Inferring Design Constraints From Game Ruleset Analysis

Michael Cook and Simon Colton
The Metamakers Institute
Falmouth University
www.gamesbyangelina.org

Azalea Raad
MPI-SWS
Kaiserslautern, Germany
www.soundandcomplete.org

***Abstract*—Designing game rulesets is an important part of automated game design, and often serves as a foundation for all other parts of the game, from levels to visuals. Popular ways of understanding game rulesets include using AI agents to play the game, which can be unreliable and computationally expensive, or restricting the design space to a set of known good game concepts, which can limit innovation and creativity. In this paper we detail how ANGELINA, an automated game designer, uses an abductive analysis of game rulesets to rapidly cull its design space. We show how abduction can be used to provide an understanding of possible paths through a ruleset, reduce unplayable or undesirable rulesets without testing, and can also help discover dynamic heuristics for a game that can guide subsequent tasks like level design.**

I. Introduction

Automated game design is a growing field of games research that builds on existing work in generative software, computational creativity, and general game playing. It centers around the development of software that can design complete games, either autonomously or co-creatively with the involvement of other people. Designing a complete game might include performing tasks such as inventing game mechanics; designing levels or game worlds, composing music or designing sound effects; creating visual assets or an artistic direction; or many other tasks depending on the kind of game being designed, and the kind of system being developed.

Many automated game design systems focus on rules-driven, objective-based games, where the player is presented with an objective to complete while navigating specific challenges [1]. Classic arcade games like *Space Invaders*, *Pac-Man* or *Frogger* fit into this model. These games require a clear set of rules, with win and loss conditions, and normally also require levels or starting layouts that pose specific challenge scenarios for the player. Although these rulesets are usually deterministic, they often have complex dynamics (in the sense of [2]) and their own unique set of requirements for what their levels should contain or challenge the player to do. Combined with the multiplicatively large state spaces for game rulesets and level designs, this makes automatically evaluating game designs very difficult.

Common solutions to this problem include the use of *playouts*, where an AI agent plays a game and its playtrace data is analysed to measure certain things about a game – for example, the complexity of a puzzle solution, or what score an averagely-performing agent could achieve before losing. However, playouts are computationally expensive, especially if they are used to play game designs that have obvious deficiencies in their design. Other systems use simple analytical techniques to assess game designs: for example, searching for simple rule patterns that imply an unplayable game, such as directly contradictory rules. Although this approach is effective, its use is currently restricted to very basic inferences.

In this paper we describe how we use abductive reasoning to analyse generated game rulesets, and how the resulting analysis helps us remove larger quantities of bad rulesets, without resorting to more prescriptive generation techniques. We also show that this abductive analysis of rulesets can reveal *intuitions* about a game design, which can be used to constrain level generation. We also posit how this abductive analysis might be used to incorporate subjective design constraints into automated game design.

The remainder of this paper is organised as follows: in *Background* we discuss existing approaches to solving the ruleset generation problem, and introduce the current version of the ANGELINA automated game design system; in *Static Ruleset Analysis* we describe the process of analysing game rulesets to extract possible routes through the game's logic; in *Level Constraint Inference* we extend this work to show how it can derive information that can be used to constrain the level design process; in *Design Space Analysis* we provide experimental results showing the impact these techniques have on the design space; in *Discussion and Future Work* we comment on the applicability of this technique to other kinds of game, and future work in the area; finally, in *Conclusions*, we summarise our work.

II. Background – Related Work

Automated game designers are tasked with exploring a vast state space of possible game designs. Each generative subtask – designing levels, designing game rules, designing objectives and so forth – is itself a vast possibility space, and when considered together these spaces exponentially expand and become tightly coupled to one another – a small change to a ruleset might render an entire set of levels unplayable or trivial. There are three broad approaches in existing automated game design work towards solving this problem.

The first approach is to leverage existing knowledge about good game design to restrict the search problem. Often this consists of building a smaller design space populated by components of existing games, and combining them in ways that preserve playability. Nelson and Mateas' work in [3] uses a 'stock set of concrete game mechanics' to build minigames. Meanwhile, The Game-O-Matic uses 'micro-rhetorics' which describe game components, and combines them using recipes which ensure the safety of the resulting game design [4]. This results in good-quality games that are reliably playable. This approach has some downsides: it is harder to discover new or surprising game designs this way, for example, as it relies on remixing to succeed. The Game-O-Matic also eschews complex level design in exchange for a simpler arrangement of shapes on the screen, which slightly reduces the complexity of the design task. Nevertheless, the Game-O-Matic demonstrates the strength of the micro-rhetoric approach, and is one of the most accomplished automated game design systems to date. We employed a similar approach in ANGELINA$_5$, which used a catalogue of known game mechanics to ensure it always produced playable rulesets.

