Gemini: Bidirectional Generation and Analysis of Games via ASP

Adam Summerville[1] Chris Martens[2] Ben Samuel[3] Joseph Osborn[4]
Noah Wardrip-Fruin[5] Michael Mateas[5]
California State Polytechnic University, Pomona[1] North Carolina State University[2] University of New Orleans[3]
Pomona College[4] University of California, Santa Cruz[5]
asummerville@cpp.edu martens@csc.ncsu.edu bsamuel@cs.uno.edu joseph.osborn@pomona.edu
nwf@ucsc.edu mmateas@ucsc.edu

Abstract

Current approaches to game generation don't understand the games they generate. As a result, even the most sophisticated systems in this regard, e.g., *Game-o-Matic*, betray this problem—generating games with goals that are at odds with their mechanics. We describe *Gemini*, the first bidirectional game generation and analysis system. *Gemini* is able to take games as input, perform a proceduralist reading of them, and produce possible interpretations that the games might afford. By utilizing the declarative nature of Answer Set Programming (ASP), this analysis pathway opens up generation of games targeting specific interpretations and makes it possible to ensure the generated games are consistent with the desired reading. For *Gemini*, we developed a game specification language capable of expressing a larger domain of games than is possible with VGDL, the most widespread representation. We demonstrate the generality of our approach by generating games in a series of domains. These domains are based on prototypes hand-created by a team without knowledge of the constraints and capabilities of *Gemini*.

Introduction

Procedural Content Generation (PCG) has been a part of games since 1978 (Worth 1978), but has mostly focused on generating pieces of games: textures, props, levels. However, there has been a push over the last decade toward generating entire games (Nelson and Mateas 2007; Togelius and Schmidhuber 2008; Treanor et al. 2012; Nielsen et al. 2015; Cook, Colton, and Gow 2017a). Many of these approaches generate games intended to have a specific interpretation, but this interpretation is by fiat; the generators march forward and say they achieved their interpretation but have no way of analyzing the generated game to ensure this.

To illustrate this idea, we look to the closest prior work in generating games with meaning, *Game-o-Matic* by Treanor et. al. *Game-o-Matic* accepts a desired meaning that a generated game is supposed to express (meanings are expressed as concept diagrams), and generates simple games whose mechanics are supposed to express that meaning. In Treanor's dissertation (Treanor 2013), he discusses an example of a game meant to match the dynamics of the Occupy Wall Street movement. While *Game-o-Matic* is capable of producing sensible games (e.g., as Treanor (Treanor 2013) relates, "As the occupy movement grows to fill the screen, overwhelming the police forces, removing Wall Street happens without any actions from the player"), it is also capable of producing nonsensical games (e.g., the goal is to have the player, as the police, push occupiers to Wall Street—which causes the occupiers to grow). By operating in a pipelined manner in a single direction from desired meaning to generated game mechanics, *Game-o-Matic* is incapable of reasoning fully about meanings expressed by generated games, leading to such problems. A system operating in this forward manner would need to codify all possible combinations of choices, feasible in only the narrowest of toy domains, due to the combinatorial nature of these choices.

In this paper we present *Gemini*, the first bidirectional game analysis and generation system, capable of providing interpretations of existing games and generating games to meet a specific interpretation. *Gemini* uses Answer Set Programming (ASP), a declarative programming paradigm which enables the bidirectionality

- **Analysis Path:** Game $\rightarrow$ { Interpretation }
- **Generation Path:** Interpretation $\rightarrow$ { Game }

with the same code enabling both paths. Earlier approaches only generated games with entity-to-entity interpretations (Nelson and Mateas 2007; Treanor et al. 2012) (e.g., A attacks B), while *Gemini* is capable of providing interpretations about arbitrary numbers of entities, and about the game as a whole. Furthermore, the earlier approaches had no ability to ensure self-consistency, in that they would choose mechanics for pairwise entity relationships, but could not reason about the complex interactions between mechanics.

