Answer Set Programming for Procedural Content Generation: A Design Space Approach

Adam M. Smith and Michael Mateas

***Abstract*—Procedural content generators for games produce artifacts from a latent design space. This space is often only implicitly defined, an emergent result of the procedures used in the generator. In this paper, we outline an approach to content generation that centers on explicit description of the design space, using domain-independent procedures to produce artifacts from the described space. By concisely capturing a design space as an answer set program, we can rapidly define and expressively sculpt new generators for a variety of game content domains. We walk through the reimplementation of a reference evolutionary content generator in a tutorial example, and review existing applications of answer set programming to generative-content design problems in and outside of a game context.**

***Index Terms*—Game design, procedural content generation, logic programming, answer set programming, constraint programming**

I. Introduction

PROCEDURAL content generation (PCG) is a game-design technique that involves creating game content via automated processes rather than via hand-authoring. PCG readily provides the means to generate entire game worlds on the fly, ranging from the galaxies of the venerable *Elite* (Acornsoft 1984) to the infinite, rolling terrains and subsurface caverns of the recent *Minecraft* (Mojang Specifications 2009). Beyond generating the physical details of a game world, PCG is applicable to more abstract content such the detailed histories of lost cultures in *Slaves to Armok: God of Blood Chapter II: Dwarf Fortress* (Bay 12 Games 2006), even generating the mechanics for the various stages in *ROM CHECK FAIL* (Farbs 2008).

As the name suggests, PCG revolves around a *procedure* which is used to generate instances of content, which we call *artifacts*. As many content generators are non-deterministic, it is meaningful to talk about the *generative space* or *design space* of a generator: the set of artifacts that it will eventually produce given some (optional) input.

In the context of a complete game design, the shape and population of this space can have a dramatic impact on gameplay experiences. A generative procedure may fail by producing an undesirable artifact such as an unsolvable puzzle, a nonsensical story, or, worse, level data or mechanical logic that crashes a game engine. Design concerns, such as avoiding pathological failures, dictate constraints on the design space of a generator, some of which may be very difficult to resolve without an extensive redesign of the generative procedure.

Often, what constitutes a success or failure may not even be clear until many artifacts have been sampled from a preliminary design space. As a design problem, PCG inherits a default "ill-definedness" in requirements. What *exactly* are you looking for in these artifacts? Thomas and Carol suggest that this uncertainty in the requirements of a design problem should be resolved, in iterative steps, by proposing candidate solutions [1]. Properties of certain candidate artifacts may inspire a revision to the definition of the content design space, spurring the need for a new generator.

To minimize commitments to a particular generative process and support direct, iterative refinement of a generative space by a designer, we look to the declarative specification methods afforded by answer set programming (ASP). We do not propose a new algorithm for generation. Instead, we suggest encoding domain-specific PCG problems as the well formalized problem of generating answer sets, which can readily be solved by several existing domain-independent algorithms. This, coupled with programming guidelines for how to create and evolve generative spaces described with ASP, represents a novel solution to the meta-level problem of generator design.

This paper is intended to provide equal weight in its technical, tutorial, and survey contributions. Our *technical contribution* is the general mapping of PCG problems to answer-set programming–incrementally specifying a design space in a declarative language and using an off-the-shelf ASP solver as the run-time generative component. This methodology links PCG, as a two-layered design problem, to the design studies and declarative programming literature and contrast deeply with feed-forward and generate-and-test PCG techniques. Our *tutorial contribution* is a game-flavored introduction to ASP terminology and software engineering practices along with a code-level walkthrough of reimplementing an existing PCG system. Finally, our *survey contribution* is the contextualized review of the only two existing applications of ASP to PCG and two more ASP applications with a generative focus outside of game content.

The overarching theme of this work is that the tools used in PCG should respect in inherent ill-definedness of content design problems and assist the designer, not only in literally producing the content, but in coming to understand what makes potential artifacts desirable. We offer the iterative exploration

Manuscript received April 14, 2011. This work was supported in part by the National Science Foundation, grant IIS-1048385. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

A. M. Smith and M. Mateas are with the Expressive Intelligent Studio at the University of California, Santa Cruz, Santa Cruz, CA 95064 (email: {amsmith,michaelm}@soe.ucsc.edu).

using declarative specification for design spaces afforded by ASP as an example of this kind of tool.

II. Motivation

We claim PCG is concerned with two design problems. The first is a concrete design problem dealing with the *production of game content* with desirable properties. The second is a meta-level problem dealing with the *production of generative procedures* with desirable properties (such as, amongst others, the ability to produce desirable content). While we might judge individual artifacts by their effects on gameplay or their aesthetic qualities, we should judge generative procedures on their runtime performance, expressiveness[1], and manipulability in the face of shifting design requirements (mnemonically, we are looking for generative procedures that are fast/fast/flexible). Solutions to the base, artifact generation problem should be found by machines (likely via automated search within a design space), and solutions to the meta-level, generative space design problem should be found by human designers (likely via iterative refinement of the design space's definition).

The essential problem in PCG, then, is to pick a design space that is both feasible to algorithmically sample while also containing only those artifacts that are desired. Solving this problem requires experimentation, both in generator design and in

Looking first at the artifact generation problem, a traditional decomposition is the generate-and-test process [2]. Generate-and-test is a proven method for automating artifact design in PCG, with success in several game content and visual art domains [3] [4] [5] [6].

The design space captured by a procedure based on this decomposition is the emergent result of composing the "generate" and "test" procedures. Some commitments made in the generate procedure (perhaps to a specific set of search operators) or in the test procedure (perhaps to an evaluation function) represent some *accidental complexity* in the larger PCG system [7]; the final artifacts could have been produced by a wildly different procedure that happened to cover the same design space. During development, generate-and-test processes are often refactored while preserving the design space they encode, often folding parts of the test deeper into the generator as a means to improve run-time efficiency [8]. At the same time, though some shifts in requirements on the design space may manifest as localized tweaks to a test procedure, other changes (particularly those that involve changing the level of abstraction used in representing artifacts) will require a complete overhaul of both the generate and test procedures.

Thus, in easing refinement of a generator by providing direct control over a design space, we should minimize commitments to procedural details. We should focus only on assertions about artifacts or the shape of the space from which they come. Direct specification should allow us to expend less effort in getting a content generation project started, and throw away less effort as design requirements change in light of new experiences. Doing this requires an alternate paradigm which allows us to factor out procedural details.

Declarative programming (in which knowledge is expressed, absent of control flow; describing *what* to compute instead of *how* to compute it) holds promise for avoiding accidental complexity [9], and it has already been used for a number of PCG applications. The declarative nature of the Tutenel's semantic scene description language [10] and Smelik's SketchaWorld system [11] is reported to reduce designer effort and provide a more intuitive mode of expression. Likewise, the use of a constraint solver in Tanagra reduced overall system complexity by placing the details of low-level geometry placement out of sight, foregrounding the constraint structure of playable platformer levels [12]. Even the use of design grammars, such as in Dormans' work in generating missions and maps for adventure games [13], is an example of content generation using a declarative representation of the design space.

Answer set programming has emerged as a declarative programming paradigm with particularly potent affordances for describing the design spaces of PCG problems. Although nominally designed for knowledge representation and search-intensive reasoning tasks, it is easily repurposed for "answer set synthesis" in which ASP is exploited primarily for its generative capabilities [14].

In LUDOCORE [15], an ASP-backed framework for producing formal models of videogames, a designer can use "structural queries" to solve for (or generate) elements of game content that are consistent with dynamic gameplay constraints. While LUDOCORE primarily used ASP to implement gameplay trace inference, generation of rudimentary dungeon maps (as the presence and connectivity of rooms) was a welcomed side-effect.

In *Variations Forever* [16], our recent experiment with PCG for mini-game rulesets (described in more detail in the survey later), we sought out ASP as a programming paradigm capable of expressively handling code-like game content. The approach, it turned out, was quite general. Further, we found reshaping the design space of our generator so engaging that we suggested using online, player-driven expansion and sculpting of the space as a novel game mechanic.

In this paper, we generalize the PCG approach first articulated in *Variations Forever* and demonstrate its use in a variety of domains. This approach includes concrete code strategies for creating answer set programs that model design spaces and meta-level strategies for evolving a design space in response to experience with generated artifacts.

III. AnsProlog and ASP

Before we launch into using ASP for content generation, the reader should have a basic level of literacy with AnsProlog, the language accepted by common answer set solvers [17]. Note that "answer set programming" refers to the programming paradigm (as one would refer to the paradigms of functional or object-oriented programming) and "AnsProlog" refers to a concrete syntax one uses to write answer set programs (as one would refer to Scheme or Java syntax). Though they share common syntax features for the description of logical terms, answer set solvers are not Prolog interpreters.

