Proceedings of the Sixteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-20)

A Declarative PCG Tool for Casual Users

Ian Horswill
Department of Computer Science, Northwestern University, Evanston IL
ian@northwestern.edu

Abstract

Declarative programming allows designers to create procedural content generators by providing descriptions of desired artifacts rather than bespoke algorithms to generate them. Unfortunately, these systems are notoriously inaccessible, requiring considerable sophistication with formal systems, and detailed understanding of the impact of equivalent formalizations on the system performance.

Imaginarium is a declarative PCG system for tabletop role playing games (TTRPGs). Following Compton's work on *casual creators*, it trades expressiveness for accessibility. As with Nelson's *Inform 7*, its source language is a highly structured subset of English. A single sentence, such as `children are parented by at least one adult` can be used to simultaneously introduce the predicates child, adult, and parent, along with a cardinality constraint over the parent relation. We describe the system, its knowledge representation language, and the issues in their design. Together, they allow users with minimal STEM background to engage in playful experimentation.

Introduction

Procedural content generation (PCG) systems create randomized objects for use in games or other media. Systems have been developed to create 3D models (IDV 2009), dungeon levels (Toy *et al*., 1980) and even galaxies (Wright *et al*., 2008). They can be built using bespoke algorithms (Adams and Adams 2006), machine learning (Summerville, *et al*., 2018), or declarative methods (G. Smith *et al*., 2011), among others. These methods require considerable expertise; while a number of systems have been developed targeting end-users for specific problems, such as Mario level generation (G. Smith *et al*., 2011), broad coverage, end-user systems are difficult to develop.

Imaginarium is a declarative PCG tool for tabletop role-playing. It allows players to build constraint-based procedural generators for in-game entities such as characters and items, or networks of entities connected by relationships.

In *Imaginarium*, a *generator* is a declarative program that specifies an ontology for some desired set of entities, written in a highly restricted subset of English. It generates *inventions*, each containing one or more logical *individuals* (entities) with specified attributes and/or relationships.

The command, `imagine a character`, directs the system to create an invention containing one individual of the kind `character`. The command, `imagine 10 penguin clerics`, directs the system to make an invention containing 10 individuals that are both `penguins` and `clerics`.

The system compiles the subset of the ontology required for the requested invention into a Satisfaction Modulo Theories (SMT) problem,[1] uses an off-the-shelf solver (Horswill 2018a) to generate a random model, then converts it back into natural language.

Imaginarium is heavily influenced by Compton's work on casual creators (Compton and Mateas 2015) and Nelson's work on *Inform 7* (Nelson 2006b, 2006a). While ultimately a declarative programming language, it seeks to leverage the affordances of natural language to allow non-programmers to use it without first learning the subtleties of first-order logic formalization or the deeper subtleties of answer-set programming (A. M. Smith *et al*., 2012).

In this paper, we describe *Imaginarium*'s design commitments, knowledge representation and implementation. A less technical description of an early version may be found in (Horswill 2018b).

Example

Suppose we want to generate random cats.[2] The command `imagine a cat` tells the system to generate a random object of the kind `cat`. It implicitly tells the system that

[1] SMT is a mechanism for embedding non-Boolean variables into SAT problems. For this paper, SMT can be safely treated as a synonym for SAT.

[2] There are in fact many cat-based TTRPGs, including Hanson's recent second edition of *Magical Kitties Save the Day!*

`cat` is a kind, membership in which is indicated in English using the noun *cat*. Since, this is all the system knows about cats, it can respond only with: `the cat is a cat`.

If we now tell it: `a cat is large or small`, it now knows cats come in two flavors, `large` and `small`, the distinction being indicated using the adjectives *large* and *small*. The system now respond either with `the cat is a large cat` or `the cat is a small cat`, those being the only two possible cats. If we want to allow the cats to be neither large nor small, we can change *is* to *can be*: `Cats can be large or small.` This allows a cat with unmarked size; however, there are still only three possible cats.