We note that while basic game generation is an interesting and complex problem in its own right, because *Gemini* can target the domain of generative "message games" its approach could also enable new kinds of interactive experiences. For example, earlier work has noted the possibilities of procedural narrative and game generation for pedagogical purposes (Samuel et al. 2017), where the experience can tailor itself to the interactions of the player. *Gemini* is currently being used to prototype such an experience, and this is the context in which the example games below were developed.

Gemini represents a novel contribution in 3 ways *(1)* it is the first generator to be able to target interpretations of games larger than entity-to-entity relationships, *(2)* it is the first game analysis system, *(3)* it is the first generator to use an analysis system to ensure consistency between the target interpretation and generated game.

Related Work

Generation of games and mechanics broadly fall under *top-down* or *bottom-up* approaches which our approach bridges. Earlier *top-down* approaches have enumerated a finite set of mechanics and then select those subject to constraints, perhaps looking to optimize some criterion. *Variations Forever* (Smith and Mateas 2010) was an early ASP based game generator that was able to generate games by using a fixed pool of mechanics (e.g., the red square will have movement like *Pac-Man*), but had very minimal constraints and mostly used ASP as a means of generating combinatorial sets. Like *Gemini*, *Game-O-Matic* (Treanor et al. 2012) is a system designed to generate games that support a reading defined by a user. However, *Game-O-Matic* only supports specifications of relationships between entities (e.g., dogs attack cats), and each relationship type is mapped to a finite pool of game mechanics (e.g., A attacks B could be represented by A shooting projectiles at B).

The most notable *bottom-up* approach is *Mechanics Miner* (Cook et al. 2013) which used reflection to allow the generator to determine the code-level parameters available to it and then generate mechanics by modifying entity properties using a set of atomic operations (e.g., modify a scalar by doubling or halving it). *Mechanics Miner* then used evolution to generate mechanics subject to playability constraints, rewarding mechanics that the player uses more than preset mechanics. *Mechanic Miner* has no notion about the real-world semantics of its mechanics, leading to mechanics that are awkward for humans to interpret (e.g., a mechanic that doubles the player's x-position has no real-world analogue). Furthermore, while *Mechanic Miner* prefers games that utilize generated mechanics, it always has its preset "platformer" mechanics to fall back upon.

Like *Gemini*, there are approaches that straddle the *bottom-up*/*top-down* distinction. Nielsen *et al.* (Nielsen et al. 2015) used evolutionary computation to generate games, either generated from scratch or mutated from human-authored games, with the goal of maximizing the distance between a "good" player (an MCTS approach) with a "bad" player (a player that does nothing). They used VGDL for their approach, which has some pre-specified mechanics (e.g., there are a finite number of movement models), but allows for some *bottom-up* creation (e.g., results of entity interaction). While interesting, their approach suffered from issues such as generating games that did not make use of player input, or games with nearly equal performance between the "best" player and a random player.

The closest work to ours is an ASP approach by Zook and Riedl (Zook and Riedl 2014) that generated mechanics from a finite pool of atomic actions (e.g., move the player n spaces to the right) such that certain constraints are met (e.g., the player can reach the exit of the level while staying alive). While their work is capable of generating evocative mechanics such as the player riding on the back of an enemy, these semantics are not reasoned about by their approach and only come from *post-hoc* human analysis. Another key difference between their approach and ours is that they are only able to generate mechanics that fit into a specific goal and rule scaffold, while *Gemini* generates those rules and goals. The largest distinction, however, is that *Gemini* produces playable games, instead of rules for a hypothetical game.

ANGELINA 3 and *4* (Cook, Colton, and Gow 2017b) also generate games that attempt to evoke specific affectual responses or have specific meanings. *ANGELINA 3* uses a fixed set of mechanics (Mario style platformer running and jumping) but generates games with specific theming (e.g., using images of Barack Obama and Hamid Karzai for a game about the war in Afghanistan) meant to offer commentary on a news article that *ANGELINA 3* itself selects from a newspaper. *ANGELINA 4* moves away from political commentary and instead accepts free text input to generate a game informed by a theme. Like version *3* it uses a fixed set of mechanics (e.g., first person navigation) and uses assets curated from online sources to match a specific theme. While *ANGELINA 4* makes games that it believes match a specific meaning, those interpretations often rely on inhuman leaps of logic that make the games unreadable to humans (e.g., given *One* as a theme it decided to make a game about a *Founder* where it used a *Ship* prop since *Ships* can *Founder*).

