; dotted circles show placeholder start and end nodes in the SVPA fragments built from other rules.

istic SVPA is determinizable, this operation can make an automaton much larger, slowing down future processing. The Boolean algebra we are using is essentially a finite set algebra with a universe comprising every terminal, an identifier for every nonterminal, and an identifier for every Expressionist tag—in other words our edges might have atomic predicates regarding set membership, set union, set complement, set intersection, and so on, with the overall edge predicates being Boolean combinations of those basic queries. A transition which generates a specific terminal, for example, will have as its predicate the singleton set containing that terminal. A transition which denotes a call into a nonterminal will have as its predicate a set containing the union of the nonterminal's identifier and the set of tags attached to that nonterminal. To create a fully deterministic automaton and to simplify the construction, we further augment the set of terminals by adding as many *rule selector symbols* as there are rule choices in the "widest" nonterminal (e.g., if a grammar has a nonterminal with five production rules and no nonterminal with six, then there will be five rule selector symbols). We also add special *root selector symbols* for the possible choices among initial nonterminals (those designated as *top-level* in the grammar). We can imagine that the set of internal symbols is the set of terminals, while the set of opening and closing symbols are derived from the nonterminal identifiers.

For each nonterminal, we build an automaton *fragment* in the following way (illustrated in Fig. 2). First, we allocate an initial node and a final node (every node has a unique integer index; our stack symbols are just integers). Starting from the initial node, every *production rule* induces an internal transition to a fresh state with the corresponding rule selector symbol. Each production step of each rule is either a terminal or a nonterminal. If it is a terminal, we extend a new internal edge with that terminal from the current state to a placeholder state. If it is a nonterminal, we create temporary call and return edges with that nonterminal's identifier and add them to a list of references to be resolved later (the current state's index is used as the stack symbol). The call edge is extended from the current state and the return edge is pointed to a placeholder state. If there are more steps in the rule, the placeholder state is replaced with a freshly allocated state and we continue with the next step; otherwise, the placeholder state is replaced with the nonterminal's end state. We also make a pseudo-nonterminal for the root of the grammar with one production rule for each top-level nonterminal and construct its fragment in the same way.

To construct the full automaton, we start by connecting all the nonterminal fragments together by resolving the targets and sources of the temporary edges. The call edges are attached to the target nonterminal's initial node and the return edges are attached from the nonterminal's final node. The automaton's initial and final nodes are the initial and final nodes of the root fragment.

## Answering Queries

*Enumerating* all outputs of an SVPA is similar to enumerating outputs from a finite automaton. It must be extended in two ways: first, to allow for stack operations by including the stack in the search state and updating it during calls and returns; and second, to go from an abstract path where each element is a set predicate to potentially many concrete paths where each element is a terminal or nonterminal production. Some grammars are essentially not enumerable because they have so many possible outputs, but even these can be addressed well using something like iterative deepening search. Finding a single output is a special case; often, we want to find the shortest output so we can apply a dynamic programming approach to find the fixpoint of the state reachability relation—this gives all-pairs shortest paths between states, and we can take the first found path between any start and end state as our output.

We can address the *targeted generation* problem by combining output-sampling with SVPA intersection using the standard product construction (as implemented in SVPAlib). The approach here is to create an SVPA for our *property*, for example an SVPA that recognizes sequences of calls with the desired tags, and then intersect that SVPA with the one derived from our grammar. The resulting automaton will *only* generate those outputs of the original grammar which have the desired property. We can enumerate outputs using the algorithm outlined above; moreover we can efficiently ask whether the automaton's language is *empty*, i.e., whether it is impossible to generate an output with the given property.

To handle cases where queries also contain forbidden tags, we can consider two low-level properties: a tag is present somewhere in the output or it is absent. If we can describe these low-level properties using automata we can proceed to create any combination of present tags and absent tags by intersecting these together. A tag-present automaton comprises one initial state and one final state; there is an edge from the initial state to the final state when a call matching the desired tag is made. All other possible transitions are embedded as self-loops from initial to initial and self-loops from final to final. In other words, the automaton advances to its terminal state when the tag is found. The tags-absent automaton is identical, still built of two states, but initial state and final state are the same state, while the other state which is the target of the tag-matching call is not final and has no path back to a final state.

We can also build more sophisticated property automata describing any visibly pushdown property: some tags in order, no repetition of a particular production rule or terminal, some set of tags if and only if another tag is found, matched pairs of tags (e.g., every problem is eventually followed by a matching resolution), and so on. Testing any property is roughly as easy as testing any other property. This is a key benefit of the automata-based approach: one algorithm solves arbitrary expressible property queries.