[1] Here, "expressiveness" refers to the ability to generate artifacts with coherent, fine detail.

A. Basic Logic Programming

AnsProlog syntax is derived from Prolog, the well-known deductive logic programming language. As such, both code-like and data-like knowledge is represented using a common scheme: logical terms. Terms may either be atoms (named symbols, numbers, or strings) or compounds consisting of functor (a symbol) and a list of logical terms as arguments. Collections of logical terms can readily represent any data structure; the following terms might describe properties of various game content artifacts:

```
teleportation_disabled.
initial_health(100).
weather_model(springtime).
allies(humans,elves).
damage(sword_of_might,11).
scripted_event(spawn(boss,temple),120).
valid_move(rock),
valid_move(paper),
valid_move(scissors),
valid_move(lizard),
valid_move(spock).
phase(1,movement),
phase(2,combat),
phase(3,diplomacy).
```

Each set of the above terms was terminated with a period to indicate that they could also be interpreted as simple sentences in first-order logic, i.e. facts. More complex logical sentences, called rules, are possible using logic variables and the if-operator "**:-**" (called the neck because it connects the head of a rule to a body). These rules might be used to derive properties of an artifact described in terms of facts like those above. Note the use of a comma in rule bodies to mean "and" and reuse of a rule's head to signify an "or" between the various clauses of a rule:

```
plateau_at(X) :-
  height(X-1, H), height(X, H), height(X+1, H).
hostile(A,B) :- enemy(A,B).
hostile(A,C) :- enemy(A,B), friend(B,C).
hostile(A,C) :- friend(A,B), enemy(B,C).
```

The first rule above roughly translates as "there is a plateau at a position if its immediate neighbors have the same height" (where *X* is the position and *H* is the particular height level they share). The second rule is more complex; it captures the logic of this statement: "I am hostile to someone if they are my enemy, if they are the friend of one of my enemies, or if they are the enemy of one of my friends."

We say that the collection of rules and facts with structurally-matching heads defines a predicate, a logical condition which may either be true or false for each instantiation of its arguments (e.g. hostility may perhaps exist between two characters Alice and Bob, but not between Alice and Eve). Some predicates are extensionally defined by a list of facts (as in a modern database) while others are intensionally defined by a set of rules (for which unbound variables are treated as universally qualified, in a logical interpretation). It is sometimes useful to think of facts as simply rules with no body (in fact, solvers treat them as such).

B. Answer Set Programming

While general facts and rules are common to all logic programming languages, AnsProlog uses two additional constructs: choice rules and integrity constraints. These constructs are the key to the generative faculties of ASP.

Where traditional logic programming is concerned with what *must* be true in some logical world, *choice rules* allow the description of what *might* be true (facts available for inference through abductive reasoning). In choice rules, braces are used to group a collection of terms, some number of which might be true as facts in the logical world. The following is traditional example of reasoning with choice rules:

```
{ rain, sprinkler }.
wet :- rain.
wet :- sprinkler.
dry :- not wet.
```

Translating, this snippet says that it might have rained and a sprinkler might have been on (or both, or even neither). It also says both rain or sprinkler necessarily imply wet (grass, perhaps). The final rule allows us to derive the expectation of dryness if there was no means of producing wetness.

ASP takes its name from its focus on "answer sets". Answer sets are the collections of ground (variable-free) facts that are consistent with the logical worlds an answer set program describes. The small program above admits four answer sets, each representing the combination of things that might be true along with the necessary deductive consequences of those selections:

```
dry.
wet, rain.
wet, sprinkler.
wet, rain, sprinkler.
```

Meanwhile, *integrity constraints* let a programmer express what *must not* be true in a logical world, independent of what other rules say. They resemble traditional rules with a missing head (they can be imagined to read "implies contradiction").

In the grass scenario, we can incorporate the new knowledge that, perhaps, our sprinkler has an automatic shut-off that prevents it from running in the rain. To do this, we simply write the integrity constraint that sprinkler can never be assumed at the same time rain (i.e. the conjunction of sprinkler and rain implies a contradiction):

```
:- sprinkler, rain.
```

The combined program now only admits the first three answer sets. While an equivalent program could have been written using choice rules alone, the benefit of integrity constraints comes from their ability to filter out (or reject) undesirable answer sets without having to understand the process by which those undesirables might arise. Integrity constraints do not simply block the final display of certain answer sets; they actually prevent undesirable answer sets from ever being generated in the first place.

C. ASP Solvers

To generate a number of answer sets for a given answer set program, one feeds the program to an ASP solver. This process is conceptually similar to how one may feed problems to a SAT (Boolean satisfiability checking) solver in order to produce one or more satisfying truth assignments. Indeed, some ASP solvers make use of unmodified SAT solvers internally [18]. Regardless of solver choice, equivalent results are guaranteed by the semantics of AnsProlog. Solvers may be treated as interchangeable black boxes. Even those solvers which are capable of using incomplete (but potentially faster) search algorithms will fall back to a complete search algorithm when the heuristic method flounders [18]. Genetic algorithms have been proposed for this role [19], and ant colony optimization has been demonstrated as a core answer set solving algorithm in small-scale examples [20].

The interaction between solver and problem in ASP differs greatly from the setup used with metaheuristic search processes. Generic, "black-box" search algorithms are only allowed to access the problem domain by invoking an evaluation procedure on complete, candidate solutions. Because the evaluation procedure is opaque from the search algorithm's perspective, the problem designer is free to use any means of implementing the procedure they see fit. Having a single, tightly controlled channel for domain-specific to enter the search algorithm makes these algorithms very general, even domain agnostic. At the same time, it also strongly limits the search algorithm's access to knowledge about the domain's structure which could speed up the search process. By contrast, though also domain-agnostic, ASP solvers require a "white-box" declaration of the domain structure in a language the solver can recognize and reason over. While this can in some cases require more effort than producing a black-box evaluation procedure, it avoids the need to invent informative numerical evaluation metrics for domains that lack a natural metric. ASP and metaheuristic search, as generation techniques, fundamentally differ in their *problem formulation*.

Common ASP solvers generally use search algorithms that are not explainable as kind of generate-and-test processes used in metaheuristic search algorithms; the Davis-Putnam algorithm often used as a base for more specific answer set solving algorithms fundamentally works property-by-property [21], rejecting whole subspaces of potential solutions before any are even fully "generated". Applied to a traditional maze generation problem where the solver must place walls on a map while ensuring the maze's finish is accessible from the start, the solver will conceptually build the map wall-by-wall, analyzing a potential maze based on properties of only those walls and passages committed so far. In this way, the solver can reject the large subspace of potential mazes which include walls that completely surround the start of the maze after only a few exploratory commitments. Having reached this dead-end in the design space, a Davis-Putnam inspired solver will backtrack and try alternatives for its recent choices. For more detail, AI textbooks such as Russel and Norvig's [22] (page 221) will generally provide approachable pseudocode and examples for the well-known Davis-Putnam algorithm and its variants.

By contrast to these algorithms, a generate-and-test process would entail generating a complete maze (even if it only had a few walls) before attempting to test the maze for validity (reachability in this case). The important distinction is that generate-and-test process always evaluate individual artifacts in a global manner whereas most ASP solvers will evaluate an incompletely defined artifact (standing in for the space of all artifacts which share the committed substructure) in a local manner, allowing the search process access to information about which substructures to credit or blame for an artifact's overall desirability.

Many ASP solvers even incorporate opportunistic learning that finds ways to simplify the search process during execution (learned "nogoods" are automatically discovered bounds on infeasible regions of a search space [23]). In a sense, these advanced solvers self-refactor (in a design-space-preserving manner) in response to emergent structures in the search space.

This is not to say that generation via answer set solving will actually be faster than with a generate-and-test process. Instead, we point out that common ASP solvers employ sophisticated search processes that are meaningfully distinct from generate-and-test processes. From the perspective of the programmer using an ASP solver, whether or not that particular solver uses generate-and-test internally is an invisible implementation detail so long as the AnsProlog language semantics are respected.

While there are several ASP solvers available, we have been most productive using Clingo [24], an advanced, integrated solver from the Potassco toolset [25]. We chose Clingo primarily for its rich documentation and simple command-line interface.

IV. A Method for Using ASP for PCG

Recall that the concrete design problem in PCG is the generation of artifacts from some design space (the set of artifacts with desirable properties). This design space needs some representation that a machine can understand before any automatic generation can be done.

Instead of jumping all the way into committing to a partitioning between isolated generate and test procedures, we propose capturing a design space as the range of answer sets admitted by an answer set program. Doing so, we can directly import the sophisticated search algorithms used in ASP solvers into our specific PCG domain "for free", without the cognitive overhead of juggling the implementation of a generative procedure. The use of a declarative, logical language to express a design space keeps the focus on properties of artifacts and the properties of the space itself.

