Program Synthesis as a Generative Method

Eric Butler
Paul G. Allen School of Computer Science and Engineering
University of Washington
edbutler@cs.washington.edu

Kristin Siu
School of Interactive Computing
Georgia Institute of Technology
kasiu@gatech.edu

Alexander Zook
School of Interactive Computing
Georgia Institute of Technology
a.zook@gatech.edu

ABSTRACT

Generative methods (also known as procedural content generation) have been used to generate a variety of static artifacts such as game levels. One key property of a generative method for a particular domain is how effectively the approach allows a designer to express the properties and constraints they care about. Generative methods have been applied much less frequently to dynamic artifacts such as boss behaviors, in part because the complex representation required to describe boss morphology and behavior is not amenable to existing generative techniques. It is challenging to describe a generative space of varied yet valid behaviors. Expanding on previous work that introduced a programming language for representing boss behaviors, we illustrate how such a language can be used by a designer to describe desirable design properties and constraints for bosses. That is, we define a generative space of bosses as a space of well-formed programs. We present a constructive algorithm that extends generative grammars to efficiently generate well-formed programs, and we show a complete example of generating *Mega-Man*-like bosses with complex attack patterns. We conclude that designing a generative space of dynamic behaviors can be fruitfully framed as a programming-language design problem.

Figure 1: An in-game screen capture of the *Moldorm* boss from *The Legend of Zelda: A Link to the Past.* Generating these types of boss encounters is challenging with current techniques due to the complex representation of boss behavior; bosses are defined by programs.

CCS CONCEPTS

•Applied computing →Computer games;

KEYWORDS

Program Synthesis, Generative Grammars, Generative Methods, Procedural Content Generation

ACM Reference format:
Eric Butler, Kristin Siu, and Alexander Zook. 2017. Program Synthesis as a Generative Method. In *Proceedings of FDG'17, Hyannis, MA, USA, August 14-17, 2017,* 10 pages.
DOI: 10.1145/3102071.3102076

1 INTRODUCTION

Generative methods [3] (often referred to as procedural content generation [20, 23, 27]) are algorithms used to generate artifacts of interest in some domain. Designers use generative methods to produce varied artifacts in a domain of interest (e.g., knitting patterns, *Mario Bros.* levels), with the generative method allowing the designer to describe their intent for generated output. The value of a method for a domain is often characterized by an *expressive range* [24]: the space of desirable artifacts a method produces while avoiding undesirable artifacts. While factors such as the ability to control the sampling of outputs or the efficiency of generation are important, a key feature is whether the designer can effectively and concisely express their intent over the generative space.

Boss encounters are compelling and dynamic highlights of a gameplay experience, serving as capstones for player skill and progression. We seek to generate boss encounters, which requires a method to capture the complex, state-driven behaviors of bosses. Generating bosses requires producing both the physical shape of a boss and the accompanying behavior that defines the way the boss acts and reacts to player actions. Existing generative methods have primarily been developed for producing static content: game levels, mazes, flora, terrain, and so on [20]. However, these methods are not well-equipped to generate complex boss behaviors due to representation required. In previous work we argued that a domain-specific programming language (DSL) with features such as variables references and function calls is an appropriate representation for boss morphology and behaviors [21]. Boss structure and properties map to state variables and boss behavior maps to program code.

The complex behavior of bosses requires an equally-complex representation. Consider *Moldorm* boss from *The Legend of Zelda: A Link to the Past*, shown in Figure 1. *Moldorm* runs randomly around the screen. Each time the player damages *Moldorm*, the boss speeds up. We use variable bindings to link different parts of *Moldorm*'s behavior to the common state tracking *Moldorm*'s speed. *Moldorm*'s state is represented by both basic types such as numbers and structured elements such as the physical objects describing its shape. The *Moldorm* example captures a wide array of boss archetypes across games, from stage bosses in *MegaMan* to Eggman in the 2D *Sonic* games to Dracula in the *Castlevania* series. In all cases, these entities are composed of structured behavioral elements referencing many structured components and state. We want a generative space of bosses to encompass the variety of qualitatively different ways these elements can be combined.

While research has explored generating behavior [4, 7, 29], we are not aware of existing methods that can generate bosses as complex as *Moldorm*. To generate bosses, we need a generative method that both (1) allows a designer to describe a rich but constrained space of programs representing boss behaviors, and (2) can be efficiently sampled.

In this work we contend that a DSL can represent complex boss behaviors [21] and supports effectively defining the *generative space* of boss behaviors as well. The properties and restrictions that define a program being well-formed (e.g., variable references pointing to in-scope variables, function arguments being of the correct type) can define desirable properties of boss design. A DSL with variable references and functions allows for a large possibility space and, just as importantly, allows a designer to express useful constraints.

For example, when a *Moldorm*-like boss increases some variable state after being damaged, it must refer to an actual piece of variable state referencing a number. The rules of well-typed programs, with valid variable references and where functions have appropriately typed arguments, naturally capture these constraints. Furthermore, we can intentionally choose the types and structure of our language to support designer constraints; for example, we may not want to increment (increase) the number representing the boss's health for bosses like *Moldorm*.

In this paper, we show how to express this and other example constraints with types, where any well-formed program will satisfy the design constraints. These constraints can be combined with syntactic restrictions (specified through, e.g., grammar production rules [5]) to define an expressive space of boss designs. In this way, we frame generating bosses as a problem of *program synthesis*, or generating code.

Our primary claim in this paper is that, for the domain of boss behaviors, standard programming language type theories such as variable bindings and functions allow us to define an expressive space with useful constraints. We present a generative system that combines grammar production rules with restrictions expressed through the theory of well-formed programs to express desirable constraints for generating bosses. We demonstrate the system with implementations of two boss domains—*Moldorm* and *MegaMan*—generating valid bosses in these domains.

