Variations Forever: Flexibly Generating Rulesets from a Sculptable Design Space of Mini-Games

Adam M. Smith and Michael Mateas

***Abstract*—*Variations Forever* is a novel game in which the player explores a vast design space of mini-games. In this paper, we present the procedural content generation research which makes the automatic generation of suitable game rulesets possible. Our generator, operating in the domain of code-like game content exploits answer-set programming as a means to declaratively represent a generative space as distinct from the domain-independent solvers which we use to enumerate it. Our generative spaces are powerfully sculptable using concise, declarative rules, allowing us to embed significant design knowledge into our ruleset generator as an important step towards a more serious automation of whole game design process.**

I. INTRODUCTION

Automatic generators exist for many game content domains: 2D textures, 3D models, music, level maps, story segments, ships and weapons, items and quests, character attributes, etc. In terms of a distinction between code and data, these kinds of content feel like data and they are interpreted by the fixed code in game engines. However, some kinds of content such as location-based triggers on a map, behavior trees, or the contents of game scripts blur the line between game data and game code (between representational and behavioral components). The field of game research known as procedural content generation (PCG) can be expanded to include richer aspects of game design if the "content" that is generated includes the kind of conditionally-executing logic that we would otherwise call a game's mechanics.

While PCG is often motivated as a means to reduce development effort/costs for game content [1], it can also provide access to richer, and more personalized, play experiences than could be reasonably hand-authored by human designers. The rich worlds of *Dwarf Fortress*[1] include procedurally generated multi-level landscapes and thousands of years of history. Meanwhile, the player-designed creatures of *Spore* (Maxis 2008) are enhanced with unique, procedurally generated skin details and body animations. Further, these personalized creatures are used to populate a vast, procedurally generated galaxy reminiscent of the seminal PCG work seen in *Elite* (Acornsoft 1984). These games had impressive generated data, but employed hand-authored mechanics.

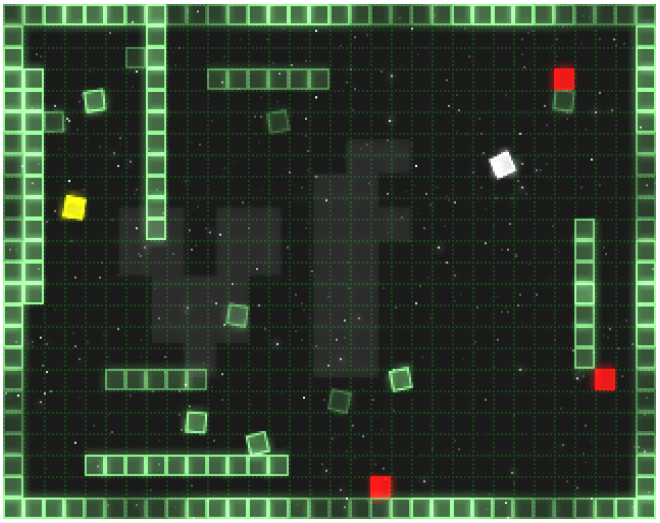

Fig. 1. **Screenshot of gameplay for a generated mini-game in the *Variations Forever* prototype.** The player controls the white character using an Asteroids-inspired movement model, trying to touch all red characters which move via a Pac-Man-inspired movement model. The encircling walls, random-walls and random-blocks algorithms have generated dangerous obstacles which can harm the player's square. This particular game's rules also define the stars and grid backdrop details as well as a third kind of character (yellow) which drifts on its own.

Though the automatic generation of game mechanics is an important and underexplored component of *automated game design*, it is important not to collapse the part with the whole (which has been done in the literature [2]). Nelson & Mateas [3] proposed a factoring of game design into four domains: abstract game mechanics (abstract game state and how this state evolves over time), concrete game representation (the audio-visual representation of the abstract game state), thematic content (real-world references), and control mapping (relation between physical player input and abstract game state). While we can imagine a procedural generator for the content of any of these domains, even this would miss out on an opportunity to illuminate several commonly accepted processes in game designs that cross-cut these domains. Conceptualization, prototyping, playtesting and tuning are essential parts of game design [4]; there is no compelling reason to think they should not be addressed in a nuanced automation of game design. However, addressing only a

The authors are with the Expressive Intelligence Studio at University of California, Santa Cruz. Emails: {amsmith,michaelm}@soe.ucsc.edu

[1] http://www.bay12games.com/dwarves/

piece of the whole game design process, the research presented in this paper focuses on flexibly generating a variety of abstract mechanics, utilizing hand-authored components for concrete representations, thematic content, and control mapping.