Earlier work by Martens et al. (Martens et al. 2016) looked at the game analysis problem in what they called a "reverse *Game-o-Matic*." This work performed a proceduralist reading (Treanor et al. 2011) of *The Free Culture Game*, an indie game about consumption and creation. They demonstrated a system capable of producing readings that were consistent with those of games scholars. However, their work did not tackle the game generation problem, and instead focused on analyzing previously authored games.

Gemini Methodology

We will now discuss *Gemini*—first *Cygnus*, the Domain Specific Language (DSL) we created, then the structure of *Gemini*, and the twin paths of analysis and generation.

Cygnus Overview

When first designing *Gemini* we designed 3 prototype games that would act as exemplars of games we wanted to be able to generate. After this exercise, we noticed that the existing DSL, the Video Game Description Language (VGDL), would not support the games we wished to generate. In part, this was due to VGDL targeting the most common control scheme of 80's arcade games – a joystick with 4-way movement and a single button, whereas we wanted mouse and touch control. However, in large part this was due to lacking general purpose rule construction – allowing only for rules that determine game termination, entity movement mechanics from a fixed pool, and entity-to-entity interactions. *Cygnus* offers refinements on these:

- **Mouse/Touch Controls** – Entities can react to being clicked on and the cursor is a special entity that mechanics can use (e.g., create an entity at the location of the cursor)

- **Fixed Movement Mechanics** – *Cygnus* allows for full control over entity movement and rotation, with atomic language constructs such as $\mathrm{look\_at}(A, B)$ "A changes rotation to be facing B" and $\mathrm{move}(A, S, D)$ "A moves in direction D with speed S" where D is either a global (`north,south,east,west`) or local (`forward,backward,left,right`) direction and S is a scalar value—from which any of the movement mechanics defined in VGDL can be constructed, as well as many not in VGDL
- **General Rule Construction** – Game rules are defined as a set of preconditions P and results R linked with an identifier O. Preconditions can consider entity interaction a la VGDL, but are not limited to that and can have arbitrary scalar comparisons (e.g., $R \geq S$ "Resource R is greater than or equal to scalar S") or user input checks (e.g., did the player press/hold/release button B)

The games describable by VGDL are a strict subset of the games describable by *Cygnus*, and *Cygnus* represents an advance in game description languages for games in the continuous control action domain. Our goal was to support a wide range of game constructs that could be compiled to JavaScript code using the Phaser game engine.[1]

Gemini Overview

While it was important to design a language capable of supporting the types of games we wished to generate, another key element of the design was being able to reason about the game at a fine granularity. *Cygnus* rules are represented by collections of AnsProlog (Baral 2003) facts of the form `precondition(P,O)` and `result(O,R)`. For example, a rule that, whenever a resource `r` is greater than or equal to zero, decreases `r` by 1 could be represented as:

```
precondition(
  ge(resource(r),scalar(0)), my_rule).
result(my_rule, decrease(r,scalar(1))).
```

These facts are then compiled into JavaScript for use with the Phaser game engine, e.g., as the code `if(r>0){r-=1;}`.

Since *Cygnus* is represented as set of AnsProlog facts, we can use standard Prolog-style rules that derive further facts, generating an answer set. We can refine the resulting generative space by providing a *design intent* as a set of AnsProlog constraints, ruling out cases we do not wish to appear. We will now describe the twin paths of *Gemini*, the analysis path, and the generation path.

Analysis Path Given the high-level goal of generating games with a specific reading, *Gemini* needs to be able to reason about games, so as to be able to ensure that generated games match the specified reading. To do this, we have encoded 215 rules that reason about game rules and derive further, higher-level knowledge about the game.