We are developing this framework to facilitate authoring in generative domains. As such, our approach complements generative methods such as search-based methods or generative grammars. While we present one implementation for generating well-formed programs representing bosses, we expect defining generative spaces through designing DSLs to be compatible with other methods. We substantiate our claim with the following contributions:

- We frame generative tasks as program generation, where the design of domain-specific languages and type systems defines a rich generative space, illustrated through concrete examples.
- We contend that context-free grammars extended with type theories are suitably expressive yet restrictive to define a generative space while allowing efficient sampling, presenting a constructive algorithm to sample programs in this theory.
- We show a complete example of a generated *MegaMan*-like boss from our system, demonstrating the feasibility of using this method for generation.

Section 2 discusses related work. Section 3 reintroduces our programming model for bosses, establishing why a complex representation is needed. Section 4 gives examples of desirable properties and constraints that can be specified within the theory of well-formed programs, and shows that where syntactic constraints do not suffice, we can use type constraints to express them. Section 5 describes how we can efficiently generate well-formed programs in a constructive way with a generative grammar-based system. Before concluding, we show in Section 6 a complete example of a generated boss behavior for a *MegaMan*-style boss, demonstrating the feasibility of defining a generative space with the DSL.

2 RELATED WORK

Generative Grammars. A common approach to generating artifacts with syntactic constraints among sub-components of the artifact (such as programs) are *generative grammars* [19], particularly *context-free grammars* (CFGs) [2, 16]. Compton et al. *Tracery* [2] enables casual creators to easily author bots and other generative systems, such as natural language and other media. Ryan et al. *Expressionist* [15, 16] allows marking grammar elements with user-provided tags to provide expressiveness in generating natural language beyond what CFGs provide. Dormans [5] uses a graph grammar to define the missions and topology of game spaces. Our system similarly requires graph grammars because the boss programming model includes finite-state-machine graphs.

Many of these systems are built on top of context-free grammars, but these systems often support (or require) context-sensitive extensions to the grammar production rules to support design goals. In some cases these are specific to the design domain (e.g., [16]) and sometimes more general computation. In our application of DSLs, CFGs are expressive enough to describe the space of syntactically correct programs, but the space of syntactically correct programs contains an overwhelming number of uselessly invalid programs that must be avoided. More powerful tools such as *attribute grammars* [22] are sufficient to express constraints such as that all variable references are valid and point to in-scope variable declarations.

Our system uses CFGs to specify a generative space, here for well-formed programs with variable bindings and a type system. This supports a range of generative models and scenarios not easily captured by prior work, specifically, programmatic behaviors such as boss encounters in games. In principle we can implement a generator in terms of a generator for a more powerful grammar (e.g., attribute grammars). Instead, our algorithm extends a CFG generation algortihm with a domain-specific extension to support robust type system and variable binding. We implement an extension to CFGs here both for illustrative purposes and to support particular technical elements for application-specific considerations.

Generating Behaviors. Game generation research has also addressed the problem of representing and generating behavior, though these systems are typically restricted to narrow design spaces. Cook et al. [4] use program reflection to modify game code and test the resulting new code, but only validate games through randomized testing, rather than ensuring semantic validity. Zook and Riedl [29] use constraint solving to generate game mechanics from a domain. This model includes features similar to ours such as variable binding, but uses ad-hoc constraints. We present a systematic theory that could be implemented in constraint programming, but also used in other contexts like generative grammars. Treanor et al. [28] define a wide range of games using grammar expansion with post-generation refinements, using a carefully authored generative grammar to ensure valid outputs. Togelius and Schmidhuber [25] and Browne and Mare [1] both employ search-based generative methods over a range of game designs, requiring fitness function evaluations to assess whether valid games are produced. Gellis [7] generates boss behaviors for a jumping and shooting enemy using search-based methods. Our work can be seen as a way to augment the bulk of these existing efforts to better encapsulate designer intuition and constrain the space of generated artifacts more readily by using standard type theories.

Game Description Languages. Researchers have developed languages to facilitate authoring game behaviors for different domains, such as *Façade*'s A Behavior Language [11], the Versu storytelling social model [6], *Prom Week*'s "social physics" model [12], and Ceptre's linear-logic model [10]. Like our DSL these are typically tailored towards particular game designs, but they are designed for authoring rather than generating. A broader class of game description languages have been proposed for the purpose of describing mechanics and analyzing or generating games [1, 9, 13, 17]. These languages aim to model a broad, general class of games, at least for a particular genre. In contrast, we specifically propose that game languages be tailored to the particular game and generation task to define a generative space.

Program Synthesis. The code generation we describe is a form of *program synthesis*, the task of discovering an executable program (in some specified language) that realizes user intent in the form of some specification [8]. Some applications focus on realizing complete specifications, such as super-optimization [18], while other applications have the challenging problem of an underspecification, such as programming by example [14]. Our problem is unconventional because it has a much weaker specification: *any* well-formed program that fits within the syntactic constraints provided by the designer. Where typical synthesis applications rely on search or constraint solving, our particular specification enables us to use an efficient constructive algorithm.

3 A PROGRAMMING MODEL FOR DEFINING BOSS BEHAVIORS

Programs are a natural specification for agent behavior, particularly relevant for the problem of generating boss behavior. Here we briefly describe our programming model for bosses (detailed in previous work [21]). We generalize the *Moldorm* example to the domain of 2D, physics-based boss encounters, typically defined through kinematic physics, collision detection, and finite-state-machine-driven behavior. This domain illustrates how a programming model with rich constructs such as variables can adequately define its behavior.