Fig. 1 illustrates how ASP solving parallels the intent of the concrete design problem. Because the design space is a conceptual construct accessible only to the designer, one can only directly generate artifacts from this space via manual construction ("hand authoring"). The design space is modeled (explained to a machine) as an AnsProlog program, which is given to an off-the-shelf ASP solver to produce answer sets, which can be interpreted in the context of some game to construct the artifacts the answer sets describe. Generated artifacts will often inspire changes to the design space, allowing new structures or rejecting undesirable emergences. In the remainder of this section we will keep the discussion at a high

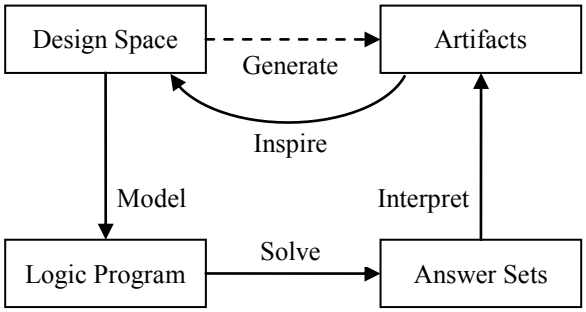

Fig 1. In our method, the *intent* to generate artifacts from a (conceptual) design space is carried out by first modeling the design space with a logic program, and then invoking a domain-independent solver to produce answer sets which can be interpreted as descriptions of the desired artifacts. Experience with generated artifacts inspires redefinition of the design space. This diagram mimics a similar diagram in ASP software engineering for which "design space" and "artifacts" replace the more generic "problem" and "solution" [26].

level, as the subsequent tutorial example and case study sections will strongly ground the method.

A. Representing Artifacts and Spaces

Artifacts, in ASP, are represented by logical facts that describe their in-game properties. There is a certain minimal set of facts that is needed to reconstruct an artifact in the context of a game (for a maze: the start and finish locations, along with traversability of the map). However, we will also describe artifacts with extra annotations about properties of an artifact which are derivable (analyzable) from its basic structure (for a maze: the reachability of a given location, the length of the shortest solution, the number of dead ends, etc.). While these annotations are not needed in the running game, they provide a powerful vocabulary for describing the shape of a design space.

The design space itself is represented with some assertions about what properties individual artifacts might have (using AnsProlog choice rules) and other complimentary assertions about properties artifacts must or must not have (using AnsProlog integrity constraints). Passing such a design space model to an ASP solver produces the collections of facts we use to construct the in-game artifact.

Supporting the idea of basic structure and optional annotations, the design space representation may also use logical rules to describe how artifacts should be analyzed (these are the rules by which those annotations are deduced). As a design space model becomes more refined, the bulk of its AnsProlog representation may be devoted to building up a sufficiently nuanced vocabulary to express a critical property that all desirable artifacts must or must not have. This complexity is unavoidable when that property really is a definitional aspect of the designer's intention.

Returning to the maze generation scenario, the program describing a design space of mazes might contain rules for deriving the minimum solution length for a maze and integrity constraints to require certain bounds on that length. While this information is never displayed to a player, it is an integral commitment of the maze design space we will describe in the tutorial example later.

By describing the schema, requirements, and analysis of artifacts in a declarative manner, the designer using ASP simultaneously avoids commitments to (some) accidental complexity, and gives the ASP solver the knowledge that it can use to adapt its internal search process to the domain at hand (via constraint propagation, clause learning, and other techniques).

B. Modeling, Interpretation, and Refinement

In applying ASP to a map generation problem, modeling is the process of capturing the design space of desirable maps as a logic program. Interpretation is the process of importing logical facts about a particular map into a game where it can be played. Clearly, both need to be addressed, at least in a tentative fashion, before we can feel the implications of our design space on the gameplay experience.

An advantage of using ASP solvers over hand-crafted generative procedures is that they can be used as black boxes – detailed knowledge of their internals is not required (or even visible). The details of the modeling and interpretation processes that map our concrete artifact design problem into ASP are, however, critical. For the meta-level design problem of sculpting an appropriate artifact design space, both processes need to work together so that a designer can sample artifacts in-game and make intelligent decisions about the next iterative design move.

The first thing that needs commitment is the schema for (or expected structure of) logical terms that will be used to represent artifacts. As described in the previous section, these terms have a universal representational ability; however, we have found many PCG problems to be well covered using only global Boolean flags, sets, and simple table structures. Concrete examples will be given for the various systems described later in the paper, but the game-themed logical terms in the previous section should spark the imagination.

Once the basic structure of artifacts has been decided, the designer/programmer can start the interpretation process by creating a loader for a few hand-written answer sets into a game engine where the artifacts will be observed. The fixed grammar of logical terms makes them relatively easy to parse and convert into the data structures required by a game engine.

With basic interpretation in place, it is time to replace the hand-written answer sets with the output of a minimal ASP-based generator. For each type of term used to describe artifacts, one or more choice rules should be created which will allow the blind generation of terms that are at least in the right language. These core choice rules can be imagined to define a basic "generate" procedure (or a default design space), though really they form a specification to which a procedure should conform.

Now, using a solver, the programmer can generate several answer sets from the extremely broad, basic generative space (which will probably include many obviously undesirable results due to a lack of constraints). The presence of undesirable artifacts in the minimally constrained generative space becomes the feedback that drives refinement of the model, which is a step in the larger design-space "sculpting" process (building out support for new structures and then carving away unwanted interactions). For example, in developing a map generator, noticing unreachable areas of terrain might suggest the addition of a constraint that would forbid this from happening

in the future (perhaps by requiring a viable path between all locations of interest).

Again, while the constraints that the programmer adds during the iterative development of a generator seem to function as post-filtering processes, most ASP solvers will embed them into the core search process, interleaving their evaluation with the construction of partial artifacts.

As the logic program grows in detail, with choice rules gaining more detailed bodies, integrity constraints rejecting corner cases, and auxiliary rules defining higher and higher level patterns in the artifacts which can be used in filtering, the details of interpretation process will likely change very little (as changes to the basic structure of artifacts is rare). In the rare case of making a representation change, often auxiliary rules can be used to deduce the facts describing a new-style artifact in the schema of the old-style artifacts, allowing the reuse and incremental upgrade of an interpretation procedure.

Interpretation need not always refer to the literal loading of answer sets into a game engine. Instead, answer sets may be interpreted as describing the inputs to a different PCG algorithm which expands them to produce the final in-game content or even as the inputs to another declarative, solver-based process. Both of these expansion techniques are employed in DIORAMA, a map generation system described later in this paper.

For a detailed walkthrough of software engineering practices for ASP, we refer the reader to the (illustrated) "Pragmatic Programmer's Guide to ASP" which is aimed at general problem solving as opposed to the specific PCG context introduced here [26]. The iterative programming practice it describes for finding solutions to logical problems exactly corresponds to a human-directed, meta-level search in the space of content generator designs in the PCG domain.

C. Modeled and Unmodeled Properties

AnsProlog integrity constraints provide very expressive, direct control over the properties of generated artifacts. In fact, they combine the convenience of a post-filter that might normally be applied in one of the phases of an explicit generate-and-test process with the runtime benefits of providing this filtering knowledge to the ASP solver (which may even run faster in the presence of additional constraints). However, this description is somewhat misleading because it only applies to properties of artifacts which can be modeled using logic expressible in AnsProlog.

In contrast to traditional (Prolog-style) logic programming (a Turing-complete programming paradigm), ASP specifically targets NP-complete problems (informally, those problems for which solutions are easy to verify). As a result of this, there are computations which cannot be expressed in AnsProlog, particularly those with infinite loops (conforming solvers will always terminate in finite time). Many solvers support extensions which allow a programmer to encode variants on the Weighted MAX-SAT problem (a problem just outside of NPC which enables combinatorial optimization as well as the use of soft constraints).

Nonetheless, many interesting properties of artifacts are impractical to accurately capture in AnsProlog. The results of running an arbitrary program or the elicitation of a human of feedback are obvious examples of such properties that would be left unmodeled in an ASP-based generator. Consider this example from the generative visual art domain: The output of a NEvAr, a Neuro Evolutionary Art system, is often interesting when its otherwise very abstract compositions resemble a human or animal face [5]. NEvAr uses neural network trained on audience feedback to capture a local, fuzzy sense of interestingness which can permit the system to generate many face-like images without a logical model of how images come to resemble faces. This neural network evaluation (rich with floating-point arithmetic) is a natural fit for a generate-and-test architecture, but is impractical with the purely-symbolic ASP framework. While future ASP solvers may eventually integrate efficient support for floating-point arithmetic, other properties of artifacts are permanently off-limits to machines.