Variations Forever (VF) is the name of both a work-in-progress game and the research project of developing the technology necessary to implement it. In the remainder of this section we will distinguish these two projects.

A. Variations Forever as a game project

VF, the game, aims to provide players with the experience of exploring a generative space of games as an in-game activity. The premise, visual style, and themes employed by the game are inspired by a set of recent, independent games.

ROM CHECK FAIL[2] is a glitch-laden arcade game in which the player's avatar, movement mechanics, level design, theme music, and enemy mechanics shift at regular intervals. In the unique, emergent meta-game, the overarching goal is simply to survive the onslaught of new mini-games. In VF, however, meta-game will involve unlocking new mini-game design elements which reshape the space of mini-games.

Warning Forever[3], *Battleships Forever*[4], and *Captain Forever*[5] each have a space combat setting with glowing vector art. Beyond aesthetics, they share the theme of recombining elementary parts in novel ways in the player's major choices (assemble a spacecraft from standard modules such as girders, thrusters, and weapons). In VF, the recombinant nature will shift from ship design to ruleset design.

This paper presents a prototype of VF (depicted in Figure 1) in which we have realized a large space of varied mini-games. This prototype does not yet include player-control over the design space; however, we will show how our approach supports such functionality.

B. Variations Forever as a research project

The goal of VF, the research project, is to create a means to automatically explore a generative space of game rulesets that supports both the variety of mini-games we desire and the hooks needed to place the exploration into players' hands. Our emphasis is on flexibility of generation, and we leave evaluation of game quality for future research.

We have adopted a symbolic approach to representing games because we believe that breaking free of the parameter-vector paradigm pervasive in PCG will be required to address the larger automation of game design. Reasoning through the intentional creation of prototypes, the identification of useful mechanics and generation of hypotheses to validate in playtesting requires a representation which resonates with the symbolic, modular, non-parametric medium used to implement every videogame: code.

[2] http://db.tigsource.com/games/rom-check-fail
[3] http://db.tigsource.com/games/warning-forever
[4] http://db.tigsource.com/games/battleships-forever
[5] http://db.tigsource.com/games/captain-forever

In this paper we describe a flexible approach to game ruleset generation that should also be of interest to those PCG researchers working in traditional content domains such as level or map generation, etc. While we will use VF as a running example, the schema we provide for creating generators is not inherently tied to the ruleset generation domain.

Our contribution is a content generation approach based on constraint logic programming which allows the declarative specification of design spaces and uses domain-independent solvers to sample these spaces. Our application to ruleset generation demonstrates the representational flexibility and ease of incremental modification that comes with the use of a non-parameterized representation. Such flexibility and modifiability are critical to integrating the generation of rulesets into larger-scale game design automation efforts.

II. Related Work

Automated game design has been studied from a number of different angles. An important bit of vocabulary we can use to reconcile these efforts is "generate and test" a popular discussion topic on a PCG community mailing list[6]. Generation tells us how artifacts of interest come into existence, and testing tells us how these artifacts are separated from their less-interesting neighbors in some space (generally involving both evaluation and filtering).

A recent example of generating and testing simple game designs is seen in the work by Togelius [5] in which a space of Pac-Man-like games is automatically explored using evolutionary computation. Game variants, represented as fixed-length vectors of integer parameters which encode quantitative properties of a game such as its time and score limit as well as basic qualitative data such as which movement logic or collision effects are used by the various "things" in the world. The mechanics of the games generated by this system are the result of combining the parameter vector with some fixed rules defining the meaning of each parameter. The mechanics of the mini-games we consider in VF are directly inspired by this work.

Togelius' parameter vectors are using an operationalization of Koster's theory of fun [6] based on reinforcement learning. The system illustrates that while games may be generated by some simple, syntactic method, they must be played to understand their semantics and to assign them value. While acknowledging the depth of automatic playtesting as both a computer-science and game-design problem, VF focuses on bringing flexibility to generation. Where Togelius' games all involved exactly four colors of "things", VF's generator is capable of choosing an arbitrary number of characters types and populating the appropriately sized collision effects table to describe their mechanics (such scalability in terms of set cardinality is awkward for fixed-length parameter vectors).

Hom & Marks' project in generating balanced board

[6] http://groups.google.com/proceduralcontent

games also took an evolutionary approach, opting for three-way crossover as the means of generating games represented by a triple of board type, piece type, and victory condition [2]. Testing, in this project, was done using general game playing software to look at the relative win rates for the first vs. second players.