If we add: `Persian, tabby, and Siamese are kinds of cat`, the system will know *Persian*, *tabby*, and *Siamese* are all nouns denoting subkinds of the kind cat. We can type more statements to tell it about more kinds of cats. When generating a cat, it will always be of one of the specified subkinds. We might also tell the system:

```
Cats are longhaired or shorthaired.
Cats are grey, white, black, or ginger.
A cat can be haughty, cuddly, crazy, or
Nietzschean.
```

Which define two mandatory properties, color and coat length, and one optional personality property. We can now generate cats such as:

```
The cat is a large, shorthaired, white
Persian.
The cat is a longhaired, ginger, cuddly
tabby.
```

However, the first of these is problematic, since Persians are longhaired by definition. We can prevent this by adding constraints such as:

```
Persians are longhaired.
Siamese are shorthaired.
Siamese are grey.
```

Finally, we name our cats and let them have ages:

```
Cats have a name from cat names.
Cats have an age between 1 and 20.
```

The first of these tells the system that all cats have a string-valued property, `name`, drawn randomly from the list in the file: `cat names.txt`. It also tells the system to describe the cat using her name rather than the generic identifier `the cat`. We now get outputs like:

```
Puck is a Nietzschean, ginger tabby, age 12
Mr. Muffins is a large, grey, Siamese, age
2
Rover is a small, white, crazy manx, age 9
```

Finally, we say: `cats can love other cats`, which introduces a verb `love` that represents an anti-reflexive, binary relation. If we now say: `imagine five cats`, the system will display a set of five random cats, together with an interactive visualization of the *loves* relation as a directed graph.

Comparison with Traditional Logic Programming

The foregoing 10 commands are roughly equivalent to the 16-line AnsProlog (Baral and Baral 2009) program:

```
entity(1..5).
cat(X) :- entity(X).
cat(X) :- persian(X).
cat(X) :- tabby(X).
cat(X) :- siamese(X).
1 { persian(X) ; tabby(X); siamese(X) } 1
  :- cat(X).
1 { age(X, 1..20) } 1 :- cat(X).
1 { name(Cat, Name) : cat_name(Name) } 1
  :- cat(Cat).
0 { large(X) ; small(X) } :- cat(X).
1 { long_haired(X) ; short_haired(X) } 1
  :- cat(X).
1 { black(X) ; white(X) ; grey(X) ;
   ginger(X) } 1 :- cat(X).
0 { cuddly(X) ; haughty(X) ; crazy(X) ;
  nietzschen(X) } 1 :- cat(X).
long_haired(X) :- persian(X).
short_haired(X) :- siamese(X).
grey(X) :- siamese(X).
{ loves(X, Y) } :- cat(X), cat(Y).
```

Using Clingo (Gebser et al. 2010) produces the output:

```
entity(1) entity(2) entity(3) entity(4)
entity(5) cat(1) cat(2) cat(3) cat(4)
cat(5) long_haired(2) persian(2)
long_haired(3) long_haired(5) persian(5)
short_haired(1) siamese(1) short_haired(4)
grey(1) grey(4) tabby(3) tabby(4) large(1)
small(1) small(2) large(3) large(4)
large(5) small(5) ginger(2) black(3)
ginger(5) cuddly(1) crazy(2) cuddly(3)
crazy(5) loves(1,1) loves(3,1) loves(4,1)
loves(5,1) loves(1,2) loves(3,2) loves(5,2)
loves(3,3) loves(4,3) loves(1,4) loves(3,4)
loves(5,4) loves(1,5) loves(3,5) loves(4,5)
loves(5,5)
```

While far less expressive than AnsProlog, *Imaginarium* compares favorably in both conciseness and readability within its domain of competence.

Moreover, *Imaginarium* can infer most of the rules for translating a model into natural language directly from the syntactic structure of the program text. Other languages would require additional declarations. These properties,

together with the system's interactive design, make it much more novice-friendly.

Design Commitments

Imaginarium commits to a number of design decisions that have knock-on effects for other parts of the design.

Implicit Declaration. Users can introduce new terms without a separate "this is a noun" declaration. The system detects new terms and infers their syntactic categories from context. Users can write: `people can know each other`, without having to first teach the system that `people` is a noun or that `know` is a verb.

Phrasal Lexicon. Users can designate nearly any sequence of words as a noun, verb, or adjective. `International conspiracy` can be used as if it were an atomic noun without first formalizing the separate concepts `conspiracy` and `international`.