Some of these rules are simple facts about the game, such as "The player controls an entity if it moves as a result of the player's input:"

[1]`https://phaser.io/`

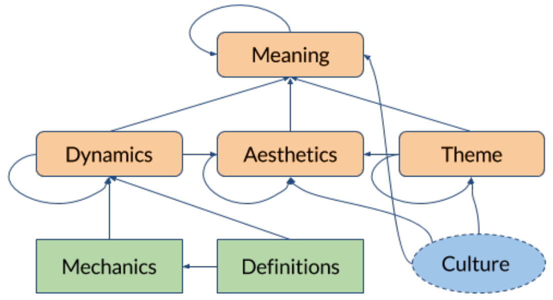


Figure 1: The flow of a proceduralist reading. *Green* nodes represent definitions (entities, resources, and other variables) and mechanics in the game, *Blue* nodes are the cultural context, and *Orange* nodes are the derived dynamics, aesthetics, and meaning. Arrows represent the directions information can flow in the process of a proceduralist reading (e.g., Mechanics and Definitions inform Dynamics which can then inform the Aesthetics or Meaning of a game).

```
player_controls(E) :-
  precondition(input(I),outcome(O)),
  result(outcome(O),move(E,_,_).
```

These rules may recursively refer to each other for more involved reasoning, such as "The computer controls an entity if the player doesn't control it and it's not static:"

```
computer_controls(E) :-
  entity(E),
  not player_controls(E),
  not static(E).
```

The reasoning chain for an example of `computer_controls(E)` can be seen in Figure 2. These rules encode part of a proceduralist reading process, since higher-level notions of meaning like antagonism and allyship refer to these notions of control and agency. The dynamics (how the game operates in situ), the aesthetics (how the game makes the player feel), and ultimately, how the game could be read by a player are facts derived from the definitions and mechanics of the game and culture-specific knowledge (see Figure 1). The analysis path is able to take in a game written in *Cygnus* and produce sets of consistent readings of the game.

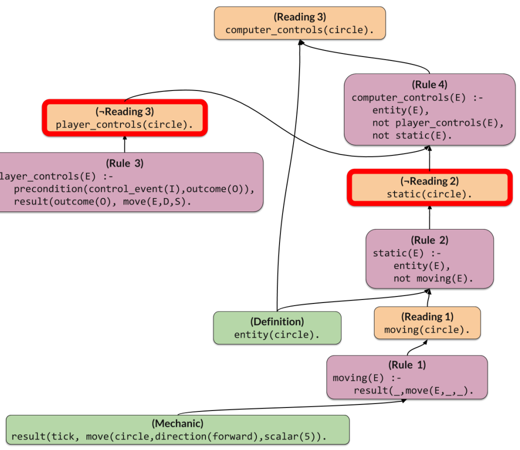
Generation Path Given the ability to analyze a set of game rules and mechanics to derive a reading, the declarative nature of ASP allows us to easily invert the process to take a fixed set of readings and generate a game that meets the desired reading. Choice rules are a construct in ASP that allow for the solver to non-deterministically choose facts such that they are consistent with all other facts and constraint rules. The simplest example in our code is:

```
{max_entity(M)} :-
  M = min_entities..max_entities.
```

which chooses an integer M to be the term of the predicate `max_entity`, such that

$\texttt{min\_entities} \leq M \leq \texttt{max\_entities}$

In general, the choice rules that make up the generation can be thought of under a few broad classes:



```
(Reading 3)
computer_controls(circle).

(Rule 4)
computer_controls(E) :-
    entity(E),
    not player_controls(E),
    not static(E).

(¬Reading 3)
player_controls(circle).

(Rule 3)
player_controls(E) :-
    precondition(control_event(I),outcome(O)),
    result(outcome(O), move(E,D,S).

(¬Reading 2)
static(circle).

(Rule 2)
static(E) :-
    entity(E),
    not moving(E).

(Definition)
entity(circle).

(Reading 1)
moving(circle).

(Rule 1)
moving(E) :-
    result(_,move(E,_,_).

(Mechanic)
result(tick, move(circle,direction(forward),scalar(5)).
```

Figure 2: A simple reasoning chain to determine whether a given entity is controlled by the computer. Nodes with a red border represent facts for which the negation is true. Building from a definition and a single mechanic, the chain invokes 4 rules (or their absence) to produce the final Dynamics reading of "The computer controls the *circle* entity," which is built on the earlier readings that the circle does in fact move (`not static`) and that the player does not control that entity (`not player_controls`).