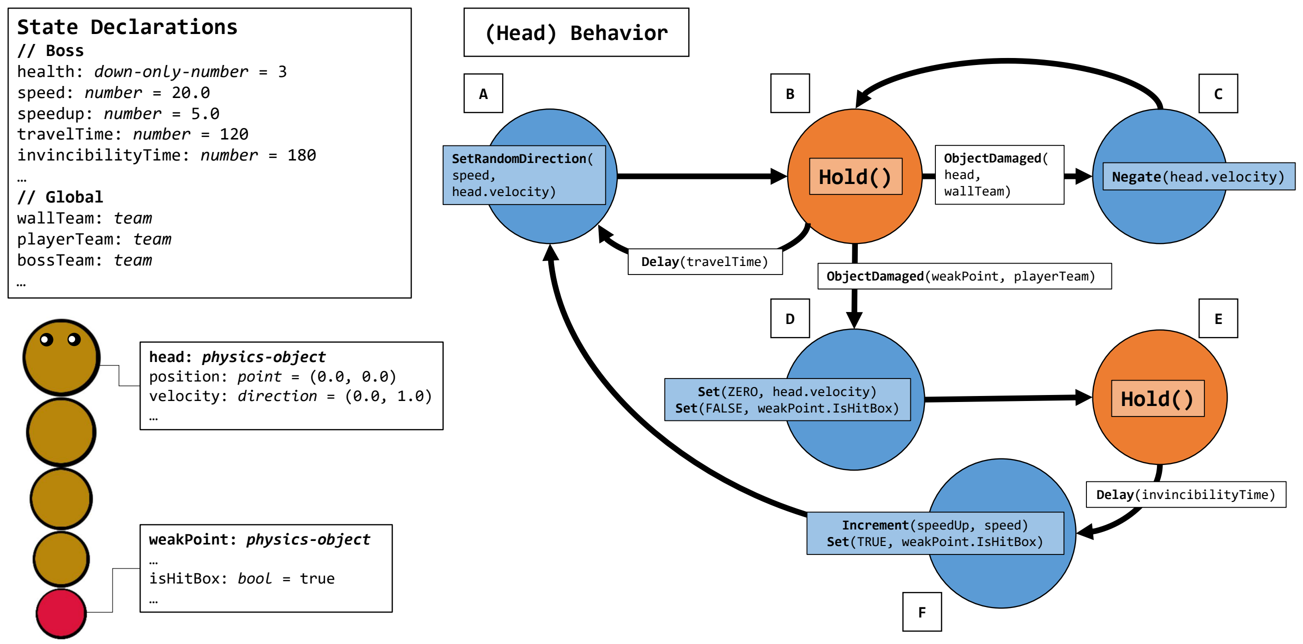
Behaviors are specified by finite state machines. Finite state machines are graphs whose nodes are behaviors (e.g., negate this vector, increment this number, do nothing, track another object every frame) and whose edges are conditional transitions between behaviors (e.g., after *n* seconds, when this hurtbox collides with this hitbox). Behaviors operate on mutable state of the boss or environment, consisting of primitive types (e.g., numbers used to track "boss health") and complex components (e.g., physical objects to describe player and boss morphology). Some behaviors include instantaneous updates when they activate (e.g., negating a vector) while others change state over time while active (e.g., track another object every frame).

The programming model for bosses is a programming language for defining both the state and behavior graphs. Figure 2 shows an example program defining the behavior for the *Moldorm* boss. The program has declarations for the state (both primitive types like numbers and physical objects like *Moldorm*'s head) and declarations for behavior graphs describing how the state and behaviors are related. These graphs are part of the syntax; the language includes both tree-based and graph-based syntactic elements.

Programs in this model are fully evaluated before the game begins. The program connects components, state, and behaviors together. During the update loop, the semantics of these behaviors and objects are given by separate implementations of these behaviors and a physics system. Thus, the function call `Increment(speedUp, speed)` does not evaluate to immediately increment `speed` by `speedUp`. Instead, it evaluates to a *behavior node* object, which, during gameplay, will increment the current value of `speed` by `speedUp` whenever that node of the finite state machine is active. Describing a boss can thus be viewed as combining a set of pre-built components and behaviors with complex connections.

The programming model provides a type system, with function calls (used, among other places, when creating the behavior nodes and edges), variable declarations and references (so behaviors can reference shared state), and a basic set of types (because there is a rich variety of possible objects to reference). We support the theory of *subtypes*: if type σ is a subtype of type τ, then any value of type σ is also of type τ. For example, *integers* and *reals* are both subtypes of *number*. We can use subtypes to express a range of designer-relevant constraints (see Section 4.2).

The example above of *Moldorm* speeding up when damaged uses each of these features. The program declares a variable `speed` of type *number* (Figure 2). Node **A** references `speed` in the `SetRandomDirection(speed, head.velocity)` function call, which evaluates to a behavior node that will set *Moldorm*'s head's velocity to the current value



```
State Declarations
// Boss
health: down-only-number = 3
speed: number = 20.0
speedup: number = 5.0
travelTime: number = 120
invincibilityTime: number = 180
…
// Global
wallTeam: team
playerTeam: team
bossTeam: team
…
```

```
head: physics-object
position: point = (0.0, 0.0)
velocity: direction = (0.0, 1.0)
…
```

```
weakPoint: physics-object
…
isHitBox: bool = true
…
```

(Head) Behavior
A
SetRandomDirection(speed, head.velocity)
B
Hold()
ObjectDamaged(head, wallTeam)
C
Negate(head.velocity)
Delay(travelTime)
ObjectDamaged(weakPoint, playerTeam)
D
Set(ZERO, head.velocity)
Set(FALSE, weakPoint.IsHitBox)
E
Hold()
Delay(invincibilityTime)
Increment(speedUp, speed)
Set(TRUE, weakPoint.IsHitBox)
F

Figure 2: Visualization of the program that defines the *Moldorm* boss. The program uses variable declarations, function calls, and variable references to describe the complex behavior of the boss. Body behaviors have been omitted for brevity.

of speed. Node **F** includes a function call to `Increment(speedUp, speed)`, which evaluates to a behavior node that increments speed. By framing generation as a programming model we ensure all generated bosses are *well-formed* programs, meaning syntactically correct programs that are well-typed with valid variable references. This ensures boss generation only considers behavior combinations that have valid behavior choices, rather than generating these options and needing to subsequently evaluate their validity.

4 EXPRESSING DESIGN CONSTRAINTS AS WELL-FORMED PROGRAM CONSTRAINTS

In this section we show how features of and constraints on boss structure are readily expressed as definitions of well-formed programs. The constraints we describe are sometimes obvious, but are difficult or impossible to specify in less expressive grammars or unnatural and complex to express in more powerful grammars. We show how to design a type system to express constraints for boss generation (e.g., the boss's weak point should be part of the boss rather than the player or environment). Our examples use types in ways not supported by common programming languages, motivating the development of our system.