Unmodified First Use. Phrasal items introduce ambiguity: `international conspiracy` could be intended as a single phrasal noun, or as an adjective modifying a noun; this ambiguity conflicts with implicit declaration. Therefore, the first use of a term must be in unmodified form. If `conspiracy` is to be a kind, the source text must use it without adjectives before using it with adjectives.

Regular Grammar with Closed-Class Words as Parsing Anchors. Again, to reduce ambiguity and aid recognition of new terms, user-defined terms are separated in the grammar by fixed tokens to ensure their boundaries are unambiguous. Rather than allowing constructions like `adults parent children`, we use `adults` *`can`* `parent` *`many`* `children`. The fixed tokens *`can`* and *`many`* explicitly delimit the boundaries of `adult`, `child`, and `parent`. The only exception to this rule is that adjective may modify nouns. These are disambiguated by the unmodified first use rule.

Individuals are Known at Compile Time. Compiling SMT problems involves grounding first-order axioms into propositional logic. This requires knowing the potential extensions of predicates (the individuals to which terms like `person` and `parent` could potentially apply). The system must therefore know the set of individuals that exist in the models being made, even though it will not yet know their exact kinds and properties.

Monotone Semantics. Unlike AnsProlog, *Imaginarium* generators obey the laws of monotonic classical logic. The models of a set of sentences are the intersection of the models of the individual sentences. This eases debugging at the cost of reducing expressiveness.

Knowledge Representation Language

Imaginarium's KR language is patterned on English.

Monadic Concepts

Monadic (unary) predicates are surfaced either as **(common-) nouns** or as **adjectives**. Nouns are used to define taxonomies of object kinds (aka classes or types). Every individual is required to have at least one kind. Attributes attach to kinds and are inherited by subkinds.

Adjectives define binary and enumerated attributes of kinds. The sentence, `a person can be melancholic` adds the binary attribute `melancholic` to the kind `person`. The constraint, `poets are melancholic`, requires it to hold of all `poet`s in all models. Enumerated attributes are introduced using an "or" construction. `Cars are two door, four door, or hatchback`, introduces three new adjectives, `two door`, `four door`, and `hatchback` and makes them mutually exclusive.

Relations

Dyadic predicates are surfaced as verbs. Relations can be marked as (anti)symmetric and/or (anti)reflexive. For example, `people can't employ themselves`, marks the verb `employ` as being an anti-reflexive relation between people, while `children must be parented by at least one responsible adult` defines the verb `parent` as a relation between `adult`s who are also `responsible` and `children`. It also adds the axiom: $\forall c.\,\text{Child}(c) \Rightarrow \exists p.\,\text{Adult}(p) \wedge \text{Responsible}(p) \wedge \text{Parent}(p, c)$.

Note that relations cannot be tagged as transitive (in the logical sense) since transitivity is not first-order definable.

Properties

Properties are attributes, also attached to kinds, whose values are not surfaced through adjectives. They are currently restricted to numeric-valued attributes, such as age, and string-valued attributes, such as a character's name. They are implemented using SMT variables. For example, the construction `people have an age between 1 and 70` attaches a numeric-valued property to the `person` kind.

Parts

Parts are attributes whose value is another individual within the underlying logic. The construction, `people have an animal called their pet`, attaches a new attribute, `pet`, to the `person` kind and specifies that its value must be a newly generated individual of the `animal` kind.

Individuals

Most individuals in an invention are defined by the imagine command; `imagine 10 characters` forces the invention to include 10 individuals of the character kind. However, the user can also add specific individuals to the ontology that must exist in all inventions. For example,

`The Big Bad is an evil character`, forces the existence of a `character` individual in all models, called `The Big Bad`.

Constraints

Explicit constraints specified by the user take the form of individual statements that correspond to first-order clauses or their generalization, pseudo-Boolean constraints. The statement, taken from a class assignment, `flying monsters are winged`, maps to the first-order clause:

$$\forall x.\, \text{monster}(x) \wedge \text{flying}(x) \rightarrow \text{winged}(x)$$

or, in disjunctive form:

$$\forall x.\, \neg\text{monster}(x) \vee \neg\text{flying}(x) \vee \text{winged}(x)$$