- **Game Definitions** – General facts about the game, such as the number of entity types, the number of resources, the visuals associated with each entity type, or the number of timers found in the game
- **Rule Definitions** – As previously mentioned, a rule is a set of preconditions and results. *Gemini* is able to arbitrarily choose any combination of conditions, such as player input, resource comparisons, entity-to-entity interaction, or a timer elapsing, and pair them with any combination of actions, such as the creation or deletion of an entity, the modification of a resource, or the movement of an entity.
- **Mechanic Choices** – While *de novo* mechanic generation is a core aspect of *Gemini*, we have also hand encoded mechanics, such as an entity following the cursor or an entity chasing another entity.

The inclusion of hard-coded mechanics could feel at odds with our goal of game generation, but we found from playtesting that many of the generated games contained control schemes that, while able to be played, induced too much confusion and placed too much of a cognitive load on players to be useful for a generated experience (e.g., a game generated where clicking on an entity causes it to move up, pressing the mouse button while not clicking on it causes it to move right, and pressing the up key causes it to rotate to the cursor is technically controllable but is unpleasant for a human to play). However, while the base definitions of the mechanics are hard-coded, *Gemini* is able to modify any rule with the addition of preconditions and results, which leads to the generation of novel mechanics that are not hard-coded. In essence, the hard-coded rules act as a biasing of the generator towards games that have interpretable movement models with little other effect on the generative space.

There are 163 game design common-sense constraints placed on the generated games to ensure readability and playability. By readability, we mean ensuring that they are designed for human players. For instance, games become very difficult to parse if there are a large number of preconditions for a given rule (e.g., the player can only move right when a red circle and blue square are touching, resource R is above 3, resource S is between 2 and 7, they are holding the mouse button, and pressing the right arrow key simultaneously), so we limit the number of simultaneous preconditions to 3. Others perform static analysis on the rules to ensure that all outcomes are achievable. For instance, if there is a rule predicated on a resource being greater than a threshold, there must also be a way for that resource to increase. Furthermore, to eliminate dead lock, the way to increase that resource must not be predicated on the aforementioned rule. While most of these rules are always hard constraints, there are 19 that are able to be ignored if the user specifies that they can be ignored. For instance, a general rule is that if a resource is initialized, it must be used in at least one precondition and it must be modified by at least one rule. However, points are a common occurrence in games and they often are modified by the game, but no rules are conditioned on them, so if the user wants a point system in their game they can disable the constraint.

In addition to common-sense design rules, *Gemini* can also accept a design intent, the readings and specifications with which the user wants the generated game to conform. These can be as simple as specifying the number of different types of entities, or more complex specifications such as:

```
:- not reading(hand_eye_coordination).
:- not reading(maintain,resource(r(1))).
label(resource(r(1)),concentration) :-
  reading(good,resource(r(1))).
label(resource(r(1)),stress) :-
  reading(bad,resource(r(1))).
```

or more simply, "The game should be read as a hand eye coordination game that requires the player to maintain resource $r(1)$. If $r(1)$ is determined to be good, it should be labeled as concentration, and if $r(1)$ is determined to be bad, it should be labeled as stress." (We note that in AnsProlog headless rules are read as constraints that forbid the body from being true, hence the `not` in the first two rules.)

After generating the rules and definitions of the game, it must be made playable. The second phase takes in the *Cygnus* game definition and compiles it to JavaScript for use in with the Phaser game engine. The compiler operates by converting the input into the Rensa (Harmon 2017) format, a Prolog-like relation specification. The compiler operates by constructing Rensa relations from *Cygnus* code, e.g., `resource(value1)` $\rightarrow \langle$ `value1` *instance_of* `resource` $\rangle$. The Rensa relations are then converted to JavaScript code in a context sensitive manner, owing to the intricacies of the Phaser engine. For example, code resulting from clicking on an entity is handled in a callback function, as is code resulting from entity-to-entity collision, but when a result is predicated on both clicking and entity-to-entity collision the callbacks must be nested such that the collision checking code is called from within the click callback.