Our running example is *Moldorm*-like bosses that share the following features:

(1) Bosses have a complex morphology, consisting of multiple shapes that may move independently.
(2) Bosses have *weak points*, one or more parts of their morphology that can (periodically) be damaged by the player.
(3) Bosses react to damage, *escalating* their behavior with successive hits (e.g., by moving faster).

We discuss how each feature of our programming model can express constraints relevant to this generative space. These constraints take the form of grammar production rules. In Section 6, we provide a case study in another domain: generating *MegaMan*-like bosses. *MegaMan*-like bosses have simple morphologies, but more complex movement and attack patterns involving running, jumping, and projectile shooting (creation and movement).

4.1 Expressiveness of Variable References

Variable references provide an expressiveness not afforded by simpler representations by enabling a generator to refer to previously generated parts of the program.

4.1.1 Weak Points. Consider defining a boss's *weak point*, the physical part of the boss where the player can inflict damage to the boss by colliding their weapon. Our model uses the `ObjectDamaged(object,opponent)` condition for this, defining a transition to take when a hurtbox belonging to **opponent** (e.g., the player) collides with the hitbox specified by **object**. As we are defining a generative space, we want the weak point to be selected randomly from the boss's morphology, which is also generated.

How can we express the choice of a random, generated component in a behavior? Context-free grammar rules are not sufficient, because the set of possible objects relies on the context of what morphology was generated. Instead, we can express this with variable references (Figure 3). The key part in the production rule for our weak point is defining the object as a reference (notated as `Alias` in the figure) to an existing declaration. The typing rules constrain this reference to be of type **physics-object**, so any program

```
// declaration for the ObjectDamaged function
fun ObjectDamaged :
    (physics-object, team) ->  behavior-edge

// production rule for generating boss morphology
// declares a variable of type physics-object
BossMorphology ->
    declare physics-object = ...

// production rule for generating a weak point
DamageCondition ->
    ObjectDamaged(
        #Alias#, // reference to a physics-object
        player) // hard-coded team
```

Figure 3: Pseudocode illustration of portions of two of the grammar production rules involved in defining a generated *weak point* for a boss. The boss is constructed from a set of declared physics objects. Choosing a weak point relies on this context. The *Alias* non-terminal describes any valid variable reference. Because `ObjectDamaged` only accepts physics objects, this production rule describes only aliases to morphology.

generated will reference a valid generated **physics-object**. This allows us to express a generative space where bosses can have an arbitrarily generated morphology and the weak point may be any part of that morphology.

4.1.2 Escalation. Consider another case for variable references: the *escalation* after a boss is damaged. *Moldorm*'s escalation is `Increment(speedUp,speed)`, meaning the variable state used when setting *Moldorm*'s speed is incremented by a fixed value after damage. How can we generate state altered by *escalation*? As with our weak point example, we can use an `Alias` non-terminal to automatically select a previously declared numerical variable and increase it. As numbers may be used in all sorts of contexts (boss size, speed, attack duration, attack power, time period between vulnerability), this expresses a wide variety of potential behavior. We use the same rule in our *MegaMan*-like example in Section 6.

4.2 Constraints via Typing

Types provide basic guarantees where, for instance, the arguments to functions are sensible. We can use a richer structure of types than is typically used when programming bosses in general purpose languages to constrain our generative space. For example, a two-dimensional **vector** type is a common structure to represent both physical position and velocity, supporting operations such as vector addition or negation. However, not all vectors are the same; while it is sensible to negate a velocity, it is usually not sensible to negate an object's position. Instead, we can use *subtypes* to guarantee such constraints, defining two vector subtypes: a **point** (e.g., position) and a **direction** (e.g., velocity). In our DSL, the `Negate` function only accepts a **direction**, preventing negation of positions. Below we illustrate the value of subtypes for defining weak points and escalation behavior.

4.2.1 Weak Points, Revisited. The rules in Figure 3 work when the only declared physical objects are for the boss. But what about the case where we also declare the environment and the player? In this case the weak point could appear on the player.[1] The declared type of `ObjectDamaged` ensures that only a **physics-object** will appear as its first argument. We can use typing to create a constraint to prevent player or environment objects from being referenced by `ObjectDamaged`.

We create an appropriate constraint by introducing a new type **boss-object** as a subtype of **physics-object** (meaning all **boss-object**s are **physics-object**s, as in Figure 4). We modify the declaration of boss morphology, annotating the component objects to be of type **boss-object**. We then modify the production rule for the weak point, annotating that it specifically references a **boss-object**. This prevents the player or environment objects (both of type **physics-object** but not **boss-object**) from being used.

```
type boss-object is-subtype-of physics-object

// specifically annotate as boss-object type
BossMorphology ->
    declare boss-object = ...

// constrain reference to be boss-object type
DamageCondition ->
    ObjectDamaged(
        #Alias:boss-object#,
        player)
```

Figure 4: Variant of rules in Figure 3 that use *subtypes* to further constrain the weak point to necessarily be part of the boss's morphology. This is necessary if the program includes declarations for other physics objects, such as the player or the environment.

4.2.2 Escalation, Revisited. Our escalation generation expresses a wide range of potential behavior, including many valid, but undesirable cases. A common numeric variable for bosses is the boss's "health." When health reaches zero, the fight ends. Escalation, as currently specified, allows us to generate a boss where the health is incremented by the escalation each time the boss is damaged. We can prevent this undesired boss behavior by constraining health to only decrease via subtypes.

We create two subtypes of **number**: numbers that may only be decremented (**down-only-number**) and numbers without this restriction (**unrestricted-number**). We then adjust our production rules: first, we annotate nearly every numeric literal (except health) with **unrestricted-number**. We then change the type declaration for `Increment` so that it only accepts values of type **unrestricted-number**. Finally, we annotate the declaration of the variable representing health to be of type **down-only-number**. Now, our production rule for randomly incrementing a number is constrained to avoid health. The corresponding `Decrement` function, on the other hand, can accept any **number**.

[1]Undesirable in this case, though it is an aesthetically fascinating design for a boss encounter.