Another common approach is to use genre-specific knowledge to focus a system on one particular type of game. In some sense, every automated game design system uses this technique, since no system to date has been truly general and cross-genre. However, systems like Data Adventures [5] or Puzzle Dice [6] focus on a very specific game structure, and as such are able to work in a reduced design space, by knowing in advance what mechanics a game might have, or being able to define objective functions for search algorithms in advance. We employed this approach in ANGELINA$_3$ to narrow our design focus on Metroidvania-style games, enabling us to define more specific objective functions for the design of levels in games the system made.

Finally, many automated game design systems use *playouts* to evaluate the games they design. Playouts involve using AI agents to play the game as a player might, and analysing the results of those playouts to infer properties about the game. Togelius and Schmidhuber use playouts to assess how hard a game is to learn, and from that infer how 'fun' the game might be as a result [7], while Khalifa et al use playouts to assess ruleset coverage and score achievement [8]. We have employed playouts in all versions of ANGELINA – ANGELINA$_1$, for example, used multiple playouts per game with different settings for each playout agent, to try and simulate more and less risk averse players.

III. Background – ANGELINA

ANGELINA is an automated game design project that has been in development in some form or another since 2011. The latest version of the software is developed in Unity, and currently focuses on two-dimensional, grid-based, turn-based game design. The software is designed to be run as an always-on system that is continuously developing games or discovering design knowledge. We call this *continuous automated game design* [9]. ANGELINA moves between distinct design tasks, like creating game logic or designing levels, and updates a central project file for each game it works on as tasks are completed. In this paper we focus primarily on the design of rules and levels in ANGELINA.

ANGELINA describes games using a domain specific language similar in structure to VGDL [10] or PuzzleScript [11]. Game descriptions are written in JSON, and broken down into several sections that describe different parts of the game. Figure 1 shows a template for a game, with some parts cut for length, which we will describe in stages below.

The first few lines of a game description are called the *preamble* and contain basic information about the game, including its name, the filename it is saved under, as well as basic artistic settings like the user interface colour scheme and background music. Most of these settings are optional but give the system scope to express a tone or theme through aesthetic choices. After the preamble is the *pieces* list, which defines a set of types of game object used in this game. Each piece can be instantiated one or more times in a game, and may be animate or inanimate, but these details are defined later in the game description. A piece's definition includes a name, by which it is referred to in the rules, as well as information on how to draw the object on-screen.

Most importantly for this paper, the *rules* section describes the game's core logic. It contains two lists: one containing the main game logic, which is executed at every game step; and one set of end conditions which describe ways the game can end, as well as the nature of each ending (such as a win, or a loss). A rule is comprised of two parts: a *trigger*, which is a condition that decides whether or not the rule activates; and one or more *events* which are executed in order when the trigger condition is met. We use a special syntax to allow events to reference objects 'in scope' of the rule: `$n` references the nth object mentioned in the rule so far. For example, in the rule:

```
"trigger": "OVERLAP enemy player",
"events": ["DESTROY $2"]
```

The term `$2` refers to the second object in the trigger, `player`. Thus, this rule says that when an `enemy` piece overlaps a `player` piece, the `player` piece is destroyed.

Finally, the *levels* section defines the spaces the player explores as part of the game, which is empty at the point of ruleset design. Currently, ANGELINA designs games which present a traditional sequence of levels to the player one after another. Levels are defined here in the order they are played in, and can either be loaded from another file or included as raw data. In the example in Figure 1, the level shown is written as raw data, with each number in the grid corresponding to an index into the pieces list defined earlier in the game description (this list being 1-indexed as opposed to the more traditional 0-indexing, since we use 0 to represent empty space).

IV. Static Ruleset Analysis

ANGELINA's design process is broken down into distinct *tasks* which compartmentalise a certain creative act, like

```
"gamename": "Before Venturing Forth",
"filename": "before_venturing",
"music": "ominous",
"color_accent": [0.4, 0.56, 0.31],
"pieces" : [
 {
  "name": "player",
  "sprite": "fighter",
 },
 //...cut for length
],
  "rules" : [
 {
  "trigger": "OVERLAP player enemy",
  "events": ["DESTROY $2"]
 },
 {
  "trigger": "OVERLAP any wall",
  "events": ["PUSHBACK $1"]
 },
 {
  "trigger": "OVERLAP enemy player",
  "events": ["DESTROY $2"]
 },
 {
  "trigger": "PLAYER_ARROW_KEY",
  "events": ["RELMOVEALL player $1", "ENDTURN"]
 },
 {
  "trigger": "ENDTURN",
  "events": ["DO_AI_HUNT enemy player"]
 }
 //...cut for length
],
  "endconditions" : [
 {
  "outcome": "1",
  "triggers": ["ALL_COVERING player goal"],
 },
 {
  "outcome": "-1",
  "triggers": ["COUNTPIECE player 0"],
 }
],
"levels" : [
 {"type": "raw",
 "width": "5",
 "height": "5",
  "data":
     [0,4,4,0,3,
   1,0,0,0,2,
   0,0,0,0,0,
   1,0,0,0,0,
   0,0,4,0,3]
 },] //cut for length
```