Often, only human inspection is capable of accurately judging the most subtle properties of artifacts (such as beauty or fun). Aside from literally embedding a human in the generation process (as in the case of interactive genetic algorithms, manual post-filtering, or inductive logic modeling[2]), the primary tactic is to substitute the subtle property for a more practical one (as done by the neural network in NEvAr above). In ASP, a common tactic for producing approximate property descriptions is to collect definitions for a series of concrete failure cases – in the maze generator described later in this paper, accurately capturing the "difficulty" of a maze is an AI-complete problem, but recognizing a collection of ways in which a maze appears "too easy" is a tractable logical modeling problem.

Where unmodelable properties of artifacts are a requirement of the design space, a declarative, solver-based generator can be nested inside of a larger generate-and-test process, as done with Choco, the numerical constraint solver, in Tanagra [12]. It bears mentioning, however, that ASP provides a dramatically wider modeling palette than do numerical constraint solvers, especially in concert with a gameplay modeling framework such as LUDOCORE [15]. Some solvers actually modestly extend AnsProlog syntax to allow the use of an embedded numerical constraint solver, supporting mixed structural-numerical search over integer domains [27] (generic solvers will treat such problems purely structurally, incurring inefficiency). While increasingly specialized solvers have better runtime performance, they generally come with reduced productivity for the designer/programmer (who is now responsible for more complex modeling and interpretation work) [28].

The focus on quickly modeling properties of artifacts is perhaps the most important idea in our method. We claim that it is easier to sculpt a design space by repeatedly carving away undesirable regions (identified by describable flaws) than it is to guess a procedure which implicitly defines the same space.

D. Applicability

Although it is difficult to characterize the class of problems for which ASP is a desirable implementation technique (as problems can be solved to various degrees and by several solutions), we can, however, describe the space of solutions afforded by ASP.

[2] Inductive logic programming has been proposed as an ASP-compatible way of automatically producing approximate player models for games [15].

ASP provides a means for rapidly creating and easily modifying *search-based* generators over finite data structures. We emphasize the "search-based" label here to point out the fact that ASP-based generators are meaningfully grounded in search while being strictly incompatible with the definition of term given in a recent PCG survey [3]. ASP solvers use search to generate content without the use of a real-valued fitness function or contingent generation of new artifacts based on the score assigned to previously evaluated, complete artifacts. In the existing applications we survey later, three of the four systems lack any numerical properties which are meaningful to optimize, and the exception, DIORAMA, uses optimization as a means to implement layered preferences.

We should clarify that ASP's focus on finite data structures refers to a countable set of alternatives to search over. A vector of real numbers has a fixed number of dimensions, but it has an infinite number of concrete instantiations. Meanwhile, planar graphs (perhaps representing connectivity between rooms in a generated dungeon map) have a variable structure, but readily become a finite search space if we upper-bound the number of nodes that may be used. The same applies to bounded trees, sequences, and other data structures.

Thus, ASP is applicable to problems where the apparent task is to select structures with desirable properties from a vast but countably finite space of structures, where these properties can be described using AnsProlog. While few PCG problems meet this description exactly (due to requiring properties that are impractical to model in logic or involving infinite spaces), many problems will contain significant subproblems that do meet this condition. Whether ASP is an attractive choice, then, hinges on the complexity of factoring out the ASP-solvable problem and integrating its solution into a larger generator. The DIORAMA system, described later, provides an excellent example of this nesting of an ASP-based generator inside of a general Python program.

For applications where finiteness would exclude the use of an ASP solver, another declarative, solver-based solution will often be available and provide similar benefits. For unbounded structural spaces, HYPROLOG implements the same abductive reasoning that supports the generativity of ASP [29]. For continuous spaces, CLP(Q,R) blends non-linear constraints and optimization for the rationals and real numbers with traditional logic programming [30]. Such alternatives gain wider applicability in exchange for increased language and integration complexity. In particular, both of these systems are implemented in the context of Turing-complete Prolog systems which requires programmer attention to ordering of code fragments and the avoiding of infinite loops. We have found that ASP represents a sweet-spot for PCG in the realm of declarative, solver-based generative approaches because the ability to rapidly try out alternative designs with minimal debugging time is so important in design problems

V. TUTORIAL EXAMPLE

To illustrate a simple but complete application of ASP to a toy PCG problem, we now describe our reimplementation of the "chromatic maze" generator described in Ashlock's "Automatic Generation of Game Elements via Evolution" [31]. The original system used a straightforward genetic algorithm

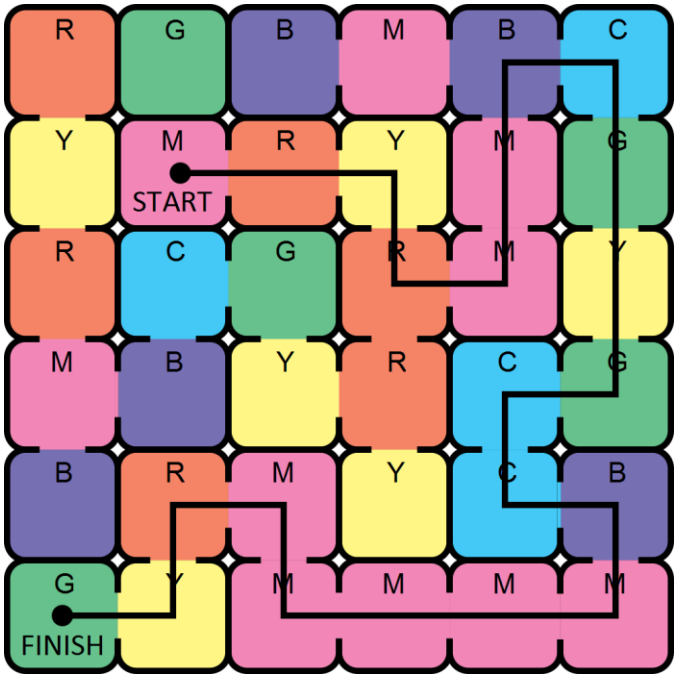

Fig 2. A chromatic maze created with our ASP-based generator. Valid moves consist of single steps on a red-yellow-green-cyan-blue-magenta color wheel (repeats allowed). The dark line represents a shortest path between the start and finish tiles.

with a fitness function measuring maze solution lengths calculated using a dynamic programming algorithm adapted to the unique mechanics of chromatic mazes. The paper additionally described a chess maze generator which we have also recreated but will not detail in this paper.

A. Chromatic Mazes

Chromatic mazes are a kind of puzzle using a square map consisting of colored tiles. Where other mazes might have explicit walls, passage between tiles in chromatic mazes is regulated by the adjacency of the tile colors on a color wheel. Red-yellow-yellow-red-magenta is a valid path, but red-blue is an illegal move (for the color wheel used in our generator). In addition to the colored grid, two tiles are marked for the start and finish (entrance and exit) of the maze. See Fig. 2 for an example chromatic maze with an overlaid solution.

The content generation task for chromatic mazes is to invent several unique, playable mazes with preferably long shortest-path distances between the start and finish. While path lengths are a very crude measure maze difficulty, they provide us with a reasonable example of how to work with numeric desirability metrics in PCG problems.

B. Modeling the Design Space

While modeling and interpretation are usually carried out in parallel, we will rearrange the development of our ASP-based generator to form a less chaotic story.

First, laying down a schema for logical terms that describe a chromatic maze, we want to generate terms shaped like the following:

cell(*Color*,*X*,*Y*), start(*X*,*Y*), finish(*X*,*Y*).

Creating the choice rules that would generate these kinds of terms requires some background definitions to specify the

range of values for the *Color* and *X*/*Y* variables. These ranges are easily asserted using these simple AnsProlog statements:

```
color(red; yellow; green; cyan; blue; magenta).
dim(1..6).
```

This snippet captures the valid range of our representation language using choice rules:

```
1 { cell(C,X,Y) :color(C) } 1 :- dim(X),dim(Y).
1 { start(X,Y) :dim(X) :dim(Y) } 1.
1 { finish(X,Y) :dim(X) :dim(Y) } 1.
```

The unfamiliar elements of syntax used here describe the scope of quantification for variables and put bounds on the number of generated facts. In this case, the colon operator and the numerals together ensure that exactly one cell fact is produced for each possible assignment of *X* and *Y* in the first rule (36 facts), and exactly one start and one finish fact are produced for the entire puzzle.