A cardinality constraint such as the following encoding of personalities from *The Sims 3*:

```
A character is any two of absent-minded,
artistic, avant-garde, bookworm, can't
stand art, computer whiz, eccentric, ex-
citable, gatherer, genius, green thumb,
handy, insane, natural cook, neurotic,
nurturing, perceptive, photographer's
eye, savvy sculptor, or virtuoso.
```

would compile to a single pseudo-Boolean constraint:

$$2\neg\text{character}(c) + \text{absentminded}(c) + \text{artistic}(c) + \text{avantgarde}(c) + \text{bookworm}(c) + \ldots + \text{virtuoso}(c) = 2$$

This states that for all individuals c, the number of personality properties holding of c, plus 2 when c is not a character, must always be 2. The system does not currently allow an adjective to appear in two different cardinality constraints.

Taxonomic Relations

Finally, nouns and verbs (kinds and binary relations) can be arranged in taxonomies, with children inheriting the attributes of their parents. Any instance of the parent must also be an instance of exactly one of its children. If we say `bird, reptile, and mammal are kinds of land animal`, then the system knows that `bird` implies `land animal` and that all `land animal`s must also be `bird`s, `reptile`s or `mammal`s, but never more than one at once.

Verb taxonomies provide a way of specifying an abstract relation than can be instantiated by multiple concrete relations. For example, in *Fiasco* (Morningstar 2009), every player character must have relationships with two other players, but the specific relationships vary based on dice and player selection. This can be encoded with something like the following:

```
Characters can relate to each other.
Characters relate to 2 other charac-
ters.
Loving is a kind of relating to.
Hating is a kind of relating to.
```

etc.

The first statement marks `relate to` as a symmetric relation on `character`s. The second says that all characters relate to exactly 2 characters and not themselves (the relation is anti-reflexive). And the subsequent statements give possible verbs that constitute relating to.

The kinds and relations each form a lattice, although their top and bottom elements are not explicitly represented.

Development Tools

Imaginarium includes a number of quality of life features to ease development:

- **Interactive graph visualizations** for the ontology and the generated relations.
- A **unit test** rig allowing generators to specify sentences that should/shouldn't be satisfiable.
- **Syntax highlighting** for sublime text.
- **Support for git clone and fetch** under Windows to make sharing dissemination of generators easier.
- The ability to add **buttons** to the UI.
- An **autograder** for running batches of class assignments against benchmark tests.

Implementation

Imaginarium's parser is very simple and is implemented as a set of discrete sentence patterns.

When generating models in response to an `imagine` command, the system first compiles the relevant parts of the ontology into a SMT problem, then uses an off-the-shelf SMT solver to find a model, and finally converts the model back to English.

Compilation

The compilation process works in three phases. First, the system determines the set of logical individuals to appear in the models. Then it generates propositions and clauses for all monadic predicates (nouns and adjectives) by walking the lattice of kinds relevant to a given individual. Finally, it generates propositions and clauses for the verbs.

Definitions. In the following, we will use $<$ to denote the ordering the relation for the lattices of kinds and relations. So $A < B$ means A is a specialization of B and B a generalization of A. We will use $A \lessdot B$ to indicate that A is an immediate descendant of B, i.e. $A < B \wedge \nexists I.\, A < I < B$.

For each individual i, we know at least one kind K_i that it is declared to satisfy, either because it was specified in the `imagine` command, or for individuals that are parts of other individuals, it was specified in the part declaration.

For simplicity, we will assume there is only one such declared kind for each individual; generalization is straightforward.

The *potential kinds* of an individual i, the set of all kinds to which i can possibly belong, is $\mathcal{K}_i = \{K \mid K \leq K_i \vee K_i \leq K\}$. Conversely, the *potential extension* of a kind K, the set of individuals that can potentially be of that kind is:

$$\mathcal{P}_K = \{i \mid K \in \mathcal{K}_i\} = \{i \mid K \leq K_i \vee K_i \leq K\}.$$

Individual Determination. The system determines which individuals (entities) are to exist in the model by starting with the individual(s) requested in the `imagine` command and recursively adding their parts. For example, if user gave the command `imagine a person`, and the ontology lists `person`s as having `pets`, then the system would add an additional individual for the pet and, recursively, any individuals required for the pet's parts. Finally, if there are proper nouns in the ontology, these are also added regardless of the specific `imagine` command.