In addition to the main triple, the representation also allowed optional rule modification tags which could be appended to a game ruleset. While these tags could have been folded into additional Boolean values in the genetic description, the authors decided to treat them specially to simplify the crossover logic of their generator. This also points to awkwardness in using parameter-vector techniques to represent rulesets. Incidentally, the general game playing tool they used already expected a code-like representation as input.

Automation of ruleset generation has also been addressed without including a distinct test component. The METAGAME system [7] and the EGGG system [8] are both capable of generating games as complex as Chess without any testing. However, this ability comes at the cost of an immense knowledge engineering effort to embed as much intuition about the game design space into the generator as possible. METAGAME's author refers to the generator as "long and complicated and full of *special cases*". These systems illustrate that generators can actually contain and represent large amounts of domain knowledge. It is desirable to have a generator that is improvable, that is, can be easily augmented with new knowledge which helps it avoid uninteresting or problematic regions in the design space. Our generation approach indeed aims to ease the process of adding new knowledge to a generator (particularly that which focuses the generative space).

In a filtering-heavy approach to designing games, Nelson & Mateas use a common-sense knowledgebase to filter out only those combinations of mechanics and representational art that "make sense" from a larger generative space [3]. Because the mechanics used in their WarioWare-style games are so simple, detailed playtesting is not required for evaluation. Rather a common-sense knowledgebase is effective in filtering a game where a duck avoids a bullet shot by a gun (a reasonable premise) from a games where a person fills a piano with ducks (less reasonable), both of which are in the latent generative space tested by this system. For a given target concept, such as "duck", the system might produce any of a small space of games which both make sense and include the target concept. This project illustrates that filtering one generative space to make another is an important technique in design automation.

Finally, though it only consumes and does not produce rulesets, the dual human-and-machine playtesting supported by our BIPED system [9] is relevant. This playtesting tool reads in rulesets represented as logic programs and produces gameplay traces in a similar logical representation that are derived either from human players or from logic programming tools which can solve for edge and limit cases in a game's design. This project shows that symbolic, logical representations for game rulesets can be comfortably used by both humans and machines and they can serve as an effective interchange format for the playtesting stage of design.

III. INTRODUCTION TO ANSWER SET PROGRAMMING

The essence of our approach here is a separation of the generative space from the procedure that enumerates it. We use an answer set program to specify the generative space. Answer sets from this program can be fed to a traditional game engine, resulting in fully playable games. The definition of the generative space is designed to be concise and to support easy modification of the space.

Answer set programming (ASP) is a form of constraint logic programming [10]. ASP is often used to implement abductive reasoning, a logical foundation for traditional AI applications such as automated planning, fault diagnosis, and natural language understanding [11]. Common to all of these applications is the process of generating a logically-described artifact (a plan, a fault, a statement, etc.) from which desired observations follow deductively. Outside of abductive reasoning, ASP can be used to generate logically-described artifacts which meet internal-consistency constraints. It is precisely this generative facility (synthesis) with the integration of deductive reasoning (analysis) which makes ASP attractive to us in automating game design.

ASP extends the predicates used in traditional (deductive) logic programming with two special constructs: choice rules and integrity constraints. Choice rules give a program license to assume a set of logical statements if needed. To give an example, "`{rain, sprinkler}.`" says "it may have rained, the sprinkler may have been on, or both". Appending this to a background theory such as "`wet :- rain. wet :- sprinkler. dry :- not wet.`" (which says "it is wet outside if it rained or if the sprinkler was on, otherwise it was dry") allows an ASP solver to enumerate different logical worlds (called answer sets) consistent with these rules. There are *four* possible worlds: "`dry.`", "`rained, wet.`", "`sprinkler, wet.`", and "`rained, sprinkler, wet.`". In this example we have already defined a tiny generative space and shown the elements that a solver would extract from it.

Integrity constraints balance the generative nature of choice rules; they allow the programmer to express certain conditions which should be considered unreasonable to hold. Suppose we are interested in explaining how it got to be wet outside, and we further know that a mechanism in our sprinklers keeps them from activating in the rain. We can expand our example above with two integrity constraints which resemble logical rules with a missing head: "`:- not wet.`" and "`:- rained, sprinker.`". Intuitively they say "don't show me logical worlds where any of these conditions hold". Running our ASP solver on the program now results in only two answer sets, both containing the `wet` fact and one of `rained` or `sprinkler`. Appending additional knowledge to our definition has scoped the generative space down to a

smaller one in which our observations hold.