Fig. 1. An abridged game description file from ANGELINA.

designing a level or testing difficulty. Game logic is currently designed in the *ruleset sketching* task, which aims to produce a set of rules which can support a game. It does not have to guarantee the ruleset will produce an exciting or interesting experience, as part of this is reliant on level design. Rather, the objective here is to identify a ruleset with the most potential, and without obvious shortcomings.

In this section we describe a process of static analysis that we use to grade and filter randomly generated rulesets, using abductive reasoning to work backwards from goal states to find paths through the design space that are initiated by player action. We show that this can identify valid solutions (and routes to failure) for a game, and even provide additional design information that can impact future phases of the game design process.

A. Simple Inspection

Before ANGELINA performs abductive analysis, it first applies a simple surface-level filtering to the ruleset being considered, similar to approaches we have used in past versions of the software. This involves searching for *a priori* logical consistencies, such as a ruleset that contains two ending conditions with the same trigger, but opposite win/loss outcomes. This example is inconsistent because, in the design space we wish to consider, the player should not be able to win and lose a game in the same time step. This simple filtering has been used in many previous versions of ANGELINA as well – in our evaluation we compare this approach in isolation to full action chain generation.s

B. Action Chain Generation

The main analysis phase uses abduction to find paths through the game's design space. The result of this analysis is a series of *action chains*. An action chain is a sequence, A, of actions, $a_1, ...a_n$, which transform a starting game state into a state which triggers an end condition. Note that these are not sequences of concrete game actions, like a solution to a puzzle or the kind of sequence of game moves that an MCTS agent might produce. Instead, each step in the action chain describes a *type* of game event, a set of one or more possible actions. For example, an event chain for an adventure game might be:

- Press arrow keys to move player
- Move player over key to collect it
- Move player over door to unlock it

This chain of events doesn't describe where the key or door is in relation to the player, or how much movement is required to get there, or whether other obstacles (like enemies) will be encountered along the way. It doesn't mention these because these can only be known by looking at a ruleset in the context of a particular level. In the absence of any levels, what this action chain expresses is that there is *some* logical sequence of actions that change the abstract state of the game from a starting state into an end state. It expresses a weak endorsement that this ruleset should yield solvable level designs. Below, we describe the algorithmic process for generating a set of action chains for a particular end condition. When analysing a ruleset, ANGELINA will generate a set of action chains for every end condition in the ruleset.

C. Outline of Algorithm

ANGELINA begins by creating a new empty action chain, and adding a single action to it: the trigger of the end condition it is solving for. It then adds this action chain to an empty list of action chains called the *open list* – this contains every action chain that is neither complete nor abandoned. The process of generating action chains terminates when the open list is empty: that is, every action chain derived from the original end condition has either completed successfully, or been removed and ended in failure.

ANGELINA picks the first chain in the open list, and examines the last trigger in the chain. If this trigger is a player-initiated action, then the chain is considered to be complete, and it is added to a list of action chains called the *complete list*, and ANGELINA proceeds to the next chain in the open list. If this trigger is not player-initiated, then it becomes the current *goal* – ANGELINA must now find all possible ways to extend this chain by finding things that cause the goal to trigger.

To do this, ANGELINA uses a lookup table that maps triggers to *enabling rules*. For a given trigger, this table lists all rules which could, in some circumstance or other, activate the trigger. For example, the trigger `COUNTEQUAL X Y` activates when the number of objects of type `X` is equal to the integer value `Y`. It can be caused by the rules `SPAWN X`, which could increase the number from a value less than `Y` towards `Y`, or `DESTROY X`, which could decrease the active count from some value greater than `Y` towards `Y`.

For each enabling rule, ANGELINA binds the variables in the rule to the specific piece types defined in the goal trigger. For example, if our goal trigger is `COUNTEQUAL cake 2`, we search the lookup table for the unbound form of the rule, `COUNTEQUAL X Y`. When we find the enabling rules `DESTROY X` and `SPAWN X`, we rebind `X` back to `cake` from the example rule, to derive the two bound rules: `DESTROY cake` and `SPAWN cake`. We call these bound rules *subgoals*.