Clauses for Monadic Predicates (noun and adjectives). We next generate the clauses formalizing the potential membership of each individual i in each kind $K \in \mathcal{K}_i$. To formalize that i may potentially belong to K, we generate the clauses:

$$K(i) \rightarrow S(i), \text{for each S} \succcurlyeq K$$
$$K(i) \rightarrow A(i), \text{for each adjective } A \text{ implied by } K$$
$$\neg K(i) + \sum_{S \prec K} S(i) = 1$$

The latter pseudo-Boolean constraint ensures that $K(i)$ holds iff exactly one $S(i)$ holds, i.e. individuals of kind K are also of one of its immediate subkinds.

Finally, alternative sets attached to the kind K, such as, Ks `are` $A, B,$ or C, compile to pseudo-Boolean constraints:

$$\neg K(i) + A(i) + B(i) + C(i) = 1$$

This process is performed for every individual i and for every $K \in \mathcal{K}_i$. Finally, the assertion $K_i(i)$ is added, asserting that i is always of kind K_i.

Clauses for verbs. Let K_{vs} and K_{vo} be the declared kinds for its subject and object positions of a verb v.

For each v, and individuals $s \in \mathcal{P}(K_{vs}), o \in \mathcal{P}(K_{vo})$, we generate the clauses for the constraints on $v(s,o)$, i.e. v holding of s and o:

$$v(s,o) \rightarrow K_{vs}(s)$$
$$v(s,o) \rightarrow K_{vo}(o)$$

For all v's immediate generalizations $v' \succcurlyeq v$:

$$v(s,o) \rightarrow v'(s,o)$$

If v has specializations, we require that if $v(s,o)$ holds, then one of its immediate specializations $v' \prec v$ must also hold:

$$\neg v(s,o) + \sum_{v' \prec v} v'(s,o) = 1$$

Model Finding

Having generated the propositions and constraints for the SMT problem, the system calls an off-the-shelf randomized solver (Horswill 2018a) to generate a random model M.

Natural Language Generation

Finally, the model is presented to the user. Natural language sentences are generated for each individual, and the relations defined by verbs are presented using a graph visualization.

The NL generator is not particularly sophisticated. Since verb information is displayed in the graph, the NL system is primarily used to generate a noun phrase presenting the monadic predicates (nouns and adjectives) true of the individual.

Let the *description* of i, the monadic predicates that hold of i in M, be $D = \{P \mid M \vDash P(i)\}$. The *filtered description* of i, $F = \{P \in D \mid \nexists P' \in D. \forall x. P'(x) \rightarrow P(x)\}$, is subset of D not already implied by other predicates in D. The head noun H for the NP describing i will be whatever specialization of i's declared kind appears in F. That is, it is the unique $H \in F$ for which $H \leq K_i$. The NP is then simply the nouns and adjectives $F - \{H\}$, in some arbitrary order, followed by the H at the end.

The NP is then embedded in a statement of the form "*Name* is a *NP*", where *Name* is the i's name. If i has properties, their values are added to the description.

By default, i's name is a mechanically generated identifier such as `character7`. However, if it has a defined property called `name`, its value is used instead. Declarations can also be used to change the templates used to generate names and descriptions for specific kinds.

Future Work

Imaginarium's KR language has a number of limitations that present opportunities for further work. Defeasible rules (rules that can be overridden by other rules) would be useful. This would allow statements such as, `poodles are usually large`, to be overridden by: `toy dogs are small`.

Another important capability would be better control over a generator's sampling distribution. While impractical in the most general case – full control would move us from an NP-complete problem to a #P-complete one – there are a number of limited but useful features that could be added.

User Experience

In preparation for a formal user study next year, we used the system in a course in which STEM and non-STEM students built generators and games incorporating generators. While the system was well received, it made clear that additional work is needed to reduce friction for non-STEM students.

One issue is that the system is still, in the end, a programming language. While the program itself requires relatively little knowledge of computing, it still requires students to install a text editor and navigate to the proper folder to find a source file. These are intimidating for some students.

It would also clearly be helpful for the system to do more proactive consistency checking to identify likely mistakes. Trivial paraphrases, such as the difference between "work for" and "be working for", seem equivalent to users but are treated by the system as distinct, unrelated concepts. Users can waste large amounts of time debugging seemingly strange behavior if they don't notice there are two versions of what they consider to be one concept.