The toy problem of rain and sprinklers is traditionally given as an example of abductive reasoning, but here we aim to emphasize how choice rules and integrity constraints in ASP give us language-level support for the conceptual processes of "generate" and the filtering aspect of "test". True to our intuitions for formal logic, and unlike Prolog, constructs in ASPs (both the predicates and the conjuncts in their bodies) can be freely re-ordered, meaning that the programmer need not worry about backtracking or other issues of how generation and testing are interleaved at the execution level.

IV. Representing Rulesets in Logical Terms

To put ASP to work in the ruleset generation domain, we need to somehow represent elements of game rulesets in the heads of ASP's choice rules and encode the logical conditions which ensure the generated rulesets are valid into the bodies of these rules. This implies a representation of ruleset elements as logical terms.

Though logical terms are a standard knowledge representation format in AI, we review them here because they have not been used in published PCG work (with one intriguing, unpublished exception[7]). A logical term is either an atom or a compound term. Atoms are symbols, numerical constants, or logical variables. Compound terms combine a symbol called a functor with a sequence of logical terms as arguments, as in `afraid_of(6,7)`.

An example of a logical term encoding an element of a game script is the following: `scripted_event( spawn(boss_creature, temple), 120)`. This term, if it asserted in an answer set, might mean "a boss-class creature should be spawned in the temple after two minutes of play". The meaning given to `scripted_event` is given by the code that consumes it, which might be other rules in the ASP or the game engines which deliver this content to the player.

The rules for the kind of games we are considering for VF contain various types of information: collections of objects that participate in the game and their properties, policies for handling events that arise during play, conditions under which we can consider the game to end in victory or defeat, and, additionally, miscellaneous procedures and configuration details which we can use to add to the variety of play experiences. Lists, variable sized look-up tables, and nested expressions are all difficult to represent with fixed-length parameter vectors.

Each of the elements we would like to include in rulesets has a straightforward representation in logical terms. Lists, such as a list of valid move types, are represented by a pattern of terms that may be instantiated several times: "`move(rock). move(paper). move(scissors).`" (a numerical argument may added to represent a strict ordering if needed). Tables, such as a mapping from event to a handling character's response action with a performance modifier, are represented by simply asserting the presence of each tuple of the data the table contains: "`on(poke, giggle, quietly). on(jab, yelp, loudly). on(stab, die, slowly).`". Code-like nested expressions are also natural: "`when(equal(health, 0), go_state(defeat)).`".

While we could use a simple, context-free grammar to syntactically generate masses of such terms, building our generator as an answer set program gives us the means to powerfully sculpt the space of generated rulesets using the same language used to define it. For example, when generating terms of the form "`on(poke, giggle, Adverb)`" we can require that the adverb come from a table of modifiers compatible with the giggle action or forbid the use of certain adverbs in conjunction with the poke event by consulting a blacklist of known problem cases. Further, such a table or blacklist might itself be the output being simultaneously produced by another part of the same generator. The ability to specify defaults and override them with many levels of specialization, important for flexible modeling, comes from the non-monotonic reasoning used in ASP solvers.

V. VF's Generative Space

Having introduced basic answer set programming and a representation of game rulesets in its terms, we now describe how the concrete elements of rulesets for VF are generated and how these elements influence one another. The generative space of the VF game prototype is meant to exercise the expressiveness of the generator's logical formulation and does not yet represent the complete set elements in the other *Forever* games we intend to reference. Figure 1 provides a visual guide for one element of VF's generative space.

A basic element used by all mini-games in VF is the play space. It is always rectangular, but has a numerical grid resolution parameter which is used by certain character movement models and obstacle-placement policies. Specifying the simple selection of a numerical parameter looks like this:

```
resolution_factor(2;3;4;6;8;12;16).

1 {space_resolution(4*F)
   :resolution_factor(F)} 1.
```

This snippet asserts that it is true that several numerical symbols are valid resolution factors, and that the *exactly one* ground clause of the `space_resolution` predicate should be emitted in answer sets. The clause in curly braces is a choice rule which gives permission to emit terms of a certain form (where the *F* variable is bound by the `resolution_factor` in this case).

Our mini-game play space has an overall topology which is either toroidal (like Pac-Man), spherical (strange but nonetheless distinct), or flat (resulting in "falling off" the edges of the world) if not otherwise specified. The generator outputs between zero and one instances of `space_topology` predicate (as dictated by numerical bounds on the choice rule) using a scheme similar to the above with symbols to name the various topologies instead of numbers.