The last five lines of AnsProlog code above actually constitute a working generator for chromatic mazes! The design space it describes, however, needs some refinement as the finish tile may not be reachable from the start tile (the generative space contains some undesirable mazes). Expressing that a maze *must not* be impossible to complete is indeed a job for integrity constraints. The following constraint accomplishes this, but sets up the need for additional definitions:

```
:- not victory.
```

The next snippet grounds the idea of "victory" required by the above constraint using traditional logical rules. In it, we have elided the definition of a predicate for passage between tiles; however, this predicate is easy to define in terms of the color and cell facts with some arithmetic to model grid-adjacency.

```
player_at(0,X,Y) :- start(X,Y).
player_at(T,X,Y) :-
  player_at(T-1,SX,SY),
  passable(SX,SY,X,Y),
  0 {player_at(0..T-1,X,Y)} 0.
victory_at(T) :- player_at(T,X,Y), finish(X,Y).
victory :- victory_at(T).
```

In exchange for describing the mechanics of our mazes in AnsProlog, we gain a high level vocabulary for talking about properties of potential mazes. At this point, our answer set program captures the idea of provably-valid maze designs; all that is needed to upgrade our generator to one that produces desirable mazes (with long shortest-path lengths) is to add one more integrity constraint:

```
:- victory_at(T), T < 22.
```

While AnsProlog provides an additional construct (the **maximize** statement) that we could used to produce a best-possible maze with the introduction of just one more predicate, we have found that absolute optimization is rarely desirable for PCG problems. Instead, a good generator will produce a large space of artifacts with critical metrics guaranteed to fall in a good-enough range (as in the case for our chromatic maze generator) – *satisficing* [2].

C. Interpreting Artifacts

Thus far, we have treated the collection of facts describing a maze as if it were the maze itself. This is a productive mindset for modeling the design space, but human players will greatly appreciate a graphical display with literal, colored tiles. As mazes need not be loaded into any existing game engine, we were satisfied to create a program that produced colored ASCII-art in a terminal display.

The mapping from logical terms written as output from the ASP solver to pictures on the screen is quite straightforward, so we will not describe the details of the small Python program that accomplished this. Instead, we would like to share a particular change we made to the generator code to support more flexible interpretation.

During the development of a generator, often viewing artifacts as they will be experienced in-game is not enough to provide useful feedback in the design process. To get a better idea of the mazes we were generating, we wanted a visualization of the shortest-path distances in addition to the literal tile colors. Instead of producing a specialized debug-visualizer, we opted to generalize our visualizer to render arbitrary, colored ASCII-art tables using the following rules:

```
tile_color(X,Y,C) :- cell(C,X,Y).
tile_char(X,Y,s) :- start(X,Y).
tile_char(X,Y,f) :- finish(X,Y).
tile_char(X,Y, T #mod 10) :-
  T > 0,
  player_at(T,X,Y),
  not start(X,Y),
  not finish(X,Y).
```

Translating, the above logic describes a table display for chromatic mazes where s's and f's mark the start and finish and other tiles are represented with the last digit of their distance from the start (useful for debugging). These single characters are colored by the corresponding maze tile.

Using this approach, treating the tile_char and tile_color terms as the primary output of our generator, we were able to completely reuse our external visualization program in a reimplementation of the chess maze design space (with o's and t's for occupied and threatened tiles on the chess board). In actuality, our chromatic maze generator was produced by a series of small refinements to our chess maze generator: changing the annotation for tiles, changing the passage criterion between tiles, and updating the visualization logic, leaving the general calculation and bounding of shortest-paths and victory times unmodified during the evolution.

D. Comparison

We have captured the essence of Ashlock's evolutionary chromatic maze generator in an ASP-based generator. Our model encodes the very same design space implicitly defined by the intent of the evolutionary generator (though what counts as the actual generative space for an evolutionary generator is not entirely clear). Where shortest-path length was a black-box fitness function used in the original system (its implementation was not constrained by the surrounding system), it is a white-box modeled property of mazes in our system, subject to directly enforced constraints. In place of explicit

data structures and algorithms for computing solution lengths in the original system, our system defines the meaning of solutions in just a handful of AnsProlog lines, exploiting the search processes already embedded in the external solver for both analysis and generation.

Just as the development of the evolutionary generator prompted its author to perform a parameter study to understand the effects of varying population sizes and mutation rates, the design of our ASP-based generator prompted its own parameter study. With the (output) fitness value of the original generator under our direct control (as an input), we were able to directly locate mazes with solution lengths outside the range of those found by the original system. This allowed us to discover some previously unreached chess mazes, and lead to our formally proving the non-existence of chromatic mazes with a shortest-path length above a certain bound (the natural interpretation of an ASP solver's complete search algorithm returning no solutions).

Using a generation paradigm which foregrounds an iterative design process, we are obliged to read the fine details of individual generated mazes and think what would make them more desirable. Consider the run of magenta tiles in the bottom row of the generated maze in Fig. 2. Towards making our generated mazes more interesting, we might want to ensure that all of our mazes had shortest-paths without such obvious runs. While enforcing that no color is ever used adjacent to itself would be one way of doing this, it would eliminate mazes which used runs of a single color to sneakily lead the player further from their goal. Instead, what we want is a constraint on colors traversed on all possible shortest-paths, while still allowing runs on other paths. Such a constraint is expressible in exactly three more lines of AnsProlog and no modifications of existing rules, but would be non-trivial to encode in a new fitness function for the original evolutionary generator. The fact that feasible-infeasible two-population genetic algorithms are an active area of research is evidence of the subtlety required to blend constraint enforcement with the optimization of an existing numerical metric [32].

E. Metrics

While it is not possible to directly compare the following metrics with the original implementation given the published details, we provide them to the reader to convey the idea that ASP-based generator development is likely easier than one might suspect and that the generators produced (with no careful attention paid to runtime performance concerns) are not burdensomely slow.

Our chromatic maze generator, which reproduced in full in the appendix of this paper, consists of 49 source lines of idiomatic AnsProlog (half of which are identical to those in the chess maze generator). Running this program through Clingo and our Python visualization program on a single core of an Intel Core2 Quad at 2.66Gz, we can generate and render a single 6-by-6 chromatic maze with a solution length between 20 and 35 steps in 250 milliseconds. To understand the effects of start-up costs, we asked the generator to generate and save 10,000 unique, desirable mazes from the same design space. The operation took one second, using an average 100 microseconds per maze. Generating a globally optimal 6-by-6 maze (requiring 35 steps to solve) took 2.5 seconds.

Towards matching the largest scale of mazes presented in the original paper, we modified our maze generator to produce a sequence of increasingly longer mazes at the 21-by-21 scale (using the **maximize** statement instead of the simple threshold shown previously). While Ashlock demonstrated a 292-step maze at this scale (requiring an unknown amount of computation), we can only report the generation of a 114-step maze after two hours of computation (with longer mazes requiring increasingly more time to find).

While these performance numbers are highly sensitive to variable parameters such as the size of the maze, a more important (though difficult to accurately measure and interpret) metric is the human programming effort required to produce the model of the design space. The first author, an experienced AnsProlog programmer, developed generators for knight-only chess mazes, chromatic mazes and a toroidal variant on chromatic mazes, along with the visualization logic, without prior planning, in less than four hours.

While this example permits a direct comparison between an ASP-based generator and an equivalent evolutionary generator, the problem of chromatic maze generation should not be interpreted as a benchmark problem for PCG. In particular, as defined, chromatic maze generation is not actually a design problem. Instead, when the design space is locked down, the problem becomes an engineering problem for which numerical results comparing two implementations that were not optimized for runtime performance would be misleading.

Attempts to characterize the runtime performance that should be expected from common answer set solvers have yielded results very similar to those seen for SAT solvers [33]. That is, while solvers generally employ algorithms with worst-case exponential complexity (in terms of a program's grounded size), solvers will terminate very quickly on a wide range of problem instances. Only when random programs have a mix of atoms and rules that approaches a critical ratio (reminiscent of the "phase transition" for SAT instances) does a solver actually encounter exponential blow-up in its search process. On ASP terms, the "hardest" problems appear to be those where exactly one answer set from an extremely large space of possibilities is the only solution. Anecdotally, we have found realistic PCG problems (large chromatic maze generation not included) such as those surveyed in the next section to fall on the "easy" size of the phase transition. That is, their constraints are relatively simple to satisfy (admitting an estimated number of valid answer sets in the quadrillions for the case of *Variations Forever*), but they way in which they are satisfied leads to interesting game content.

With this tutorial example, we hope to show the reader how new ASP-based generators can be created at a code level, emphasizing the minimum of design commitments made during development. Practical applications of ASP to PCG will likely involve artifacts with far richer structure (and smaller numerical constants) that are consumed in the context of games with much more complex mechanics than maze traversal.

VI. Case Studies

In this section, we describe existing applications of ASP to content generation problems in the domains of real-time strategy game maps, arcade game mechanics, musical composi-

tions, and simple narratives. Of known ASP applications, we judge these to be the most relevant to PCG, with two of the four actually producing content that is consumed in gameplay. While ASP has been widely applied, it is only these systems which have employed the technology in a directly generative manner (as opposed to general problem solving or satisfiability testing). In describing them here, we hope to inspire wider adoption of this approach to content generator design.