*Output counting* has a more efficient algorithm than just exhaustive enumeration. We implemented the standard dynamic programming formulation of finite automaton path counting, extending it for the symbolic setting by letting a single edge induce multiple paths (as described above for output enumeration) and for the pushdown setting by tracking stack state as well as automaton state and matching calls and returns. Briefly, we want to know how many strings are in the automaton's language; in other words, how many paths there are from initial to final states. Because in general automata may have loops, we can rephrase our question to ask how many strings of length exactly $k$ are in the language, and sum from $k = 0$ to $k = k_{\max}$. Expressionist grammars may not have loops, so we take the longest path through the induced SVPA as a bound on $k$.

We begin by allocating a matrix $M$ of dimensions $k \times |S|$, where $S$ is the set of states in the SVPA. Each entry in the matrix is a set of $(Q, N)$ pairs with $Q$ a stack state (i.e., a list of integers) and $N$ an integer. $M_{0,s}$ is initialized to $\{((), 1)\}$ if state $s$ is an initial state and $\{\}$ otherwise. For each value of $k$ up to the limit, we fill in the next layer of the matrix based on the previous layer, with the values of the states whose edges enter each state $s$ at $k - 1$ contributing to the values of $M_{k,s}$. Each state's $(Q, N)$ values for a given $k$ represent how many ways a state can be reached from the initial configuration within $k$ steps. If at any time we add new ways to reach a final state, we accumulate those as part of our final return value. We also accumulate how many strings there are of each intermediate length. In this way, we can determine the size of grammars with over $10^{64}$ possible outputs in seconds, which Table 1 shows (as explained below). We refer the interested reader to our source code for more detail (Osborn 2017).

| Grammar | Creation | Counting |
|---|---|---|
| Small | 1.9 | 3.4 |
| Medium | 6.2 | 5.1 |
| Large | 2.3 | 2.9 |
| Extra-Large | 24 | 17 |

Table 1: Times (in seconds) to create an automaton from representative grammars and to count its possible outputs.

| Difficulty | Intersection | Check |
|---|---|---|
| Easy | 0.2 | 0.4 |
| Moderate | 0.01 | 0.02 |
| Hard | 11.3 | 19.0 |

Table 2: For each test case, times (in seconds) to intersect each constructed automaton with properties of interest (corresponding to realistic content requests) and to check for (and produce) a satisfying output.

## Evaluation and Discussion

To evaluate our approach as a practical solution to the problem formulated above—satisfying content requests specifying the set of tags that generated content should have (required tags) and the set of tags it should not have (forbidden tags)—we applied it to four actual Expressionist grammars that have been authored for videogame applications (and range in size). The first grammar was authored for character dialogue generation in the ongoing project *Talk of the Town* (Ryan, Mateas, and Wardrip-Fruin 2016); it has 2.8M generable outputs, which in the context of Expressionist makes it our *small* grammar, and its expressive range spans 333 *expressible meanings*, i.e., unique sets of tags that may be attached to generated outputs. Our *medium* grammar was authored for the hacker level of the released game *Project Perfect Citizen*, honorable mention for the Independent Games Festival's Nuovo Award; it has 65 quadrillion generable outputs and 37 expressible meanings. The *large* grammar was authored for character thoughts in the in-development game *Juke Joint* (Ryan et al. 2016a) and features 6.3 quintillion generable outputs and 1,918 expressible meanings. Finally, our *extra-large* grammar was authored for an experimental natural language generation system for *Talk of the Town* (Summerville et al. 2016)—it has a staggering $10^{64}$ outputs and an unknown number of expressible meanings (the algorithm previously used by Expressionist to count these exhausts available RAM and fails).

To carry out our evaluation procedure, we first reduced each grammar to a SVPA using the method described above and then counted the number of generable outputs for each (which could be a useful authoring metric). Table 1 shows how long this procedure took for each grammar on a workstation laptop from late 2013 (2.6 GHz Intel Core i7 CPU, 16 GB RAM). While the durations may appear large, we note that this procedure only has to be carried out once prior to any session of querying the resulting SVPA. For considerations of targeted generation at runtime, this could still happen offline, e.g., during compilation of the game code. When it comes to authoring support, however, waits of several seconds could be too long if each edit is followed by a fresh reduction of the grammar. Interestingly, the medium grammar took longer to reduce than the large grammar—this must be due to structural characteristics of each, but further investigation remains for future work.

Next, for each grammar, we formulated an example content request that could realistically be encountered at runtime and demanded content that satisfied this request (these examples are available alongside our source code as part of its unit tests). These content requests represented three increasing levels of technical difficulty: as an *easy* test case, a content request for the small grammar with two required tags and two forbidden tags; as a *moderate* test case, a content request for the medium grammar with two required tags and two forbidden tags; and as a *hard* test case, a content request for the large grammar with three required tags and three forbidden tags. To execute a test case (and thereby satisfy the example content request), we must first intersect the reduced grammar SVPA with a property SVPA (capturing the required and forbidden tags at hand).