Related to the space, but not an element of the game me-

[7]The *Warzone Map Tools* project uses ASP to generate strategically-optimized maps for an RTS game. http://warzone2100.org.uk/

chanics, the game may utilize (or not) any of two background layer display algorithms (twinkling stars or dotted grid lines). The generator code for this aspect introduces dependence between a generated element and a flag that might be toggled via player exploration in future VF prototypes:

```
tech(backgrounds).

{background(L) :background_layer(L)} :-
    tech(backgrounds).
```

The play space is primarily populated by characters, identified by color. The generator internally selects an active subset of colors from a larger list, and whether a color is active or not is used as a logical precondition for the rest of the character-related generator rules.

Every active character color is assigned a unique movement model (determining their response to keyboard input and, eventually, autonomous behaviors). The VF prototype includes Asteroids, Pac-Man, and Rogue inspired movement models. The generator code to support this combines quantification over multiple variables to produce a one-to-one mapping specific to the active characters:

```
1 {agent_movement(C,M)
  :movement_model(M)} 1 :-
          active_color(C), color(C).
```

In additional to a movement model, characters have another required property called their spawn model which dictates whether exactly one of them should spawn at the start of the game or a larger, random number. The generator code has identical structure to that of the movement model.

Much more interesting is the character-character collision effects table. This table, which describes only active characters, produces `agent_collide_effect`, a predicate which is not only used by the game engine during mini-game execution, but also by the generator itself as we will describe later. The table describes which collision resolution behavior should be applied to the character of the first color if it hits another of the second color. There are `kill` and `bounce` options with a default of simply passing through on collision. In future versions of VF, the set of collision resolution behaviors that are considered will be conditioned on player exploration as well.

Beyond the basic space and characters, if the generator has the `obstacles` exploration flag enabled, the game will consider any combination of three obstacle placement patterns: an encircling wall, stick-like scattered walls, or isolated blocks with slight rotations (all three are active in Figure 1). The selection of these algorithms is described with the same schema used to select background art layers. If obstacles are enabled, an optional collision resolution behavior is selected for each character and encoded as the `obstacle_collide_effect` predicate (obstacles themselves cannot be "killed"). In this case, not just the size but the very existence of a table in the ruleset is conditioned on other generated outputs (the active set of colors).

In order to make games in this space playable, we need to assign the player control of one of the characters. This assignment is based on color; if the player controls a character with the "many" spawn model, then they will only control one such character and the rest will perform their default behavior.

At this point, games include a player who can fly a character around a variously configured world, bumping into other characters and obstacles to trigger effects out of tables, but there is still no goal to our mini-games. The final element of the mini-game description gives it one. The `goal` predicate must have a single instance in each game design to enable victory-condition checking. Its form may either be "`goal(kill_all(Color))`" which monitors for when all characters of a given color are killed or "`goal(escape)`" which monitors for when the player character reaches the world boundary (which only exists in the flat space topology).

This final output illustrates a clear representational flexibility that our symbolic representation has over fixed-length parameter vector representations: some of our game goals are parameterized by active character colors, while others such as `escape` are not. There is no penalty for mixed structures such as this. The `escape` goal, in particular, is additionally forbidden in games utilizing the encircling wall obstacle generator (from which is impossible to escape). This is an example of capturing special-case knowledge in the generator extracted from experience playtesting broken games.

The complete generated ruleset for the "kill all the red guys" game shown in Figure 1 is represented by these logical statements produced by the generator:

```
space_resolution(32,24).

space_topology(spherical).

background(grids; stars).

active_agent(red; yellow; white; cyan).

agent_movement(red,asteroids; white,asteroids;
               yellow,roguelike; cyan,pacman).

agent_population(red,many; white,singleton;
                 yellow,singleton; cyan,many).

agent_collide_effect(red,white,kill;
                     cyan,yellow,kill).

player_agent(white).

obstacle_distribution(enclosure; random_walls;
                      random_blocks).

obstacle_collide_effect(red,kill; white,kill).

goal(kill all(red)).
```

VI. Zooming in on Games of Interest

While the exclusion of known problematic interactions between mechanics is something that could be encoded directly into the preconditions in the body of the choice rules used in the generator, there are many occasions in which we would like to temporarily scope down our generative space without disturbing any existing logical formulae. The mechanism of integrity constraints in ASP provides exactly this

functionality.