A. DIORAMA

DIORAMA[3] is an open source, comprehensive map generator for the real-time strategy *Warzone 2100* (Pumpkin Studios 1999) which generates natural-looking, detail-decorated terrain maps with desirable gameplay properties. Internally, Diorama uses ASP to solve two gameplay-critical subproblems in map generation.

The cliff structure of the terrain in a *Warzone* map has a strong impact on gameplay by blocking land-vehicle passage between tiles with sufficiently different height values. In the first phase of the map generation process, amongst other details described later, a coarse grid of cells is assigned numerical height values with terms like cellLevel(*X*,*Y*,*Height*) (analogous to the assignment of colors to tiles in chromatic mazes). From cellLevel facts, passage between tiles can be derived using traditional logical rules (analogous to the traversability of chromatic mazes). Additional user-settable constraints enforce the presence of interesting geographical features: undulating plains, smooth sea beds, raised or sunken player bases, and a prescribed number of unreachable mountain tops. Logical rules for describing undulation and other modeled properties, of course, are also included in the AnsProlog program.

The problem of placing player base locations and oil wells (drivers of the economy in *Warzone*) is tightly coupled with the terrain generation problem; cliffs provide a natural, indestructible defense against ground-based attacks. Accordingly, DIORAMA combines these concerns into a single design space model, effectively solving for terrain, base, and well placement all at the same time. The baseLocation(*Player*,*X*,*Y*) and oilLocation(*Well*,*X*,*Y*) predicates complete the minimal structure of the generated artifacts at this stage.

To express a preference amongst the myriad possibilities that conform to terrain design rules, the AnsProlog program declares that the solver should first absolutely maximize the distance between player bases, and then, amongst solutions with that maximal base distance, find a placement of oil wells that maximizes the minimum inter-well and well-base distances. Additional constraints optionally enforcing partial cliff-based defensibility of bases and wells are also active in this process.

We learned that ASP actually replaced a genetic algorithm solution for this search-intensive phase in a previous version of the generator [34]. Without an ASP solver, such multi-layer global optimization would be difficult to express with a procedure that had separate generate and test phases. Even dropping global optimization in favor of enforcing a fixed lower bound on base and well distances would have been difficult otherwise.

[3] DIORAMA documentation and source: http://warzone2100.org.uk/

Though DIORAMA used a fixed priority scheme to layer different levels of preferences at this stage, a system of numerical weights could have (but was not) used to express trade-offs between preferences at the same level, e.g. that one point of resource distance unfairness is a safe trade of two points of cliff defensibility unfairness. In this case, the AnsProlog program would have asked the solver to simply enumerate maps that maximize the sum of the trade values.

Having committed to bases and oil wells on a naked height map with known traversability, DIORAMA performs several non-ASP passes to improve the aesthetics of the final map. To break the unnaturally straight lines of the original cell grid, the map is warped in a post-processing phase and cell borders are smoothed. A plausibly-designed road network is overlaid which visually guides players from their bases to oil wells and randomly placed abandoned towns. These phases add visual flair and cannot break the map's gameplay, so they proceed in a non-backtracking fashion, enriching the map in-place.

Generating a smoothed version of the abstract terrain is an example of a subproblem in terrain generation for which ASP is not particularly applicable. Instead of selecting artifacts with particular properties from a vast-but-finite space, continuous smoothing is a process more easily described imperatively in a general purpose programming language. That is, while geographical features are modeled properties at the (coarse) cell level, DIORAMA leaves the aesthetics of (fine) tile level details unmodeled and trusts the imperative passes to cover them in a feed-forward manner.

In a gameplay mode of *Warzone* that allows players to start a match with generous bases pre-built for them, maps may include a customized layout of essential buildings and fortifications. DIORAMA has an option that will cause an ASP-based base layout phase to be injected into the generation process after the terrain has been warped and smoothed. At a high level, location(*Building*,*X*,*Y*) facts are generated using choice rules for the origin point of each building, from which blocked tiles are deduced and the number of lined-up buildings is computed. Overlapping buildings are easily rejected with an integrity constraint. The size of a boundary zone around blocked areas is calculated and globally maximized, secondarily maximizing the modeled tidiness property of the arrangement. The resulting layouts have an organized-looking grid layout, fit to locally warped terrain features while ensuring units can still navigate around the base.

In order to perform terrain-adapted base layout, this phase needs access to terrain details already committed in the previous phase. DIORAMA dynamically assembles a specialized generator for the situation by concatenating AnsProlog fragments with facts describing the committed world details. Such dynamic construction of design space models is very common, and it represents a kind of adaptability to design problems that ASP provides that goes far beyond parameterization via numeric parameters (which are used often enough, e.g. in specifying the number of players for which a map should be designed).

B. *In this system, ASP was used to encapsulate two search-intensive subproblems as feed-forward generators in the context of a larger, multi-paradigm generator. Overall, DIORAMA's generative procedure is a non-backtracking*

pipeline of generate-only components; the test procedure, if any, is human inspection of the final maps. Even though map design does not involve a finite space (as terrain heights have a continuous domain), ASP was still able to solve the most difficult subproblems, leaving the other phases of generation free from any hand-written backtracking or generate-and-test search.Variations Forever

In our recent project, *Variations Forever* (VF) [16], we generated code-like content that describes the game mechanics for simple arcade games. An AnsProlog program embedded within our Flash game captured a configurable design space of mini-game rulesets that was consulted each time the player completed a challenge.

VF demonstrated the use of ASP-based generators for online PCG. The design space was parameterized, in a sense, by Boolean flags such as tech**(**obstacles**)** that could be used to toggle the presence of various mechanics in mini-games (the optional presence of fixed obstacle tiles in this example). In a more developed meta-game, these flags would be selectively unlocked through player exploration, making design-space sculpting a core mechanic. By communicating with an ASP solver over the network, our generator delivered content to the locked-down environment of a web browser at runtime.

In contrast to many of the uniform structures used in DIORAMA (assigning numeric heights to a regular grid of cells), VF focused on more complex relational structures describing the interaction of a smaller number of objects. In the mini-games, colored squares representing agents moved about, interacting with other agents and the environment. A string of facts like the one below might (partially) define a ruleset. Other facts parameterize an external obstacle placement algorithm and influence the visual presentation of the mini-game (such as background art selection and camera control policy).

```
player_agent(green),
movement_model(green,pacman),
agent_collide_effect(red,green,kill),
obstacle_collide_effect(green,bounce),
goal(kill_all(red)).
```

With a larger number of term types used in VF (in comparison to DIORAMA), interpretation of answer sets is more complex than setting values in data structures: collision effect rules control which event handlers are installed in the game engine's physics simulation; movement models affect both player keyboard controls and per-frame physics; and the game's stated goal becomes a continuously monitored condition. Despite the complexity of interpretation, many interesting properties of the resulting mini-games were describable using AnsProlog rules.

The rich, relational representation allowed interesting emergences such as a seemingly-unbeatable game which required the player to achieve the stated goal by using additional agents as indirect pushing tools. The other side of rich representations is undesirable emergences, such as games with the goal of escape combined with an encircling wall pattern (impossible to complete). In either case, a small set of rules described the essential property of the emergence. Then, using an integrity constraint, the properties were either required (effectively producing a specialized generator for that flavor of artifact) or rejected.

The ruleset generator in VF includes a winnable_via**(***Method***)** predicate which describes templates for games which can be beaten by a given method (e.g escape, direct kill, indirect kill, etc.). Using such a rule to define recognizable sub-genres within VF's space eased understanding of unfamiliar mini-games at design time and enabled the generation of hints at play time. That is, the estimated mode of winnability was not required by the game engine, but it provided optional feedback to the designer and player about which patterns that game ruleset contained.

The tight, design-time loop of modifying VF's generator with property definitions after sampling only a few artifacts is a common (even enjoyable) experience. The temporary employment of certain integrity constraints to zoom in on a subspace of interest (called *speculative assumptions* in LUDOCORE vocabulary [15] [15]), allows the focus on mini-games which highlight or stress some aspect of a game engine (assisting in debugging of the larger system) without re-designing the generator. Rapid iteration on our ruleset design space, in this way, was important for identifying and addressing potential failure cases of our game ruleset generator.

C. ANTON

The state-of-the-art automatic music composition system, ANTON [14], uses ASP to produce detailed melodic, harmonic, and rhythmic musical compositions informed by a seamless blend of local and global composition knowledge. Although the musical content is not consumed in the context of any game (instead in the form printed music notation, rendered audio tracks, and even real-time performance), the application provides examples of potential new directions for PCG systems.