These types are of limited use in general programming, but of great value for specifying this particular constraint for boss behaviors. The power of using a DSL is that designers can control the type system, introducing domain-specific types to enforce domain-specific design considerations.

4.3 Boss Behavior Generative Space Design is Programming Language Design

The examples shown in this section express design properties and constraints for boss encounters in terms of well-formed programs. That is, we have defined our generative space for bosses to be the space of all valid programs in our DSL. We do this by choosing the constructs, types, features, and grammar of our DSL to capture our domain-specific design goals. The process of designing the generative space is one of designing the domain-specific language.

5 GENERATING WELL-FORMED PROGRAMS

Our primary claim is that the semantics of well-formed programs are an effective tool to describe generative spaces for boss designs. This is only useful if we can efficiently generate well-formed programs, so we describe one method for generating programs here. We generate boss behaviors constructively to enable efficient sampling of variable output. Generative grammars, particularly expressive ones such as attribute grammars [22], are capable of describing well-formed programs. We instead extend context-free grammars to express well-formed programs directly. We do this both for illustrative purposes as well as to support extensions for application-specific design considerations (Section 6.2). We used this method to generate the example in Section 6.

In a context-free grammar, a context-free *production rule* is defined by a *non-terminal* symbol to be expanded and a tree (or graph for graph grammars) of non-terminals and terminals to replace it with. Generating a program for a given set of production rules starts with a single non-terminal representing the entire program. Each step recursively applies rules until only terminals remain. In a context-free setting, a rule may be applied to any non-terminal that matches the non-terminal of the rule. However, the rules we use to generate well-formed programs must be more restrictive, ensuring, for example, variable references must be valid. Below we describe these extra conditions under which it is safe to apply a production rule to a given non-terminal in a partially expanded program. We focus on the tree grammar; the extensions to a graph grammar are straightforward.

5.1 Type-Aware Grammar Expansion

In the tree grammar for programs, non-terminals represent program expressions. Note that we are not generating function definitions; these are supplied along with the production rules. A given concrete expression has some determinable type τ that is one of:

(1) A literal of type τ.
(2) A function call or other compound construct (with zero or more sub-expressions for arguments) that returns τ.
(3) A variable declaration of type τ (with a sub-expression of type τ describing the value)
(4) A reference to an existing variable of type τ.

We can efficiently determine the type of any concrete expression, with only limited context of the rest of the program.

For generation we can reverse this process. Given a desired type for an expression we are generating, for each possible generated outcome we know:

(1) Which literals can be of that type.
(2) Which function calls may be of that type, and what types their arguments must have to be that type.
(3) Which type the value sub-expression of a variable declaration must be.
(4) Which in-scope variables have that type.

If we know which type an expression is supposed to be, we can filter productions to those with valid types. Determining which literals and functions can be of appropriate type is straightforward; we treat variable bindings in more detail in Section 5.3.

5.2 Determining the Required Type of Non-Terminals

Our algorithm must determine the allowable type of each non-terminal to filter possible productions to only those of suitable type. We can get the required type from context. Top-level expressions have known required types based on the language structure. For example, expressions for nodes of the behavior graphs must be of type **behavior-node**. All sub-expressions can only appear as function arguments. Given the required type of the parent function call (which we know), we can determine the required type of each of the parameters, and thus the required type of the current expression.

Our implementation also supports annotating production rules with more restrictive types, as in Figure 4. If such an annotation exists, the intersection of the annotated type and the context-derived type is used.

5.3 Generating Variable References

Variable declarations and references require special care during code generation as well-formed variable references require global context of the program. To generate well-formed variable references, the algorithm must track all valid variables at each point in the program.

To support grammar expansion that produces well formed variable bindings, we make the two assumptions about the programming model and the grammar production rules. First, the algorithm must know the scoping rules for the DSL. Our DSL for bosses uses lexical scoping rules typical of what one would find in general purpose languages. Second, the algorithm must know which constructs in the DSL are declarations or references. Declarations are described by a given identifier and a given type for the variable being declared. References are described by a given identifier, which, in well-formed programs, should match the name of a previous, in-scope declaration whose type is a subtype of the required type of the reference expression.

Our algorithm tracks the current variable context for each point in the program. Whenever a generated terminal is a declaration, its variable is added to the appropriate contexts. When expanding a non-terminal (with required type τ) whose productions include terminals that are references, the possible productions are limited to those referencing in-scope variables of a subtype of τ.

Our system fully expands earlier points in the program before later points (an in-order tree traversal) to have the most complete knowledge about which variables are in scope. This is not necessary for soundness; the algorithm can always ignore any potential production and still guarantee a well-formed program. But expanding in a different ordering can lead to a situation where no references are available (though they would be in a different order), leading to failure.

6 A COMPLETE EXAMPLE OF A GENERATED BOSS

To demonstrate the feasibility of defining a generative space for bosses with a programming model, we applied the algorithm described in the previous section to generating *MegaMan*-like boss encounters, in which an enemy moves, jumps, and shoots in a series of attack patterns in a side-scrolling 2D room. In this section, we walk through a complete example of a generated artifact from this space, shown in Figure 5. We explain in detail various parts of the rules used to generate the artifact and how the generative space used in this example relied on the theories of well-formed programs to be expressed.

6.1 Walking through the Example

We define the type of boss we generate here with several properties (contrasted to our previous *Moldorm*-like example):

(1) Simple morphologies, where bosses are a single box.
(2) Complex movement patterns and behavior, with running and jumping around a side-view 2D room.
(3) Projectile shooting, where both the shooting behavior and the properties of the projectiles themselves can be varied and complex.
(4) *Escalation*, similarly to *Moldorm*. As the boss is damaged by the player, properties of the attack patterns (e.g., speed, damage, frequency of shots) increase.

This example combines the features discussed in Section 4 to describe its generative space.

6.1.1 Defining Movement and Attacks. Movement and attack behaviors are described using grammar rules. *MegaMan* bosses may also damage the player during movement, so we refer to both patterns of behavior as *attack patterns*. Since these behaviors are graphs, we use *graph grammars*, which have been applied, for example, to generating missions for *Zelda* dungeons [5].