For each subgoal, ANGELINA now searches the game ruleset for any rule which contains the subgoal in its body. Every matching rule represents one possible way the goal trigger can be activated, and thus represents one possible way to extend the action chain. Since there may be multiple subgoals, and each subgoal may match multiple rules, we can't simply extend the current action chain we are working with. Instead, when ANGELINA finds a rule containing a subgoal, it duplicates the current action chain it is considering, adds the matched rule's trigger to the end of the action chain, and adds this new chain to the open list. When this new chain is later considered by the algorithm, this newly-added trigger will become the new goal it attempts to expand.

ANGELINA creates duplicate action chains for each rule it finds that triggers a subgoal. When it has done this for all of the subgoals, or if no subgoals are found, it removes the current action chain from the open list and discards it (since it has now either been replaced with one or more duplicates of itself that extend the chain, or found no possible extensions and is thus not a usable action chain.

D. Interpreting Action Chains

ANGELINA creates action chains for every end condition when analysing a ruleset. It can use both the contents and quantity of discovered chains to analyse a ruleset and decide whether it should be kept or discarded. The current version of ANGELINA requires a game to have at least one action chain which leads to a win condition, and one action chain which leads to a loss condition, although this is not a requirement of games in general (for example, a puzzle game like *A Good Snowman Is Hard To Build* [12] does not have a loss condition in a traditional sense).

We don't require all end conditions to be reachable, as long as the game can be both won and lost. This approach suits the current version of ANGELINA, but it does have implications for other approaches to automated game design, and future versions of this system. For example, the automatic generation of tutorials or help text is often based on an analysis of the game's ruleset. If game logic is expressed in the rules which are not used in completing the game, or if there are objectives which cannot be achieved, this might produce confusion in automatically generated help text, unless such a system also used action chains to analyse a ruleset in advance.

Another detail that may not be applicable to all systems is the requirement that all actions chains start with player action. In ANGELINA's current game engine play is turn-based and game logic is applied at regular intervals when a turn is ended. We have built the system with the assumption that a turn only ends when the player takes some kind of action. In a real-time game the game's logic is executed regardless of player input; taking action in Pac-Man is not required for the ghosts to hunt down and kill the player. Thus, for other game systems action chains may not have to start with player input, and may have secondary termination conditions (such as NPC behaviours).

V. Level Constraint Inference From Action Chains

In the previous section we described how action chains can provide insight into whether a ruleset supports valid play leading to end conditions. One of the strengths of the use of action chains is that they are agnostic to the content of the game itself. They assume nothing about the content of the levels, instead the technique looks for any possible permutation of game events that might lead to a particular end condition. However, while developing this system we realised that the action chains themselves contain contextual information that can be repurposed later in the design process, because they point to specific ways in which the game can be completed. This information not only helps us filter out rulesets – it can also be used to constrain the level design space and thus make that process more efficient as well.

By way of example: consider a (trivially simple) game with one end condition: the player wins if there is exactly one cake object in the game world. There are two rules: if the

player presses an arrow key, they move a person object in the direction they press; and if a person object overlaps a cake, they destroy it. By describing the rules in plain English, the reader may have already deduced that the only valid levels for this game are ones where there are two or more cakes in the world at the beginning of the game – if there is one cake the player automatically wins, and if there are no cakes then the player cannot win, because no rule allows for us to create cakes. This constraint on the game's level design is intuitive and obvious, and it would be helpful to find a way for ANGELINA to infer this automatically.

To do this, we annotate action chains as they are generated, with constraint information that applies to that action chain only. To return to an earlier example, consider the goal trigger `COUNTEQUAL X Y`. This trigger has two enabling rules: `SPAWN X` and `DESTROY X`. In the action chain generation phase we consider both as equally valid routes through the game space, but in reality there are hidden requirements on these routes being valid: `SPAWN X` can only bring about `COUNTEQUAL X Y` if the number of `X` currently in the game is less than `Y`, and *mutatis mutandae* for `DESTROY X`. We attach these conditions to every enabling rule in the lookup table. These relationships are written by hand, and while this is not time consuming, it is a clear point of human involvement in the system. We aim to investigate automating this in the future.

Whenever ANGELINA extends an action chain using an enabling rule, it also adds any attached conditions to the action chain as well. Thus, every action chain also includes a list of constraints which must apply to a level in order for the action chain to be valid. These constraints do not affect the ruleset generation phase (currently – we discuss this later, in section VII) but they *are* used to constrain the generation of levels. Prior to the use of level constraints, ANGELINA would generate levels randomly, and then use an evolutionary system with playouts to search the level design space. Now, ANGELINA can reduce the size of the level design space before and during evolution. In the cake example above, ANGELINA can understand that there must be at least two cakes in each level design, and can filter its initial population, as well as the results of crossover or mutation, to ensure these constraints are upheld.