It is very important that *Gemini* make things playable, for a number of reasons. Perhaps the most important reason from a non-research perspective is that *Gemini* was created in service to a larger creative project so generating just rules or VGDL (to then be interpreted by a cumbersome framework) was not a possibility. However, the most important reason is that the debugging process for rule writing requires playability. A game might seem plausible when just reading the rules, but only when played is it shown to be untenable. E.g., a game where the player controls an entity chasing another a fleeing entity seems reasonable and challenging; however, in the absence of any other rules, this will inevitably lead to the chased entities getting stuck in the corners of the screen. Had we only read the rules, we would have missed this interaction rising from the dynamics.

Examples and Discussion

We will now describe example games that have been created in support of a larger creative work. *Gemini* is only part of this experience, as the generated games sit side-by-side with a choice-based interactive narrative, with both the narrative and the game side communicating player actions (Samuel et al. 2017). The graphics of the games are abstract with the meaning being conveyed to the player via color, shape, and most importantly the mechanics at play. The design intents of these games come from a team that was tasked with creating prototype games, games that the team would have been happy with were they generated by *Gemini* for this larger experience. We note that none of the prototype games would have been expressible in VGDL and that none of the prior systems would have been capable of satisfying any of the design intents. We now describe the design intents given to *Gemini* and describe a few of the generated games that result. Finally, we discuss the ways in which *Game-o-Matic*, the only prior game generator that generated games with specific meaning, would have been incapable of handling these design intents and the ways in which all of the generated games outstrip the expressive capabilities of VGDL.

The Dinner Game One of the games was dubbed "the dinner game," a game that is supposed to be mimetic of eating dinner with friends. In the prototype, food would appear at either the left or right side of the screen—the player's goal was to take (click on) the food and then pass it (click again) to the other side. The stated intent of the game was to represent sharing food at a table with friends.

All of the generated games share the representation of a communal table of food, represented as red circles, and the player must pass food to their friends, represented as blue circles. The design intent of the game is simply:

1. The game should involve sharing of one entity type, amongst another entity type
2. The game should involve maintaining a resource
3. There should be no more than 3 sharers, and those entities should remain constant

From that intent, we get games such as:

Dinner 1

1. Satiation is constantly decreasing
2. If satiation drops below 0, the player receives a message asking them to pass the food
3. The friends consume the food when they overlap, which increases satiation, but only if hunger is high enough
4. Food spawns periodically
5. The friends orbit around the food, but do not move toward it
6. The cursor repels food away when the mouse button is pressed
7. Hunger increases constantly throughout time
8. Hunger decreases when a friend consumes the food

Dinner 2

1. Satiation is constantly decreasing
2. If satiation drops below 0, the player receives a message asking them to pass the food
3. The friends consume the food when they overlap, which increases satiation
4. When one piece of food is consumed, another spawns
5. The friends wander, changing directions randomly
6. The food moves forwards, changing directions to face towards a random friend when the mouse button is pressed

Some facets of these games remain constant, a consequence of the design intent. For instance, maintenance of a resource is recognized as the resource either increasing or decreasing over time and the player must take action to keep the resource from passing a threshold in the appropriate direction. Similarly, sharing is recognized as when entities of a type consume an entity of a different type, the result of which helps the player achieve a goal of the game. These result in the first 3 rules being identical for both games.

However, we see that the control schemes for each game are completely different, and that the *Dinner 1* game decides to use the secondary resource "hunger," not mentioned in the intent, as a way of regulating the players' actions.