Until now, we have focused on the use of modeled properties for the purposes of use in integrity constraints, that is, requiring or rejecting properties. The Palestrina rules of composition (a codification of renaissance counterpoint in western tonal music theory) in ANTON serve a dual purpose: the system can use the same rules to diagnose and informatively report flaws in an externally provided composition. It might report the message "middle note of triad doubled" and identify a particular part and time in the composition, or "invalid minor harmonic combination" citing another location. Different composition styles carry different error pattern definitions. As a body of computational music theory, it is natural to expect ANTON's knowledge to be used in both analysis and synthesis.

The different modes of operation in ANTON are supported by a program builder that dynamically assembles the appropriate AnsProlog fragments to craft a design space appropriate to the problem at hand. In a composition mode, with a certain style and other configuration set, the design space contains musical scores (represented with chosenNote**(***Part***,***Time***,***PitchNumber***)** terms) that strictly conform to the assembled rules. In diagnosis mode, a logical encoding of a human-created composition is combined with style rules to produce a design space of *critiques* (including error**(***Part***,***Time***,***Reason***)** messages), as opposed to musical passages.

Automated critique of human-design artifacts can, of course, help clean up these artifacts, but it can also be used to understand and debug a design space (even suggesting particular constraints to relax in the iterative design of a generator). Producing intelligent answers to a query such as "*Why* won't you generate artifacts like *this*?" is not an activity supported by any generators for game content, but perhaps it should be in the future. Answers to such queries can even point out inconsistencies in the background theory used by the generator (for example, it's possible that the Palestrina rules contain contradictions that are not obvious to human examination).

The ability for the composition mode to accept partial human-created pieces allows the system to serve a number of purposes beyond tabula-rasa generation: supplying fragments allows the constrained generation of music that ends with a certain pattern or embeds a certain motif; supplying one part and not others allows solving for a harmonization consistent with a given melody; and specifying only a chord progression allows for natural melodic improvisation. By incrementally committing to (or forbidding) details suggested by the system, the user can carry out a mixed-initiative interaction with the system. Nearly all ASP-based generators that employ dynamic program construction will inherit this give-and-take capability by default.

Though the connection is subtle, there is a similarity between the music composition design task and the task of designing platformer levels (rhythm has been identified as part of this link [35]). ANTON composes music by selecting a series of local moves a part should take: step up, leap up, rest, repeat, etc. The trajectory through the space of absolute pitches and times is derived as a side effect of these local moves. Rhythmic and melodic composition rules are often written in terms of the global trajectory and have a complex relationship to the local moves. Similarly, Launchpad, a platformer level generator, works by selecting a sequence from a set of local actions the player should take: move, jump, and wait [36]. The player's trajectory is made concrete by generating geometry which must fit rhythmic density and style constraints. The line critic in Launchpad can be seen to be functioning as a melodic composition rule, with ANTON's harmonic composition rules speaking to the gameplay of as-yet-unconsidered platformer levels with interacting, parallel tracks.

D. RoleModel

Our final example using of ASP for content generation is our work-in-progress RoleModel story generator [37]. This system's narrative model includes characters (as a collection of static traits and dynamic attributes), events (when a subject acts on an object), and contextual details (a character's sentiment for his actions or notes that modify the effectiveness of an action). Roles, for which the system is named, are perceived archetypes of characters which, instead of being inherent properties of their person, are a judgment made based upon their actions in the story world. That is, they are an emergent result of how the story unfolds.

Traditional story generators often adopt a single generation paradigm such as a story grammar or world simulation (character modeling) approach. A declarative model of narrative generation has been successfully demonstrated to allow these two particular paradigms to be blended; however, the example system uses a specialized search algorithm to interleave world simulation with story grammar traversal, generating story fragments in a strict temporal order [38].

The strength of the story generation approach used in RoleModel derives, in part, from the declarative knowledge representation abilities of logic programming, and also from the use of sophisticated ASP solvers to implement the search process used in generation. The system is capable of generating both character profiles and the actions those characters take in a such a way that characters act in accordance with their traits, as well as fitting any author-specified constraints on world state and perceived roles (perhaps that an Alice character must have killed her friend Bob without appearing to be an aggressor).

RoleModel brings to light a useful metaphor that applies to other PCG problems as well. Suppose you had the proverbial million monkeys, each typing away at typewriters. If a Shakespeare-level work were to be produced, how would you recognize it? Choice rules stand in for the monkeys, producing a default search space consisting of the right primitives, but likely lacking appropriate global structure. The traditional logic rules define the patterns to watch for in the typewritten results, and the integrity constraints represent an editorial policy for rejecting monkey business in terms of the modeled properties. ASP allows the designer of a content generator to *think* in terms of a generate-and-test process while enjoying the performance of the adaptive search processes in the off-the-shelf solver at runtime.

Finally, in many domains, story generation included, invention of interesting modeled properties can not only lead to more interesting generated outputs, but can contribute to our shared human knowledge in that domain. The concept of roles in RoleModel would be difficult to experiment with in a generation paradigm for which emergent roles could not be directly manipulated.

VII. CONCLUSION

This approach has been rooted in the recognition that PCG is really concerned with two problems: the reliable generation of desirable content artifacts from a design space, and the design of content generation processes which are *fast*, produce *fancy* artifacts, and do so in a *flexible* way (in support of iterative design). Existing applications demonstrate that answer set programming is a proven method for quickly (in terms of both running time and designer effort) creating trustable, search-intensive solutions to rich content generation problems in a way that minimizes commitment to procedural details. The coupling of a declarative specification of design spaces with powerful solvers in ASP makes this possible.

In this paper, we have laid out a widely applicable framework for creating content generators. Between here and what might truly be called a *generative space programming* paradigm, what is lacking is a natural, integrated way of leveraging procedural knowledge where it is applicable. Future PCG research should investigate how declarative, solver-based generators can be connected with other generative procedures in a way that preserves the desirable properties of the parts.

APPENDIX

Below is the *complete* source for our ASP-based chromatic maze generator (tested in Clingo 2.0.5). To recreate our generation of the globally optimal 6-by-6 maze, invoke "`clingo appendix.ans -c min_solution=35`". The logical terms then seen output by the generator are what would be fed into our Python program for visualizing tile maps.

Clingo is available for free download from the Potassco website (http://potassco.sourceforge.net/). Experimental extensions are also available (from http://potassco.sourceforge.net/labs.html) including Xorro, a tool for sampling answer sets with near-uniform probability, and Metasp, a metaprogramming layer for expressing complex optimization criteria including Pareto efficiency.

```
%%%% A chromatic maze generator
#const t_max = 35.
#const min_solution = 20.
#const max_solution = 35.
#const size = 6.

%% A range of space and time

dim(0..size-1).

time(0..t_max).

%% Neighbor adjacency on grid

adjacent(X,Y,X+1,Y) :- dim(X), dim(Y).
adjacent(X,Y,X-1,Y) :- dim(X), dim(Y).
adjacent(X,Y,X,Y+1) :- dim(X), dim(Y).
adjacent(X,Y,X,Y-1) :- dim(X), dim(Y).

%% Cycling terminal-displayable colors

color(red;yellow;green;cyan;blue;magenta).

next(red,yellow).
next(yellow,green).
next(green,cyan).
next(cyan,blue).
next(blue,magenta).
next(magenta,red).

%% Allowable color transitions

ok(C,C) :- color(C).
ok(C1,C2) :- next(C1,C2).
ok(C1,C2) :- next(C2,C1).

%% Allowable steps

passable(SX,SY,X,Y) :-
  adjacent(SX,SY,X,Y),
  cell(C1,SX,SY),
  cell(C2,X,Y),
  ok(C1,C2).

%% Description of chromatic mazes

1 { cell(C,X,Y) :color(C) } 1 :- dim(X), dim(Y).

1 { start(X,Y) :dim(X) :dim(Y) } 1.

1 { finish(X,Y) :dim(X) :dim(Y) } 1.

%% A flood-fill style player exploration

player_at(0,X,Y) :- start(X,Y).
player_at(T,X,Y) :-
  time(T),
  player_at(T-1,SX,SY),
  passable(SX,SY,X,Y),
  0 { player_at(0..T-1,X,Y) } 0.

%% The time of earliest completion

victory_at(T) :- finish(X,Y), player_at(T,X,Y).

%% That completion happened at all

victory :- victory_at(T).

%% Requirements on all generated puzzles:

:- not victory.

:- victory_at(T), T < min_solution.

:- victory_at(T), max_solution < T.

%%%% Visualization support logic

tile_grid(size,size).

tile_char(X,Y, T #mod 10) :-
  T > 0,
  player_at(T,X,Y),
  not start(X,Y),
  not finish(X,Y).

tile_char(X,Y,s) :- start(X,Y).

tile_char(X,Y,f) :- finish(X,Y).

tile_color(X,Y,C) :- cell(C,X,Y).
```

ACKNOWLEDGMENT

The authors would like to thank Martin Brain for being an insider in the ASP community, sharing tools and techniques, and clarifying terminology. They would further like to thank Mr. Brain and his fellow hacker, Florian Schanda, for creating the centerpiece of the ASP-for-PCG approach with their leisure-time project, DIORAMA, created for the love of the game and given freely to the fans of it.