VI. Design Space Analysis

In the previous sections we explained that action chains help to filter and reduce the search space for both ruleset design and level design. To assess the impact our approach has on on the design spaces in question, we performed some experiments, which we describe here. As a preface to the results we give, it is worth noting that automated game design systems are highly bespoke in nature: in terms of the types of game they aim to generate; in terms of the engine they use to develop games in; in terms of the algorithms they use to generate game content and the representation they choose for each part of the game. The results we provide here are unique to ANGELINA in that sense, but we believe the scale of the results speaks to the impact of our technique, and suggests it is widely applicable to other automated game design systems.

A. Ruleset Design Space Reduction

Our first experiment aimed to assess what proportion of the generative ruleset space is filtered out by using action chains. We sampled 50,000 random rulesets generated by ANGELINA with no filtering, and then evaluated each ruleset against the simple filters described in section IV-A, and then again using the action chains described in section IV-B. Using simple filters removes 9.6% of the rulesets sampled, while using action chains and selecting only games which support at least one winning and one losing action chain removes 99.5% of the ruleset samples, a tenfold increase.

This state space reduction speaks for itself, but we can also consider it in terms of time cost also. Evaluating 50,000 rulesets with our static analysis approach took 17.3 seconds on a 2017 MacBook Pro, which filtered out over 49,500 bad rulesets without evaluating them through level design or playtesting. To contrast this, we sampled ten exploratory level design processes from ANGELINA (exploratory level design is currently its next phase after ruleset design, to evaluate a ruleset by attempting to design levels for it). On average, designing a single level for a ruleset took 4 minutes and 24 seconds. Being able to rapidly cull not just unplayable rulesets, but rulesets without proper direction or goals, saves us huge amounts of time by focusing the next phase of game design on more promising rulesets.

Removing 99.5% of rulesets may sound drastic, and may lead the reader to wonder whether the remaining rulesets are quite similar or repetitive, but in fact the remaining 0.5% is still a huge design space with a wide spectrum of good and bad games in. A common approach to content generation for games is to restrict the design language or source corpus in such a way that most of the resulting possibility space is high-quality, but derivative of the source material (for example, an earlier version of ANGELINA remixed handwritten rules in such a way that there were no unplayable combinations). Here we take the opposite approach, by giving ANGELINA a generic, high-level, parameterised design language with a vast possibility space. The advantage here is that there are many games in this space that have never been conceived of by us, and some that have never been conceived of by any game designer. But the tradeoff is a much lower ratio of good to bad games. Removing 99.5% of these rulesets is simply the first step towards tackling this problem, but the remaining 0.5% still requires a lot of filtering and observation, and will only continue to grow as we expand ANGELINA's design language in the future.

B. Level Design Space Reduction

In section V we described how action chains could also identify constraints for the level design process. To understand how impactful these constraints are, we assessed what proportion of the level design space these constraints typically filter out. Because every game has different constraints, we used

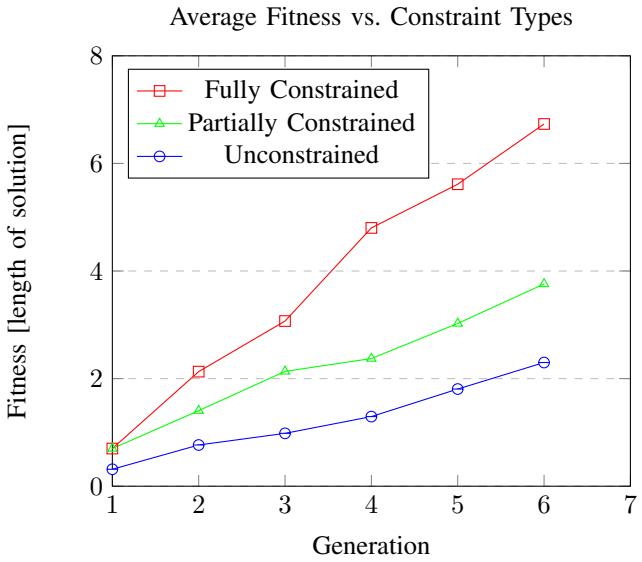

Fig. 2. A graph showing the highest fitness of the population throughout the evolutionary design of a level, sampled across 10 different level design sessions for 10 different games.