Table 2 shows how long our method took to perform the intersection and checking steps for satisficing outputs for each test case (note that checking also yields a witness output satisfying the property). Again, we find unintuitive nonlinear behavior, with the easy test case taking an order of magnitude longer than the moderate case; this will also require investigation beyond the scope of this paper to interpret. More troublingly, the results demonstrate a threshold between content requests that can be satisfied extremely quickly and ones that take far too long. We note that intersection of more than one property on the extra-large grammar was not possible for us to carry out due to extreme memory usage. Finally, we attempted to check harder test cases on the smaller grammars (with content requests containing 15-30 total stipulated tags), but these also timed out. All of these results suggest that the structure of the grammar is very important to the success of our method, and in particular the performance of automata intersection algorithms is key. Initial profiling suggests that the timeouts and memory exhaustion occur *after* the intersection algorithm is complete, when checking the intersected automata for unreachable states; perhaps we can find some shortcuts here to improve our performance.

## Conclusion and Future Work

The method that we have introduced here, which works by reducing Expressionist grammars to symbolic visibly pushdown automata, represents a first demonstration of targeted generation in pursuit of multiple requested meanings. As mentioned above, earlier work could only target a single required tag, and for additional desired tags the heretofore leading approach resorted to a greedy search that could not guarantee that generated content would have all the requested meanings. While the results here represent a clear leap forward in grammar-based text authoring, our evaluation revealed some troubling nonlinear qualities in the computational task, which manifest in a mysterious threshold between trivial and intractable test cases. This may be due to the specific library we are using, or there may be inherent considerations of computational complexity that we have not anticipated; in future work, we will deeply investigate these matters, considering alternative related approaches. Keep in mind, however, the magnitude of technical challenge that this task represents. Our moderate test case, for example, requires implicitly searching through a space of 65 quadrillion possible outputs to find one that has two specific properties and does not have two others. This is a monumental search task, and our method carries it to completion in 1/50th of a second. Still, we can check moderately complex properties on complex grammars, but we cannot yet check complex properties even on relatively simple (order $10^7$) grammars.

Another avenue for future work pertains to a feature of Expressionist that we have not yet mentioned: authors can attach to production rules *probabilities of application*, which makes the underlying grammar formalism a *probabilistic* CFG. As such, we would like to extend our SVPA approach to handle such probabilism, both for design support (concerning the likelihood of generable content with properties of interest) and runtime support (probabilistic generation). Moreover, some Expressionist authors have exploited the free-text nature of tags in the tool to make use of special *condition logic* tags, which gate the expansion of nonterminals according to the system state (Ryan et al. 2016b). Handling the semantics of preconditions is another possible extension, especially since SVPAs offer symbolic guards which can be from any Boolean algebra.

Text generation in games and interactive storytelling is typically carried out *iteratively*: outputs may occur in series, or may be constrained according to the context or earlier outputs, or may affect the generation contexts of future outputs. As such, we are beginning to think about how an SVPA could capture these dynamical aspects of text generation to provide even more advanced design support at authoring time—for instance, about the space of possible *sequences* of generated outputs.

While we have focused on Expressionist in this paper, due to its reification of properties of interest as tags, we anticipate this approach being applicable to other tools and methods, too, such as Tracery and templated dialogue. Here, the work will be in determining a set of properties of interest and a method for discerning them. For example, in lieu of Expressionist-like tags, we could recognize surface characteristics of terminal symbols in Tracery (such as their length) and then build property automata whose edges are associated with these properties. This would make it possible to generate outputs with desired surface characteristics.

We are particularly interested in the artistic affordances of *grammar sculpting*. By this, we mean the direct construction of new grammars using the operations of intersection, union, concatenation, and subtraction, since these can be applied between pairs of grammars just as we have done with property-grammar pairs. An author could form a permissive grammar and subtract out a smaller grammar of repetitive phrases or unpleasant juxtapositions; or combine two grammars on related themes to obtain a new grammar synthesizing those themes, perhaps along their shared tags or nonterminal IDs; or instantiate the same base grammar differently for different regional dialects in a fictional world (or indeed, for different characters in the world). While this would be very experimental, especially since the resulting grammars may only be machine understandable, we can already imagine some interesting use cases—for example, authoring a permissive grammar and then automatically subtracting out classes of undesirable outputs.

## References

Alur, R., and Madhusudan, P. 2004. Visibly pushdown languages. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, 202–211. ACM.

Aydin, A.; Bang, L.; and Bultan, T. 2015. Automata-based model counting for string constraints. In *International Conference on Computer Aided Verification*, 255–272. Springer.