Diversity and Inclusion

One important advantage of TTRPGs over digital games is that they allow characters to have social identity elements such as gender, race, and sexual orientation without standardizing rules for them. When playing a game set in the 1920s (or contemporary) US, players can make an affirmative choice whether to acknowledge the historical inequities of that period in the design of their characters, or to make deliberately anachronistic choices that better reflect their own values. Game rules often explicitly encourage players to discuss these issues during setup. By contrast, digital games must fully commit to specific rules for everything they model. As Compton argues (2017), they define not only possibility spaces , but *impossibility* spaces. You can only have dark-skinned elves if the designers explicitly allowed for dark-skinned elves.

The importation of procedural content generation into TTRPGs thus risks the importation of digital games' problems with it. PCG systems inevitably embody the assumptions and values of the humans who make them (Philips *et al.* 2016). While a character generator that presumes heterosexuality or binary gender, or that only generates "white" sounding character names, might match the lived experience of its author, it could be deeply alienating to others.

To combat these problems, more work is needed to allow broader, more diverse populations of players and designers to build their own generators. Equally importantly, we need systems that allow players to easily "mod" generators designed by others, creating their own "house rules" for PCG. Both these problems require considerably more work.

Related Work

A number of designer-facing logic- and rule-based systems have been developed for games. Perhaps the best known such system is Nelson's *Inform 7* interactive fiction system (Nelson 2006a, 2006b), which was a major influence on this work. Nelson argues that programming languages based on natural language, while inappropriate for general-purpose programming, are a good match for tasks like IF authoring, where the domain itself is natural language text.

Compton's work on *casual creators* (Compton and Mateas 2015) is another major influence on this work, both her arguments about trading expressiveness for accessibility, and her arguments about facilitating a user experience of exploration and surprise while minimizing the frequency with which users encounter complete failure. Her system *Tracery* (Compton *et al.* 2014), a tool to allow naïve users to develop text generators based on context-free grammars, has literally thousands of users.

Although uncommon, a number of logic programming and rule-based systems have been used in video games in the past. *The Sims 3* used a simple forward-chaining production system to allow designers to author behavior rules based on character personality (Evans 2009).

Prom Week (McCoy *et al.* 2012), and its underlying engine *Comme Il Faut* (McCoy *et al.* 2011) also used a forward-chaining rule-based system to track character responses to social actions.

The *Versu* interactive fiction system (Evans and Short 2013) used an innovative logic programming system based on exclusion logic. However, it proved extremely difficult for authors to use, and so Nelson developed *Prompter* (Nelson 2013), an *Inform 7*-like natural language front end that proved more accessible.

Most recently, the forthcoming *City of Gangsters* (SomaSim 2021), a tycoon game set in the Chicago prohibition era, uses the SMT solver used here to generate character personalities, as well as a logic programming system to deduce the social effects of character actions.

Finally, there are a few examples of designer-facing constraint-based PCG tools. The first and best known of these is *Tanagra* (Smith et al. 2011), a constraint-based Mario level editor. More recent examples include *Gemini*, a game generator that uses ASP to reason about the aesthetics of the games it generates (Summerville *et al.* 2018), and *AutoDread*, a backstory generator for IF characters (Horswill and Robison 2018).

Conclusion

Imaginarium is a simple declarative PCG system for tabletop role-playing. It allows the playful creation and sharing of generators by non-programmers, without their first having to learn formal logic or knowledge representation. While users must still read a tutorial to use it, it is significantly more accessible than traditional logic programming languages. That said, considerable work can be done to increase both its usability and expressiveness.

Acknowledgements

The author wishes to thank to Ethan Robison, Kate Compton, Adam Summerville, Gillian Smith, Willie Wilson, the students and TAs of CS 295/396, and the AIIDE reviewers for their feedback and patience.

References

Adams, T., and Adams, Z. 2006. *Slaves to Armok: God of Blood Chapter II: Dwarf Fortress*.

Baral, C.. 2009. Declarative Problem Solving and Reasoning in AnsProlog in *Knowledge Representation, Reasoning and Declarative Problem Solving*.