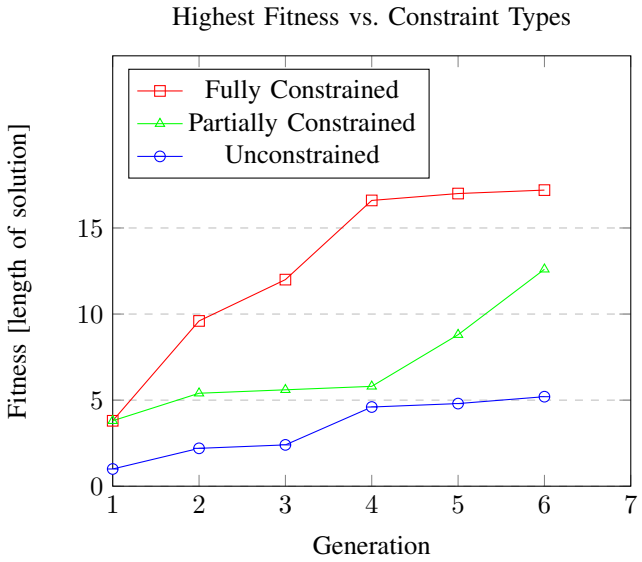

Fig. 3. A graph showing average population fitness throughout the evolutionary design of a level, sampled across 10 different level design sessions for 10 different games.

ANGELINA to generate ten game rulesets, and then generated 50,000 levels for each game, with no constraints applied. On average, the constraints filtered out 76% of the sampled levels (ranging from 67.6% to 89.7%). By relaxing our constraints on the level generation to only require they satisfy the constraints for winning the game (for example, a tutorial level which has no lose condition) this still filters out 71% of the sampled levels on average (ranging from 55.3% to 89.7%). The range of results is derived from from how varied each ruleset is – some games may have rulesets with very specific requirements, whereas other games may rely more on the arrangement of pieces rather than their specific quantity.

ANGELINA uses an evolutionary system to design levels for its games, and an MCTS player to playtest levels. The evolutionary system uses Currently the system uses a simple objective function based on the length of the shortest solution path found by the MCTS player. This by no means relates to level or game quality in general, but we have found it to be a good initial guideline towards finding average-quality puzzle levels. We leave the question of fitness function selection for level design to a future paper. We apply level constraints both during the generation of the initial random population, and during the evolutionary process when levels are regenerated or crossed over. To assess what impact this has on the fitness and convergence of the evolutionary system, we ran some additional experiments in which ANGELINA designed a level for the same both with and without constraints.

For these experiments, we took ten game rulesets generated by ANGELINA, and generated levels for each under three different conditions: *fully constrained*, where constraints were applied to the initial population and during evolution; *partially constrained*, where constraints were applied only at the initial population generation; and *unconstrained*, where no constraints were applied at all. We measured both the highest fitness at each generation, and the average fitness of the whole population, and averaged the results for each across the ten different level design sessions. The evolutionary setup for the experiment was the same as a normal exploratory level design task in ANGELINA: a population of 12 levels, run for 5 generations, with a 5% mutation rate and elitism.

C. Results Analysis

First, we discuss Figure 2 which shows the average fitness of the whole population at each generation. We can see that both partial and full constraints initially outperform the unconstrained evolution, which is to be expected since they both start with better-filtered populations. However, in later generations the fully constrained system vastly outperforms both partial and unconstrained runs. We can clearly see that the repeated use of level constraints ensures a minimum baseline of quality in the crossover and mutation operators, which contributes to growth in the average fitness over time. By contrast, the partially constrained system tends towards the same performance as the unconstrained system, since over time it is allowing levels into its population that potentially violate the basic playability constraints.

Figure 3 shows the highest fitness scores for the three experimental setups at each generation. We can see that both partial and full constraint usage outperforms unconstrained level evolution initially, but the partial constraints then slump as it is unable to improve itself as rapidly as the fully-constrained system. We can see that later generations do see an increase in fitness, but much later than the fully-constrained system and with a lower end fitness. We can also see evidence of the fitness plateauing for the fully constrained system around a solution length of 17, which we discuss below. Both

graphs show that constraining the generation of levels in the mutation, crossover and insertion of an evolutionary system lead to higher-quality populations over time.

In the later generations of Figure 3 we see a plateau in the fully constrained system. This is not a hard ceiling – fitnesses as high as 26 have been recorded for a 4x4 grid level design – but extremely high fitnesses tend to be less reliable and more prone to collapse on repeated playouts. High fitnesses can either be because the MCTS system has truly found a stable, long solution; *or* because the MCTS playout wasn't able to properly solve the level and found a very long, imperfect solution. In the latter case, given more iterations or deeper rollouts the MCTS solver may find a drastically shorter solution, and this can even happen on repeated MCTS playouts with the same settings, since our playouts are not seeded or retained across generations. This means that the plateau represents a soft limit, beyond which it is hard to find reliably longer puzzle solutions for a 4x4 grid.