There are three possible productions for an attack pattern: (1) moving for some duration, (2) jumping, and (3) shooting a projectile. Examples in our artifact are nodes **1G**, **1C**, and **1B**, respectively. The production rules for moving/jumping also allow shooting inside of them. The overall structure of any *attack behavior graph* is that the boss randomly selects a pattern (node **1A**), executes it fully, then returns to node **1A** to select another). The use of grammar rules for attack patterns allows for a rich structure of distinguishable behavior. The generative space includes a variety of structurally varied combinations of these basic elements.

6.1.2 Parameterizing Attack Patterns. Each part (i.e., node) of an attack pattern is parameterized. Some parameters are numbers, such as the duration between actions, the movement speed, the initial jumping speed, or projectile speeds. Other parameters include the orientation of a projectile shot (e.g., left or right) or the projectiles themselves. All parameters are generated as arbitrary expressions, either as variable references (such as `speed1` in **1C**) or function calls (such as `RandOrientation` in the same node).

Consequently, the parameters occasionally reference each other. For example, the move speed used in node **1G** shares a reference with the speed of projectile `p3`. Our algorithm ensures all generated artifacts are well-formed, so all of these variable references will also point to valid declarations. The expressiveness of our representation allows for generative behavior like the sharing of references.

Moreover we take advantage of subtypes (like the examples in Section 4.2) to restrict this variable sharing to sensible values. Some numerical parameters represent a **speed** (e.g., projectile speed), while others represent a **duration** (e.g., the delay between nodes **1H** and **1I**). This is a beneficial constraint because the reasonable range of values for durations (in seconds) is very unlikely to be similar to the reasonable range for speed (in game units per second).

The bosses in this space can fire a variety of projectiles in ways that can be based on state. For example, the pattern in **1B** randomly chooses between any of the 3 projectiles while **1H** always fires the second projectile. As with numerical references, our algorithm guarantees these projectile references are sensibly connected.

6.1.3 Escalation. As with *Moldorm*, the bosses we generate here escalate their behavior when damaged. Mechanically, these bosses differ from *Moldorm* in that they do not modally react to damage, and damage does not interrupt movement or attacks. This is expressed with a parallel *escalation graph* rather than being integrated in a single graph as *Moldorm* is.

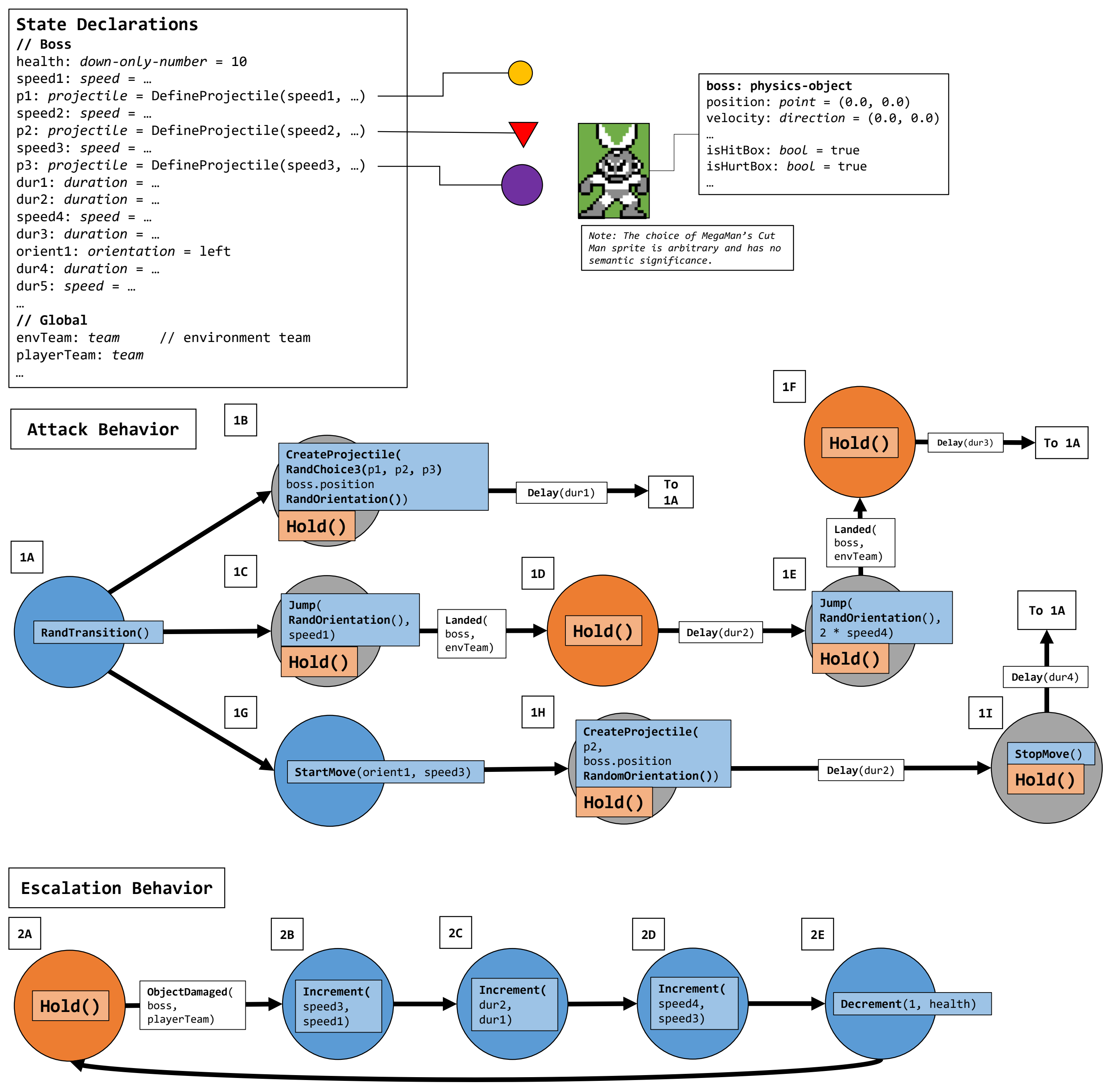
As in Section 4.1, our production rules for this example force an **Alias** for the argument to `Increment`, which expressed any valid variable reference to a previously declared number. Any of the numerical values used in the attack patterns can be referenced. For example, as can be seen in Figure 5, the nodes **2B**, **2C**, and **2D** increment parameters including the speed of projectiles, jumping speed, and the delay after shooting. In this particular case, `speed3` is both incremented and used as the value to increment `speed1`, meaning `speed1` increases superlinearly. Finally, we use the strategy of having a **down-only-number** type (explained in Section 4.2.2) to prevent the `health` value from being incremented.