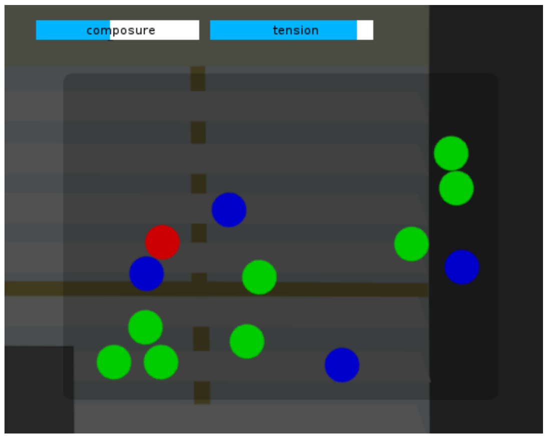


Figure 3: Screenshot of a generated game. The player controls the red circle by clicking and dragging it. The blue and green circles randomly wander around the screen, wrapping around if they collide with an edge. While a blue circle overlaps the red circle, composure increases, but if a green circle collides with the red circle, composure takes a drastic drop. The number of green circles increases as tension increases.

The Supervisor Game The dinner game examples were for design intents derived from the earlier prototyping effort. We have also experimented with new design intents for games we have not previously hand-designed. This intent is in support of a game that is supposed to represent a difficult conversation with the player's supervisor. The player's supervisor is critiquing the player, and the player has to recognize the helpful advice and not be soured by the critique. This design intent is conveyed as:

1. There are two entity types, and these entities must share the same movement profile. Conversely, there are two rules, each dealing with one of the entities and the results of these rules should be opposed to each other. The one that helps the player is labeled as "helpful criticism," the other as "hurtful criticism"
2. The goal is to maintain a resource, labeled as "composure," and that resource must be seen as good
3. There is another resource, labeled as "tension," that is linked to difficulty

which results in games such as:

Supervisor 1

1. The player controls a red circle by clicking and dragging
2. While the red circle and a blue circle overlap, composure is increased
3. When a red circle and green circle overlap, the green circle is deleted and composure decreases
4. The green circles constantly spawn, doing so at a higher rate as tension increases
5. Both green and blue circles wander around the screen, changing directions randomly
6. When anything hits the edge of the screen, it wraps to the opposite side

Supervisor 2

1. The red circle is constantly repelled from the cursor
2. When a red circle and a blue circle overlap, the blue circle is deleted, composure increases, and another blue circle is spawned
3. When a red circle and green circle overlap, the green circle is deleted, composure decreases, and a new green circle is spawned
4. Both green and blue circles orbit the red circle
5. The amount that composure decreases when the red circle and green circle collide increases as tension increases

While we do not see identical rules, we certainly see the commonalities that the design intent forces, namely the identical movement patterns and the opposite results for when the red circle interacts with either a blue or green circle. We see two completely different ways that "Tension is related to difficulty" is reified, in that it increases the amount of harmful things for the first game and increases the amount of harm for the second.

Discussion

We now discuss how *Gemini* enables these scenarios in a way that would not have been possible with earlier game generation systems, focusing on *Game-o-Matic* as it is the only earlier system able to target specific interpretations.

The Dinner Game *Game-o-Matic* would be capable of supporting the concept of "sharing" as one of its micro-rhetorics (intent 1), but the other design intents are impossible for it. *Game-o-Matic* only handles entity → entity relationships, so broad game level rules (e.g., the game should involve maintaining a resource – intent 2) and entity specifications (e.g., no more than 3 sharers that should remain constant – intent 3) are incapable of being represented.

The Supervisor Game None of the intents specified for the supervisor game would be possible to be represented under *Game-o-Matic*. While *Game-o-Matic* can represent relationships between entities, and it is possible to have two entities that have all of the same relationships (e.g., *A attacks C*, and *B attacks C*), there is no way to place a constraint that those relationships should be reified in the same way (e.g., *Game-o-Matic* is as likely to make it so *A attacks C* by shooting at it, while *B attacks C* by chasing it than it is to choose the same mechanics for both) making it impossible for it to meet intent 1. As in the dinner game, game level specifications are impossible (intent 2), as are links between resources and game concepts (e.g., a resource being linked to difficulty – intent 3).

Other Systems We note that all of the produced games are impossible to represent in VGDL, as they all contain components not capable of being expressed.