Solution length works well as a metric for ANGELINA at the current time, but its unpredictability at extremely high and low values means that in the future we hope to replace it with a more complex system that allows ANGELINA to create and select its own fitness metrics for level and game design. For now, however, solution length acts as a good guideline and has led to the creation of several games which were well-received by players – the version of ANGELINA described in this paper recently designed games live at *EGX Rezzed*, a major games expo in the UK, and had its games played by thousands of expo attendees. We're confident in using it as a guideline for level design for now, and believe these fitness graphs help demonstrate the effectiveness of design constraints on producing higher-quality levels.

VII. Discussion and Future Work

The work described in this paper represents our first steps towards rapidly culling design spaces using automated reasoning techniques. There are several aspects to this work that bear discussion in its current state, and in addition we have identified areas of future work we intend to explore.

A. Identifying Complex Contradictions

Earlier in the paper we described a simple example game where the player controls a hungry character who eats cakes by moving into them. The act of eating cakes might be expressed as the following rule:

```
"trigger": "OVERLAP player cake",
"events": [
  "DESTROY $2",
]
```

In the example earlier in the paper, the game is won when a single cake is remaining in the world. An action chain for this game can be derived as follows:

- Press arrow keys to move player
- Move player over cakes to destroy them
- Win when one cake remains

In addition, this generates a constraint that there must initially be two or more cakes in any given level. However, now suppose we adjust the rule for cake destruction as follows:

```
"trigger": "OVERLAP player cake",
"events": [
  "DESTROY $2",
  "SPAWN cake"
]
```

Now, when the player destroys a cake, another cake randomly spawns elsewhere in the game. We can easily intuit here that this game is no longer winnable: the only action in the game that reduces the number of cakes also increases it, meaning it can never be reduced or increased towards another target value. However, our current method of action chain generation doesn't take this into account. There are many other examples of subtle contradictions that are hard to detect: for example, a loss condition which is satisfied before a win condition (consider the cake-eating game, except we lose if there are exactly two cakes remaining).

We have plans to solve this problem by extending the level design constraints system that is already in place. This will allow us to express more complex constraints that can be carried up the action chain that allow us to capture notions like the fact that this action chain should result in a net decrease of the quantity of a certain object type. We are also considering exploring more formal approaches to analysing these rulesets (see *Automated Reasoning*, below).

B. Intuiting Design Knowledge

Throughout the development of ANGELINA we have consulted with game designers as well as taken part in the practice of game design ourselves. One of the interesting phenomena associated with game design is the sense of *intuition* developed by game designers that allows them to view the rules of a game, in the absence of playable content such as a level or puzzle, and infer properties the game has or simple affordances the game's ruleset offers. Designers use this when prototyping games, when considering extensions to games or trying to rebalance or fix design problems, and also use it to evaluate games designed by others.

The inferred level constraints constructed from event chains represent a very basic form of intuition about a game. They don't represent precise knowledge gained from playtesting, nor do they represent *a priori* knowledge about game design in general: they represent a way for an automated game designer to rapidly gain simple guiding principles about a particular game. In this paper we've already shown that these have value as a way of reducing the complexity of search tasks in automated game design, but from a Computational Creativity perspective this may be an impactful way for the system to demonstrate understanding and insight in game design.

We are interested in pursuing these ideas further, and seeing what other kinds of intuition-like knowledge can be gleaned from analysing rulesets prior to engaging in level design. Designers can often hypothesise about the impact of a change

on a ruleset, or sketch out what affordances a particular rule might have. While this arguably involves some kind of rapid mental simulation, as well as an accumulated history of thousands of hours of playing and experiencing different kinds of game, it is nevertheless a fascinating skill that would be hugely beneficial to automated game designers. Not least because it speeds up the prototyping phase of game design, and reduces the need to exhaustively playtest ideas, but also because it provides new opportunities to frame and describe the design process to others, which is a crucial idea that underpins a lot of work in Computational Creativity [13]. Conveying intuition and hypotheses to people as a motivation for work could have a huge impact on the perception of automated game designers – not just by audiences of players, but also by people within the games industry who might be working with such software one day as co-creators [14].

C. Automated Reasoning

Our analysis process currently relies on structures such as lookup tables to bridge the semantic gap between language's keywords (like `DESTROY`) and their underlying meanings (reducing the quantity of something). ANGELINA's domain specific design language was created with readability in mind, ease of authoring, and ease of automatic modification – we didn't anticipate the need to have the language formally analysed, and so this process is not always straightforward and reasoning about more complex properties of a game gets increasingly difficult.