We can use integrity constraints to zoom-in on a subspace of games in which we are interested via several methods, but the simplest is to simply require that certain ruleset elements be present in all answer sets we see. Recalling the original rain/sprinkler example, we can simply append an integrity constraint rejecting the absence of the required configuration. If we wanted to tweak the implementation of the Asteroids motion model in VFs game engine, we might add this collection of constraints to always give ourselves control over a red character in a primitive field of asteroids to navigate, regardless of other mechanics:

```
:- not player_agent(red).
:- not character_movement(red, asteroids).
:- not use_obstacles(encircling).
```

Another use of integrity constraints to sculpt the generative space is to reject co-occurrence of mechanics known to interact poorly. The special-case knowledge for the `escape` game goal is encoded with the single integrity constraint "`:- goal(escape), obstacle_distribution(enclosure).`"

Integrity constraints need not only operate on the same elements that are exported by the generator, they may involve complex deduction. The following snippet of code (slightly condensed) zooms in on rulesets in which the game is reasonably winnable by indirectly pushing characters into each other to achieve the stated goal:

```
pushes(A,B) :-
  on_collide(A,B,bounce),
  on_collide(B,A,bounce).

kills(A,B) :- on_collide(A,B,kill).

indirectly_pushes(A,B) :- pushes(A,B).

indirectly_pushes(A,C) :- pushes(A,B),
  indirectly_pushes(B,C).

winnable_via(indirect_push_kill(A,C)) :-
  indirectly_pushes(A,B), kills(B,C).

compute {
  player_agent(A), goal(kill_all(B)),
  winnable_via(indirect_push_kill(A,B)) }.
```

In the above example, we have shown an elementary attempt at *engineering emergent gameplay*. As new mechanics are added to VF's design space and the `winnable_via` predicate is augmented with a simple complexity metric, it becomes possible to write single-line integrity constraints which translate to statements akin to "only show me rulesets for games in which a reasonable plan for victory involves an indirect chain of at least 5 steps utilizing at least 3 different low-level interaction types". Games complex enough to satisfy this constraint would likely come from a space which also includes many broken games. Fortunately, additional integrity constraints may easily be added to carve away such failure cases as they arise in playtesting. It is this use of additional deductive rules to capture complex relationships between low-level mechanics which demonstrates the power of using a logical representation.

VII. Generating Playable Mini-games

At this point we have described how to create, enumerate, and expressively sculpt the generative spaces of game rulesets. However, we have yet to show how to transform such sets of assertions about how a game should work into functioning games which operate as the generator designs them. In this section we will describe the concrete software components which bring ASP-based ruleset generation into contact with the player in our prototype of VF: the game generator and the game engine.

A. Game Generator

Our game (ruleset) generator is manifest in two distinct parts. The first is the logical definition of a design space using an ASP as described above, and the second is the software we use to enumerate games in the design space. We have adopted the freely available LPARSE and SMODELS tools[8] which, respectively, translate first-order ASPs into simplified, grounded logic programs and solve for the desired number of answer sets, outputting each as it is found.

To surface the functionality provided by these highly obscure (from a game programming perspective) command-line tools, we created a minimal web-service wrapper which allows any HTTP-capable program to request a stream of answer sets to a given ASP. This wrapper allows us to be much more flexible about the kind of game engine we use to consume the generated game designs.

The result of organizing our generator in this way is that the designer of the generative space does not need to think about (nor necessarily understand) the underlying generation algorithm. Indeed, different solvers that consume LPARSE groundings may employ radically different algorithms while being indistinguishable at the level of answer set generation.

B. Game Engine

Realizing the outputs of our game generator in the glowing vector-art aesthetic (and supporting low-level mechanics such as movement with momentum and collision detection/resolution) requires the use of a game engine. We built our game engine using the Flash game library Flixel[9] as a base. To this base, we added skeletal support for the elements we knew our ruleset generator would like to instantiate: abstractly depicted characters which can roam about, bumping into each other and obstacles, victory condition templates, basic level generation algorithms, and background image generation.

At each major design decision which would normally be hard-coded in a particular game (such as how big the game world is, which character the player controls, etc.) we added code to consult a configuration object (to be provided by the generator). Though making an engine that supports many

[8] http://www.tcs.hut.fi/Software/smodels/
[9] http://flixel.org/

possible games is significantly more difficult than making any particular game, the task is similar in complexity to the integration of a scripting language which many complex games already possess to ease the development process.