REFERENCES

[1] John C Thomas and John M. Carroll, "The psychological study of design," *Design Studies*, vol. 1, no. 1, pp. 5-11, July 1979.

[2] Herbert A. Simon, *The sciences of the artificial*, 1st ed. Cambridge: MIT Press, 1969.

[3] Julian Togelius, Georgios Yannakakis, Kenneth Stanley, and Cameron Browne, "Search-based procedural content generation," in *Applications of Evolutionary Computation*, vol. 6024, 2010, pp. 141-150.

[4] Adam M. Smith and Michael Mateas, "Tableau Machine: a creative alien presence," in *AAAI Spring Symposium on Creative Intelligent Systems*, 2008, pp. 82-89.

[5] Penousal Machado and Amilcar Cardoso, "All the truth about NEvAr," *Applied Intelligence*, vol. 16, no. 2, pp. 101-118, 2002.

[6] Scott Draves, "The Electric Sheep screen-saver: a case study in aesthetic evolution," in *Applications of Evolutionary Computation*, 2005.

[7] Fred P. Brooks, "No silver bullet -- essence and accident in software engineering," in *IFIP Tenth World Computing Conference*, 1986, pp. 1069-1076.

[8] Bruce G. Buchanan, "Creativity at the metalevel: AIII-2000 presidential address," *AI Magazine*, vol. 22, no. 3, 2001.

[9] Ben Moseley and Peter Marks. (2006, February) Funtional Reactive

Programming. [Online].
Available: http://web.mac.com/ben_moseley/frp/paper-v1_01.pdf

[10] Tim Tutenel, Ruben M. Smelik, Rafael Bidarra, and Klass Jan de Kraker, "A semantic scene description language for procedural layout solving problems," in *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2010, pp. 95-100.

[11] Ruben Smelik, Tim Tutenel, Klass Jan de Kraker, and Rafael Bidarra, "Integrating procedural generation and manual editing of virtual worlds," in *PCGames '10 Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, 2010.

[12] Gillian Smith, Jim Whitehead, and Michael Mateas, "Tanagra: a mixed-initiative level design tool," in *Proceedings of the 2010 International Conference on the Foundations of Digital Games*, 2010, pp. 209-216.

[13] Joris Dormans, "Adventures in level design: generating missions and spaces for action adventure games," in *PCGames '10: Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, 2010.

[14] Georg Boenn, Martin Brain, Marina De Voss, and John Fitch, "Anton: answer set programming in the service of music," in *Proceedings of the Twelfth International Workshop on Non-Monotonic Reasoning*, 2008, pp. 85-93.

[15] Adam M. Smith, Mark J. Nelson, and Michael Mateas, "Ludocore: A Logical Game Engine for Modeling Videogames," in *IEEE Symposium on Computational Intelligence in Games (CIG 2010)*, 2010, pp. 91-98.

[16] Adam M. Smith and Michael Mateas, "Variations Forever: Flexibly Generating Rulesets from a Sculptable Design Space of Mini-Game," in *IEEE Symposium on Computational Intelligence and Games (CIG 2010)*, 2010, pp. 273-280.

[17] Chitta Baral, *Knowledge representation, reasoning and declarative problem solving*.: Cambridge University Press, 2003.

[18] Fangzhen Lin and Yuting Zhao, "ASSAT: computing answer sets of a logic program by SAT solvers," *Artificial Intelligence*, vol. 157, no. 1-2, pp. 115-137, August 2004.

[19] A. Bertoni, G. Grossi, A. Provetti, V. Kreinovick, and L. Tari, "The prospect for answer sets computation by a genetic model," in *Proceedings of the AAAI Spring Symposium on Answer Set Programming*, 2000.

[20] Pascal Nicolas, Frédéric Saubion, and Igor Stéphan, "Answer set programming by ant colony optimization," in *Proceedings of the 8th European Conference on Artificial Intelligence*, 2002, pp. 186-197.

[21] Martin Davis and Hillary Putnam, "A Computing Procedure for Quantification Theory," *Journal of the ACM (JACM)*, vol. 7, no. 3, pp. 201-215, July 1960.

[22] Stuart Russel and Peter Norvig, *Artificial intelligence: a modern approach*, 2nd ed.: Prentice Hall, 2002.

[23] Martin Gebser, Benjamin Kaufmann, Andre Neuman, and Torsten Schaub, "clasp: a conflict-driven answer set solver," in *Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning*, 2007, pp. 260-265.

[24] Martin Gebser et al., "Engineering an incremental ASP solver," *Logic Programming*, pp. 190-205, 2008.

[25] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub, "The conflict-driven answer set solver clasp: progress report," in *Proceedings of the Tenth International Conference on Logic Programming and Nonmonotonic Reasoning*, 2009, pp. 509-514.

[26] Martin Brain, Owen Cliffe, and Marina De Voss, "A pragmatic programmer's guide to answer set programming," in *Software Engineering for Answer Set Programming (SEA09)*, Potstam, Germany, 2009.

[27] Martin Gebser, Max Ostrowski, and Torsten Schaub, "Constraint answer set solving," in *Proceedings of the 25th International Conference on Logic Programming*, 2009.

[28] Nicola Leone, "Exploiting ASP in real-world applications: main strengths and challenges," in *Proceedings of the 10th International Conference on Logic Programming*, 2009, pp. 628-630.

[29] Henning Christiansen and Veronica Dahl, "HYPROLOG: a new logic programming language with assumptions and abduction," in *2005 International Conference on Logic Programming*, 2005, pp. 159-173.

[30] Christian Holzbaur, "OEFAI clp(q,r) manual rev. 1.3.2," Austrian Research Institute for Artificial Intelligence, TR-95-09, 1995.

[31] Daniel Ashlock, "Automatic generation of game elements via evolution," in *IEEE Conference on Computational Intelligence and Games (CIG2010)*, 2010, pp. 289-296.

[32] Steven O. Kimbrough, Gary J. Koehler, Ming Lu, and David H. Wood, "On a feasible-infeasible two-population (FI-2Pop) genetic algorithm for constrained optimization: distance tracing and no free lunch," *European Journal of Operational Research*, vol. 190, no. 2, pp. 310-327, October 2008.

[33] Yuting Zhao and Fangzhen Lin, "Answer set programming phase transition a study on randomly generated programs," Hong Kong University of Science and Technology, KHUST-CS03-04, 2003.

[34] Martin Brain. "ASPs Galore!" Personal E-mail (Jul. 24, 2010).

[35] Kate Compton and Michael Mateas, "Procedural level design for platformer games," in *AAAI Conference on Artificial Intelligence in Interactive Digital Entertainment (AIIDE2006)*, 2006, pp. 109-111.

[36] Gillian Smith, Mike Treanor, Jim Whitehead, and Michael Mateas, "Rhythm-based level generation for 2D platformers," in *Proceedings of the 2009 International Conference on the Foundations of Digital Games (FDG2009)*, 2009.

[37] Sherol Chen, Adam M. Smith, Arnav Jhala, Noah Wardrip-Fruin, and Michael Mateas, "RoleModel, towards a formal model of dramatic roles for story generation," in *3rd Workshop on Intelligent Narrative Technologies (INT3)*, 2010.

[38] R. Raymond Lang, "A declarative model for simple narratives," in *AAAI Spring Symposium on Narrative Intelligence*, 1999, pp. 134-141.

Adam M. Smith is a Ph. D. candidate at UC Santa Cruz (Santa Cruz, California, USA). He received a B.S. in computer science from UC Santa Cruz in 2005.

He has had internships at Terracom Communications (Rwanda), NASA Ames Research Center, Los Alamos National Lab, and Google. His research is centered on computational creativity in game design but also includes contributions in computer vision and statistical machine learning.

Mr. Smith is a student member of the IEEE, ACM, AAAI and IGDA, and an artist in the TOPLAP algorithmic music livecoding collective.

Michael Mateas is an associate profession the computer science department at UC Santa Cruz, where he holds the MacArthur Endowed Chair. He received his Ph. D. in computer science from Carnegie Mellon University. His research in AI-based art and entertainment combines science, engineering, and design to push the frontiers of interactive entertainment.

He founded and co-directs the Expressive Intelligence Studio at UC Santa Cruz, which has ongoing projects in autonomous characters, interactive storytelling, game design support systems, procedural content generation, automated game design, and learning AI from datamining gameplay traces.

With Andrew Stern, Prof. Mateas released Façade, the world's first AI-based interactive drama. Façade has received significant attention, including top honors at the Slamdance independent game festival.