Bès, A. 2008. An application of the feferman-vaught theorem to automata and logics for words over an infinite alphabet. *Logical Methods in Computer Science* 4.

Cardona-Rivera, R. E. 2017. Cognitively-grounded procedural content generation. In *Proc. What's Next for AI in Games*.

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

Compton, K. 2016. So you want to build a generator... http://galaxykate0.tumblr.com/post/139774965871/so-you-want-to-build-a-generator.

Cook, M.; Gow, J.; and Colton, S. 2016. Danesh: Helping bridge the gap between procedural generators and their output. In *Proc. Procedural Content Generation*.

Cook, M. 2016. Alien languages: How we talk about procedural generation. *Gamasutra*.

D'Antoni, L., and Alur, R. 2014. Symbolic visibly pushdown automata. In Biere, A., and Bloem, R., eds., *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, 209–225. Springer.

D'Antoni, L., and Veanes, M. 2013. Static analysis of string encoders and decoders. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, 209–228. Springer.

D'Antoni, L. 2017. A symbolic automata library. https://github.com/lorisdanto/symbolicautomata.

Dias, B. 2016a. Procedural generation in voyageur. https://vimeo.com/182465861.

Dias, B. 2016b. Procedural meaning: Pragmatic procgen in Voyageur. *Gamasutra*.

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

Garbe, J.; Reed, A. A.; Dickinson, M.; Wardrip-Fruin, N.; and Mateas, M. 2014. Author assistance visualizations for ice-bound, a combinatorial narrative. In *Proceedings of the Ninth International Conference on the Foundations of Digital Games*.

Horswill, I. D. 2014. Architectural issues for compositional dialog in games. In *Proc. GAMNLP*.

Lessard, J.; Brunelle-Leclerc, E.; Gottschalk, T.; Jetté-Léger, M.-A.; Prouveur, O.; and Tan, C. 2017. Striving for author-friendly procedural dialogue generation. In *Proc. Non-Player Characters and Social Believability in Games*.

Lessard, J. 2016. Designing natural-language game conversations. In *Proceedings of the First International Joint Conference of DiGRA and FDG*. Digital Games Research Association and Society for the Advancement of the Science of Digital Games.

Lewis, M. 2017. Stochastic grammars: Not just for words! In Rabin, S., ed., *Game AI Pro 3*. CRC Press.

Mawhorter, P. A. 2016. *Artificial Intelligence as a Tool for Understanding Narrative Choices*. Ph.D. Dissertation, University of California, Santa Cruz.

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

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

Ohmann, P.; Brooks, A.; D'Antoni, L.; and Liblit, B. 2017. Control-flow recovery from partial failure reports. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, Barcelona, Spain*.

Osborn, J. C. 2017. Reductionist. https://github.com/joeosborn/reductionist.

Ryan, J.; Brothers, T.; Mateas, M.; and Wardrip-Fruin, N. 2016a. Juke joint: characters who are moved by music. *Proc. Experimental AI in Games*.

Ryan, J.; Seither, E.; Mateas, M.; and Wardrip-Fruin, N. 2016b. Expressionist: An authoring tool for in-game text generation. In *Proc. ICIDS*.

Ryan, J.; Mateas, M.; and Wardrip-Fruin, N. 2016. Characters who speak their minds: Dialogue generation in talk of the town. *Proc. AIIDE*.

Short, E. 2014. Procedural text generation in IF. https://emshort.wordpress.com/2014/11/18/procedural-text-generation-in-if/.

Short, E. 2016a. Bowls of oatmeal and text generation. https://emshort.blog/2016/09/21/bowls-of-oatmeal-and-text-generation/.

Short, E. 2016b. Visualizing procgen text. https://emshort.blog/2016/06/13/visualizing-procgen-text/.

Short, E. 2016c. Visualizing procgen text, part two. https://emshort.blog/2016/09/12/visualizing-procgen-text-part-two/.

Smith, G., and Whitehead, J. 2010. Analyzing the expressive range of a level generator. In *Proc. Procedural Content Generation in Games*, 4.

Spirit AI. 2017. http://spiritai.com/.

Summerville, A. J.; Ryan, J.; Mateas, M.; and Wardrip-Fruin, N. 2016. CFGs-2-NLU: Sequence-to-sequence learning for mapping utterances to semantics and pragmatics. Technical Report UCSC-SOE-16-11, UC Santa Cruz.

Togelius, J.; Shaker, N.; and Dormans, J. 2016. Grammars and l-systems with applications to vegetation and levels. In *Procedural Content Generation in Games*. Springer. 73–98.

Yu, F.; Alkhalaf, M.; and Bultan, T. 2010. Stranger: An automata-based string analysis tool for php. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 154–157. Springer.