The general flow of our prototype works as follows. On startup, the engine sends the internally-stored ASP (logic program) to our ASP solver service and begins streaming in solutions over the network. The player can randomly sample mini-game rulesets, given a basic textual preview of the game (describing what they control, the goal of the game, and other details selected by the generator). Upon selecting a mini-game, play begins. The mini-game's rules are fixed across restarts when the player character is "killed". The player rejoins the initial game selection screen upon victory or intentionally abandoning a difficult game.

In the design of VF beyond the prototype, we envision performance in mini-games linked to a resource which can be spent to either unlock new reaches of an initially small game mini-game design space or buy constraints which enforce interesting patterns. Through incrementally refining the possibilities open to the generator the player can slowly come to understand the interaction between the various modular mechanics set into the VF universe. Each new game design element the player unlocks results in new "`tech(T)`" assertions being simply appended to the text of the internal ASP, which, in concert with existing integrity constraints and preconditions, means the design space of mini-games is dramatically reshaped during play at a scale unmatched by any other game.

VIII. Discussion

A. Coupling between Generator and Engine

While the definition of the a design space is strongly separated from the means of sampling the space in our approach, there is a strong coupling between the content generator and the game engine which consumes the content. This mandatory coupling is not problematic if one considers the generator and the engine to be two parts of a single program. In the VF game prototype, the game engine and ASP code used for ruleset generation are combined into a single binary that runs in the player's browser while the ASP solver runs independently on remote server.

B. Tradeoffs in Levels of Abstraction

The relative expressiveness of the generator compared to the engine depends on the level of abstraction used by the logical terms with which they communicate. As the generator takes on more responsibility for defining the mechanics of games (e.g. working with lower-level terms to implement movement models instead of simply instantiating them), it becomes even more critical that we be able to sculpt the design space to avoid the swath of well-formed yet meaningless or broken (in terms of gameplay) constructions which a grammar might admit.

At the low-level extreme of generating the equivalent of machine instructions, clearly an astronomically tiny fraction of such "rulesets" would represent valid games and we would have a hard time writing down constraints which have any useful effect. Meanwhile, at the high-level extreme of parameterizing a game by only a few configuration values, the space of games (even if all were guaranteed to be valid) would be uninteresting for a player to explore. The challenge, which we have taken but a single step towards, is to work at the lowest, code-like level possible (enabling the richest variety) while not losing control of the design space and consequently asking the player to pay a non-game.

C. Towards Automating Game Design

VF is a single experiment in larger effort to automate the creativity-intensive practices of game design. We draw inspiration for this task from the Robot Scientist project [12] which represents a very ambitious effort to automate the practices of computational biology. Iterative experimental design processes in science have a meaningful analog in game design in terms of prototyping and playtesting games. Abductive logic programming (supported by the very same ASP tools we use), plays into nearly every creative and/or scientific task the Robot Scientist performs.

In a serious attempt at automating game design, we imagine integrating the ruleset generator with an automated playtesting tool such as BIPED [9]. The logically-described playtesting feedback can be used with inductive logic programming, such as provided by Progol [13], to learn (or induce) logical predicates which predict player choices and reactions. This same rule-learning could be used to find useful constraints on the generative space of rulesets. Knowledge gleaned from experience testing can be embedded into the generator, forming a closed-loop design system which better approximates the iterative design process used in the game industry.

From a scientific perspective, small game prototypes are analogous to experimental setups, designed to elicit a demonstration of some natural behavior (or player behavior for games). Continuing the metaphor, we envision iterative game design as process that, instead of aiming to produce games, aims to produce knowledge about play, producing games as a byproduct of the discovery process. A new challenge, now, is in building a generative space of game design patterns and player model elements which would serve as the building blocks of theories in a game-design-as-science paradigm. In the future, we will examine the use of ASPs to represent these generative spaces as well.

D. Evaluation

Towards gauging our level of success in this project, we are most interested in expressivity of our generation approach and the constraints it employs to shape generative spaces for the application of procedural content generation.

In our experience with the VF prototype, we found 100-line ruleset to be abundantly generative, generating rulesets which exposed legitimate bugs in our game engine and rais-

ing important game design issues (such as how the momentum of Asteroids-style characters model should change in collision with roguelike-style characters). With integrity constraints, it was both easy to both zoom in on failure cases for testing and to forbid the occurrence of situations we had not yet resolved.

In terms of expressivity of constraints, recall the space of indirect-push-kill (described previously). This scenario demonstrates how we can encode additional knowledge about a game design space (such as how to produce a high-level plan to win games in it) into the generator itself. Knowing that high-level descriptions such as "`winnable_via(indirect_push_kill(red,blue))`" are present in the games definitions allows us to write single-line constraints requiring or forbidding high-level patterns. Such inline self-analysis of games could also be used to prepare a small manual for each generated game. EGGG [8] was able to produce documentation for the game it generated, also by virtue of having so much design knowledge baked into the generator.