Non-Interaction Rules Rules that are not a part of interactions – **Dinner 1** $1, 7$, **Dinner 2** 1

Mouse Controls Rules that utilize the mouse – **Dinner 1** 6, **Supervisor 1** 1, **Supervisor 2** 1

State Changes Based on Resources Rules that use resource values for something other than threshold checks – **Supervisor 1** 4, **Supervisor 2** 5

Movement Models Movement models that do not exist in VGDL – **Dinner 1** $5, 6$, **Dinner 2** 6, **Supervisor 1** $1, 6$, **Supervisor 2** 4

Not only are existing VGDL generators incapable of targeting the generation of a game with a specified intent, the expressive scope of VGDL would prevent a system from generating these games if they had the aforementioned capability. Similarly, the semantics of the work by Zook and Riedl (Zook and Riedl 2014) do not cover things such as stochasticity or player input.

Conclusion

We have presented *Gemini*, a first of its kind generator that both can statically analyze a game, to derive play aesthetics and possible interpretations, and can generate games targeting a specific interpretation. *Gemini* is built on a domain specific language, *Cygnus*, which enables mechanics and entire game forms impossible with VGDL, the most popular game DSL. As *Cygnus* is embedded in AnsProlog, we use the Answer Set Programming solver *Clingo* (Gebser et al. 2010), which affords the twin analysis and generation paths of *Gemini*. We have shown representative games (not capable of being represented by existing game DSLs) produced to target specific interpretations (not capable of being handled by previous systems).

References

Baral, C. 2003. *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press.

Cook, M.; Colton, S.; Raad, A.; and Gow, J. 2013. Mechanic miner: Reflection-driven game mechanic discovery and level design. In *European Conference on the Applications of Evolutionary Computation*, 284–293. Springer.

Cook, M.; Colton, S.; and Gow, J. 2017a. The angelina videogame design system, part i. *IEEE Transactions on Computational Intelligence and AI in Games*.

Cook, M.; Colton, S.; and Gow, J. 2017b. The angelina videogame design system, part ii. *IEEE Transactions on Computational Intelligence and AI in Games*.

Gebser, M.; Kaminski, R.; Kaufmann, B.; Ostrowski, M.; Schaub, T.; and Thiele, S. 2010. gringo, clasp, clingo, and iclingo.

Harmon, S. 2017. Narrative encoding for computational reasoning and adaptation.

Martens, C.; Summerville, A.; Mateas, M.; Osborn, J.; Harmon, S.; Wardrip-Fruin, N.; and Jhala, A. 2016. Proceduralist readings, procedurally.

Nelson, M. J., and Mateas, M. 2007. Towards automated game design. In *Congress of the Italian Association for Artificial Intelligence*, 626–637. Springer.

Nielsen, T. S.; Barros, G. A.; Togelius, J.; and Nelson, M. J. 2015. Towards generating arcade game rules with vgdl. In *Computational Intelligence and Games (CIG), 2015 IEEE Conference on*, 185–192. IEEE.

Samuel, B.; Garbe, J.; Summerville, A.; Denner, J.; Harmon, S.; Lepore, G.; Martens, C.; Wardrip-Fruin, N.; and Mateas, M. 2017. Leveraging procedural narrative and gameplay to address controversial topics. *2017 Workshop on Computational Creativity and Social Justice (CCSJ 2017).*

Smith, A. M., and Mateas, M. 2010. Variations forever: Flexibly generating rulesets from a sculptable design space of mini-games. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, 273–280. IEEE.

Togelius, J., and Schmidhuber, J. 2008. An experiment in automatic game design. In *Computational Intelligence and Games, 2008. CIG'08. IEEE Symposium On*, 111–118. IEEE.

Treanor, M.; Schweizer, B.; Bogost, I.; and Mateas, M. 2011. Proceduralist readings: How to find meaning in games with graphical logics. In *Proceedings of the 6th International Conference on Foundations of Digital Games*, 115–122. ACM.

Treanor, M.; Blackford, B.; Mateas, M.; and Bogost, I. 2012. Game-o-matic: Generating videogames that represent ideas. In *Second PCG workshop*.

Treanor, M. 2013. *Investigating procedural expression and interpretation in videogames*. Ph.D. Dissertation, University of California, Santa Cruz.

Worth, D. 1978. Beneath Apple Manor.

Zook, A., and Riedl, M. O. 2014. Automatic game design via mechanic generation.