Different artifacts will escalate different aspects of the boss's parameters. These are qualitatively different experiences: one where the enemy speeds up, one where the enemy does more damage, one where the enemy's projectiles get faster. This complexity is enabled through our DSL representation.

6.2 Extensions to the Method

Lastly, we discuss a few extensions to the basic algorithm discussed in Section 5 that we found particularly helpful in defining the generative space for this example.

6.2.1 Promoting variable reuse with automatic declarations. As our examples illustrate, these bosses draw their complex behaviors by having multiple parts of their programs reference shared variable state. Assuming a declaration was generated, we can encourage reuse of it with production rules that force the use of references,



```
State Declarations
// Boss
health: down-only-number = 10
speed1: speed = …
p1: projectile = DefineProjectile(speed1, …)
speed2: speed = …
p2: projectile = DefineProjectile(speed2, …)
speed3: speed = …
p3: projectile = DefineProjectile(speed3, …)
dur1: duration = …
dur2: duration = …
speed4: speed = …
dur3: duration = …
orient1: orientation = left
dur4: duration = …
dur5: speed = …
…
// Global
envTeam: team     // environment team
playerTeam: team
…
```

```
boss: physics-object
position: point = (0.0, 0.0)
velocity: direction = (0.0, 0.0)
…
isHitBox: bool = true
isHurtBox: bool = true
…
```

Note: The choice of MegaMan's Cut Man sprite is arbitrary and has no semantic significance.

Attack Behavior
1A
RandTransition()
1B
CreateProjectile(RandChoice3(p1, p2, p3) boss.position RandOrientation())
Hold()
Delay(dur1)
To 1A
1C
Jump(RandOrientation(), speed1)
Hold()
Landed(boss, envTeam)
1D
Hold()
Delay(dur2)
1E
Jump(RandOrientation(), 2 * speed4)
Hold()
Landed(boss, envTeam)
1F
Hold()
Delay(dur3)
To 1A
1G
StartMove(orient1, speed3)
1H
CreateProjectile(p2, boss.position RandomOrientation())
Hold()
Delay(dur2)
1I
StopMove()
Hold()
Delay(dur4)
To 1A
Escalation Behavior
2A
Hold()
ObjectDamaged(boss, playerTeam)
2B
Increment(speed3, speed1)
2C
Increment(dur2, dur1)
2D
Increment(speed4, speed3)
2E
Decrement(1, health)

Figure 5: Visualization of a literal description of a example generated program for a *MegaMan*-like boss. The first graph defines attack patterns; the boss randomly selects one of 3 patterns involving combinations of running, jumping, and/or shooting. The second graph *escalates* the encounter when the boss is damaged by incrementing variables referenced by the attack patterns. This complex behavior is expressible because of variable references, and type constraints are used to constrain which ones may be incremented.

as in our example of escalation in Figure 5. But the state being referenced must be referenced twice to be shared.

This presents a small challenge: we generate the program in-order, so, to use variable state, the generator must fortuitously declare a variable of appropriate type before any of the behaviors that might want use this variable are generated. In a sense, the generator is making the decision backwards: a designer might typically choose the behavior first, then the variable state necessary to implement it, not a set of behaviors that happen to be compatible with arbitrarily chosen state.

To more easily support variable re-use, our generator automatically promotes literals to declarations. That is, any time it is expanding an expression and selects a production of a literal value, it automatically converts this to a variable declaration in the current scope and creates a reference to that variable. In this way, any literal in the system has the potential to be referenced by other code. A designer can encourage this with production rules that force variable references as we do with escalation (Section 4.1.2).

6.2.2 Generating Literals After Generating Structure. There are many literal parameter values to choose when generating a boss behavior. With the example in Figure 5, literals control the projectile shapes and speeds, the boss size, their movement speed, jumping speed, timings, and many other values. While these literals are of course irrelevant from the perspective of ensuring well-formed programs, they are critical for boss design.

There are many useful design constraints we might wish to impose on these literals. One example is to restrict the overall duration of a single attack pattern to be within some designer-specified probability distribution, defined by the sums of all of the `Delay` nodes in a subgraph for the attack pattern. While generative grammars are useful for constraining program structure, it is challenging to enforce this constraint during program generation with production rules because we do not know how many nodes will be generated, especially if we want to control the distribution. But, if we have a fixed program structure, it is straightforward to analyze the structure and enforce such constraints on the literal values.

One option for generating literals is, rather than doing so during program generation, to leave *holes* in the program for the literal values. After the structure is fixed, we can extract the set of holes and use a different generative method to choose the particular values, while being able to use the program structure as information to help guide and constrain the generation. The resulting flat parameter space is amenable to a wide variety of methods, such as search-based methods that optimize fitness functions [26, 27].

7 CONCLUSIONS

In this paper we advance the thesis that programming languages are a useful and powerful model for expressing generative spaces, specifically for domains characterized by complex state and behaviors like boss encounters. We demonstrated how the constraints of well-formed programs can afford suitable authorial control over generative spaces. Moreover, we presented an algorithm that extends context-free grammar generation to generate well-formed programs in a constructive way. To illustrate this approach we used a previously developed programming model for 2D boss behaviors [21] to examine several examples of design-relevant properties and constraints that could be expressed in the constraints of variables and types. We presented another example of a generated *MegaMan*-like boss to illustrate the feasibility of using this process.