Evolutionary methods such as Togelius' system aim to produce rulesets optimized by some metric (with Brown's LUDI system [14] even striking commercial success in autonomously designing Yavalath[10]). However, in PCG, there is an inherent demand for sizable spaces with significant, player-visible variety. Our declarative approach allows easy manipulation of generative spaces of arbitrary size, while evolutionary approach only maintain a fixed-size population (on the order of tens or hundreds) during its search for a single, optimal individual. While both approaches are highly declarative in their own sense, we believe our approach is distinctly more space-oriented, and is therefore better suited for PCG (also recall that it is not actually specific to ruleset generator).

As a final form of evaluation, we can look at our ruleset generator as a creative system (though it was not designed as one). In the field of computational creativity, a standard means of evaluating an artifact generator are to look for novelty and value in the artifacts it creates [15]. One instance of novelty we experienced was a seemingly unwinnable game. After some effort we realized the game could actually be won by indirectly pushing an intermediate character into the characters it was our goal to kill. Excited by this occurrence which was both unexpected (novel) and fun (fun), we quickly devised the indirect kill detection logic described previously to zoom in on other games of this variety, transforming the generative space.

IX. Conclusion

In seeking the technology to support a novel game design, we have developed a new content-generation approach and applied it to the challenging domain of game ruleset generation, producing a large space of playable mini-games. The flexible representations afforded us by ASP allow us to make concise-yet-powerful modifications to this design space. This representation schema has also prepared us for a more serious attempt at automating game design.

We invite the reader to try playing several mini-games in our public demo[11] and consider what new options should be made available to the generator and which combinations should be forbidden to avoid bad interactions between elements.

References

[1] C. Remo, "MIGS: Far Cry 2's Guay on the Importance of Procedural Content," *Gamasutra*, November 2008.

[2] V. Hom and J. Marks, "Automatic design of balanced board games," in *Proc. 3rd Artificial Intelligence and Interactive Digital Entertainment Conference*, 2007, pp. 25-30.

[3] M. J. Nelson and M. Mateas, "Towards automated game design," in *AI*IA: Artificial Intelligence and Human-Oriented Computing*, 2007, pp. 626-637.

[4] T. Fullteron, *Game design workshop: a playcentric approach to creating innovative games*, 2nd ed.: Morgan Kaufmann, 2008.

[5] J. Togelius and J. Schmidhuber, "An experiment in automatic game design," in *Proc. IEEE Symposium on Computational Intelligence and Games*, 2008.

[6] R. Koster, *A theory of fun for game design*.: Paraglyph press, 2005.

[7] B. Pell, "Metagame: A new challange for games and learning," in *Heuristic Programing in Artificial Intelligence 3: The Third Computer Olympiad*, Ellis Horwood, Ed., 1992.

[8] J. Orwant, "EGGG: Automated programming for game generation," *IBM Systems Journal*, vol. 39, no. 3-4, pp. 782-794, 2000.

[9] A. M. Smith, M. J. Nelson, and M. Mateas, "Computational support for play testing game sketches," in *Proc. 5th Artificial Intelligence and Interactive Digital Entertainment Conference*, 2009.

[10] I. Niemela, "Logic programs with stable model semantics as a constraint programming paradigm," *Annals of Mathematics and Artificial Intelligence*, vol. 25, pp. 241-274, 1999.

[11] A. C. Kakas, R. A Kowalski, and F. Toni, "Abductive logic programming," *Journal of Logic and Computation*, vol. 2, no. 6, pp. 719-770, 1992.

[12] R. King et al., "Functional genomic hypothesis generation and experimentation by a Robot Scientist," *Nature*, vol. 427, pp. 247-252, 2004.

[13] S. Muggleton, "Inverse entailment and Progol," *New Generation Computing Journal*, pp. 245-286, 1995.

[14] C. B. Browne and F. D. Maire, "Evolutionary Game Design," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 1, pp. 1-16, 2010.

[15] A. Pease, D. Winterstein, and S. Colton, "Evaluating machine creativity," in *Workshop on Creative Systems, 4th Intl. Conf. on Case Based Reasoning*, 2001.

[10] http://www.cameronius.com/games/yavalath/

[11] http://eis.ucsc.edu/VariationsForever