In the future we are considering building a model of ANGELINA's game engine that would allow ANGELINA to formally specify its games and then reason about their properties. Thus instead of looking up the fact that `DESTROY X` reduces the number of `X` objects in the world, it could reason about the actual impact that keyword has on a concrete model of a game, and use such a model to resolve complex conflicts within a ruleset or identify deeper constraints implied by the game. Work by Martens [15] and Smith [16] show different approaches to applying logical representation to game systems, which we plan to take as inspiration.

VIII. Conclusions

In this paper we describe how applying abductive logic to game rulesets can greatly reduce the size of a game design space, by filtering out rulesets that have design deficiencies, as well as providing insights into the game ruleset that can constrain the space of possible level designs. We showed that by applying action chain filtering to ANGELINA, our automated game design system, we were able to cull over 99.5% of the total ruleset possibility space, and 75% of the average total level design possibility space. This enables ANGELINA to work with much less game design knowledge supplied *a priori* in terms of game templates, and instead work through a large possibility space to identify interesting valid games, and design levels that demonstrate a better understanding of the ruleset in question.

Automated game design is a major challenge for game AI research, identified as one of the frontier problems in the field by a recent panorama paper [17]. The field has many challenges, but chief among these is the multiplicatively vast design space that results in combining generative tasks like level design or ruleset design. Developing new ways to cut down the size of these possibility spaces, in order to analyse more deeply the promising subsections of these spaces, will help the field advance further and hopefully impact neighbouring research efforts in general game playing and generative systems.

IX. Acknowledgements

This work is funded by EC FP7 grant 621403 (ERA Chair: Games Research Opportunities). Thanks to MPI-SWS for supporting the work of the first author.

References

[1] M. Cook and G. Smith, "Formalizing non-formalism: Breaking the rules of automated game design," in *Proceedings of the Foundations of Digital Games Conference*, 2015.

[2] R. Hunicke, M. Leblanc, and R. Zubek, "Mda: A formal approach to game design and game research," in *In Proceedings of the Challenges in Games AI Workshop, Nineteenth National Conference of Artificial Intelligence*, 2004.

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

[4] M. Treanor, B. Schweizer, I. Bogost, and M. Mateas, "The micro-rhetorics of game-o-matic." in *Proceedings of the Foundations of Digital Games Conference*. ACM, 2012.

[5] G. A. B. Barros, A. Liapis, and J. Togelius, "Murder mystery generation from open data," in *Proceedings of the International Conference on Computational Creativity*, 2016.

[6] C. Fernández-Vara and A. Thomson, "Procedural generation of narrative puzzles in adventure games: The puzzle-dice system," in *Proceedings of the The Third Workshop on Procedural Content Generation in Games*, 2012.

[7] J. Togelius and J. Schmidhuber, "An experiment in automatic game design," in *Proceedings of the IEEE Conference on Computational Intelligence in Games*, 2008.

[8] A. Khalifa, D. P. Liebana, S. M. Lucas, and J. Togelius, "General video game level generation," in *GECCO*. ACM, 2016, pp. 253–259.

[9] M. Cook, "A vision for continuous automated game design," in *Proceedings of the Experimental AI and Games Workshop at AIIDE*, 2017.

[10] T. Schaul, "A video game description language for model-based or interactive learning," in *Proceedings of the IEEE Conference on Computational Intelligence in Games*, 2013.

[11] S. Lavelle, "PuzzleScript," http://www.puzzlescript.net/, 2014.

[12] A. Hazelden and B. Davis, "A good snowman is hard to build," http://www.agoodsnowman.com/, 2015.

[13] J. Charnley, A. Pease, and S. Colton, "On the notion of framing in computational creativity," 2014.

[14] M. O. Riedl and A. Zook, "AI for game production," in *Proceedings of the IEEE Conference on Computational Intelligence in Games*, 2013.

[15] C. Martens, "Ceptre: A language for modeling generative interactive systems," in *AIIDE*. AAAI Press, 2015, pp. 51–57.

[16] A. M. Smith and M. Mateas, "Answer set programming for procedural content generation: A design space approach," *IEEE Trans. Comput. Intellig. and AI in Games*, vol. 3, no. 3, pp. 187–200, 2011. [Online]. Available: https://doi.org/10.1109/TCIAIG.2011.2158545

[17] G. N. Yannakakis and J. Togelius, "A panorama of artificial and computational intelligence in games," *IEEE Trans. Comput. Intellig. and AI in Games*, vol. 7, no. 4, pp. 317–335, 2015.