The key idea for generating bosses is to make the space of well-formed, valid programs synonymous with the desired generative space of bosses. Defining this program space is defining the generative space. By choosing grammar, types, and language constructs to capture domain-specific design concerns, we modeled properties such as generated boss weak points and attack patterns while ensuring the resulting behaviors were runnable and sensible.

Our use of programming languages as a representation for generation is not limited to 2D boss behavior, and instead encompasses a broader class of generative methods with wider applications. In the future we foresee similar models being effective to generate game combat systems, overworlds with complex topology (e.g., in *Metroidvania* games), and other game systems that demand careful relationships between the semantics of their parts. We also believe there is room for improvement by developing and integrating heuristics to guide generation toward higher quality artifacts within the valid design space. We presented two separate ideas: one of synthesizing programs using grammars and one of representing a design space as a space of programs. We expect the latter to be usable in other generative methods and settings, expanding the range of what designers can create.

REFERENCES

[1] Cameron Browne and Frederic Maire. 2010. Evolutionary Game Design. *IEEE Transactions on Computational Intelligence and AI in Games* 2 (2010), 1–16.

[2] Kate Compton, Ben Kybartas, and Michael Mateas. 2015. Tracery: An Author-Focused Generative Text Tool. *Interactive Storytelling* 9445 (2015).

[3] Kate Compton, Joseph C Osborn, and Michael Mateas. 2013. Generative Methods. In *4th Workshop on Procedural Content Generation in Games.*

[4] Michael Cook, Simon Colton, Azalea Raad, and Jeremy Gow. 2013. Mechanic Miner: Reflection-Driven Game Mechanic Discovery and Level Design. In *EvoGAMES.*

[5] Joris Dormans. 2010. Adventures in level design: Generating missions and spaces for action adventure games. In *1st Workshop on Procedural Content Generation in Games.*

[6] Richard Evans and Emily Short. 2014. Versu fi!? A Simulationist Storytelling System. *IEEE Trans. Computational Intelligence and AI in Games* 6, 2 (2014), 113–130.

[7] Abraham Gellis. 2015. *Procedurally Generated Entity Behaviors for Game Content.* Master's thesis. New York University.

[8] Sumit Gulwani. 2010. Dimensions in Program Synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP '10).* ACM, New York, NY, USA, 13–24. DOI: http://dx.doi.org/10.1145/1836089.1836091

[9] Nathaniel Love, Timothy Hinrichs, David Haley, Eric Schkufza, and Michael Genesereth. 2008. *General game playing: Game description language specification.* Technical Report. Stanford University.

[10] Chris Martens. 2015. Ceptre: A Language for Modeling Generative Interactive Systems. In *11th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment.*

[11] Michael Mateas and Andrew Stern. 2002. *Architecture, authorial idioms and early observations of the interactive drama Façade.* Technical Report. Carnegie Mellon University.

[12] Joshua McCoy, Mike Treanor, Ben Samuel, A Reed, Michael Mateas, and Noah Wardrip-Fruin. 2014. Social Story Worlds With Comme il Faut. *IEEE Trans. Computational Intelligence and AI in Games* 6, 2 (2014), 97–112.

[13] John Orwant. 2000. EGGG: Automated programming for game generation. *IBM Systems Journal* 39, 3.4 (2000), 782–794.

[14] Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN Inter. Conf. on Object-Oriented Programming, Systems, Languages, and Applications.* ACM.

[15] James Ryan, Michael Mateas, and Noah Wardrip-Fruin. 2016. Characters Who Speak Their Minds: Dialogue Generation in Talk of the Town. In *12th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment.*

[16] James Ryan, Ethan Seither, Michael Mateas, and Noah Wardrip-Fruin. 2016. Expressionist: An authoring tool for in-game text generation. In *Interactive Storytelling: 9th International Conference on Interactive Digital Storytelling, ICIDS 2016, Los Angeles, CA, USA, November 15–18, 2016, Proceedings 9*. Springer, 221–233.

[17] Tom Schaul. 2013. A Video Game Description Language for Model-based or Interactive Learning. In *IEEE Conference on Computational Intelligence in Games*.

[18] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic Superoptimization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 305–316. DOI: http://dx.doi.org/10.1145/2451116.2451150

[19] Noor Shaker, Antonios Liapis, Julian Togelius, Ricardo Lopes, and Rafael Bidarra. 2015. Constructive generation methods for dungeons and levels. In *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer.

[20] Noor Shaker, Julian Togelius, and Mark J. Nelson. 2015. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer.

[21] Kristin Siu, Eric Butler, and Alexander Zook. 2016. A programming model for boss encounters in 2d action games. In *Experimental AI in Games Workshop*, Vol. 3.

[22] Kenneth Slonneger and Barry L Kurtz. 1995. *Formal syntax and semantics of programming languages*. Vol. 340. Addison-Wesley Reading.

[23] A.M. Smith and M. Mateas. 2011. Answer set programming for procedural content generation: A design space approach. *IEEE Transactions on Computational Intelligence and AI in Games* 3, 3 (2011), 187–200.

[24] Gillian Smith and Jim Whitehead. 2010. Analyzing the expressive range of a level generator. In *1st Workshop on Procedural Content Generation in Games*. ACM, 4.

[25] Julian Togelius and Jürgen Schmidhuber. 2008. An experiment in automatic game design. In *IEEE Symposium on Computational Intelligence and Games*.

[26] Julian Togelius and Noor Shaker. 2015. The search-based approach. In *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer.

[27] Julian Togelius, G. Yannakakis, K. Stanley, and Cameron Browne. 2011. Search-based Procedural Content Generation: A Taxonomy and Survey. *IEEE Transactions on Computational Intelligence and AI in Games* 3, 3 (2011), 172–186. DOI: http://dx.doi.org/10.1109/TCIAIG.2011.2148116

[28] Mike Treanor, Bobby Schweizer, Ian Bogost, and Michael Mateas. 2012. The micro-rhetorics of Game-O-Matic. In *7th International Conference on the Foundations of Digital Games*.

[29] Alexander Zook and Mark O. Riedl. 2014. Automatic Game Design via Mechanic Generation. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence*.