Compton, K. 2017. Little Procedural People in *PCG Workshop, Foundations of Digital Games Conference*. Hyannis, Mass: ACM Digital Library.

Compton, K.; Filstrup, B.; and Mateas, M. 2014. Tracery: Approachable Story Grammar Authoring for Casual Users. In *Papers from the 2014 AIIDE Workshop, Intelligent Narrative Technologies (7th INT, 2014)*, 64–67.

Compton, K., and Mateas, M. 2015. Casual Creators. In *Proceedings of the Sixth International Conference on Computational Creativity June*.

Evans, R. 2009. AI Challenges in Sims 3 in *Artificial Intelligence and Interactive Digital Entertainment*. Stanford, Calif: AAAI Press.

Evans, R., and Short, E. 2013. Versu.

Gebser, M.; Kaminski, R.; Kaufmann, B.; Ostrowski, M.; Schaub, T.; and Thiele, S. 2010. *A User's Guide to Gringo, Clasp, Clingo, and Iclingo∗*. Potsdam.

Guzdial, M.; Liao, N.; and Riedl, M.. 2018. Co-Creative Level Design via Machine Learning in *CEUR Workshop Proceedings*.

Horswill, I. 2018a. CatSAT: A Practical, Embedded, SAT Language for Runtime PCG in *AIIDE-18*. Menlo Park, Calif.: AAAI Press.

Horswill, I. 2018b. Imaginarium: A Tool for Casual Constraint-Based PCG in *Workshop on Experimental AI in Games, AIIDE 2018*. Atlanta, Ga.: AAAI Press.

Horswill, I., and Robison, E. 2018. What's the Worst Thing You've Ever Done at a Conference? Operationalizing Dread's Questionnaire Mechanic in *AIIDE-18 Workshop on Experimental AI in Games (EXAG-18)*. Edmonton, Canada: AAAI Press.

IDV. 2009. SpeedTree.

McCoy, J.; Treanor, M.; Samuel, B.; and Reed, A. A. 2012. Prom Week.

McCoy, J.; Treanor, M.; Samuel, B.; Wardrip-Fruin, N.; and Mateas, M. 2011. Comme Il Faut: A System for Authoring Playable Social Models in *Proceedings of the 7th AI and Interactive Digital Entertainment*, edited by V. Bulitko and M. O. Riedl. Stanford, Calif.: AAAI Press.

Morningstar, J. 2009. *Fiasco*. Durham, NC: Bully Pulpit Games.

Nelson, G. 2006a. Inform 7.

Nelson, G. 2006b. Natural Language, Semantic Analysis, and Interactive Fiction.

Nelson, G. 2013. *Writing for Versu*. San Francisco, Calif.: Linden Lab.

Philips, A,; Smith, G.; Cook, M.; and Short, T. 2016. Feminism and Procedural Content Generation: Toward a Collaborative Politics of Computational Creativity. *Digital Creativity* 27(1): 82–97.

Smith, A. M.; Andersen, E.; and Mateas, M. 2012. A Case Study of Expressively Constrainable Level Design Automation Tools for a Puzzle Game in *International Conference on the Foundations of Digital Games*. Raleigh, NC: ACM Press.

Smith, G.; Whitehead, J.; and Mateas, M. 2011. Tanagra: Reactive Planning and Constraint Solving for Mixed-Initiative Level Design. *IEEE Transactions on Computational Intelligence, AI and Computer Games* 3(3): 201–215.

SomaSim. 2021. City of Gangsters.

Summerville, A.; Martens, C.; Samuel, B.; Osborn, J.; Wardrip-fruin, J.; and Mateas, M.. 2018. Gemini: Bidirectional Generation and Analysis of Games via ASP in *Proceedings of the Fourteenth Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2018)*, 123–229. Edmonton, Canada: AAAI Press.

Summerville, A.; Snodgrass, S.; Guzdial, M.; Holmgard, C.; Hoover, A. L.; Isaksen, A.; Nealen, A.; and Togelius, J. 2018. Procedural Content Generation via Machine Learning (PCGML). *IEEE Transactions on Games*.

Toy, M.; Wichman, G.; Arnold, L.; and Lane, J. 1980. Rogue.

Wright, W.; Hutchinson, A.; Chalmers, J.; Gingold, C.; and Librande, S. 2008. Spore.