1 MEASURING QUALITY OF GRAMMARS FOR PROCEDURAL LEVEL GENERATION

Abstract

Grammar-based procedural level generation raises the productivity of level designers for games such as dungeon crawl and platform games. However, the improved productivity comes at cost of level quality assurance. Authoring, improving and maintaining grammars is difficult because it is hard to predict how each grammar rule impacts the overall level quality, and tool support is lacking. We propose a novel metric called Metric of Added Detail (MAD) that indicates if a rule adds or removes detail with respect to its phase in the transformation pipeline, and Specification Analysis Reporting (SAnR) for expressing level properties and analyzing how qualities evolve in level generation histories. We demonstrate MAD and SAnR using a prototype of a level generator called Ludoscope Lite. Our preliminary results show that problematic rules tend to break SAnR properties and that MAD intuitively raises flags. MAD and SAnR augment existing approaches, and can ultimately help designers make better levels and level generators.

1.1 INTRODUCTION

Grammar-based level generation is a form of Procedural Content Generation (PCG) that raises the productivity of game level designers. Instead of hand-crafting levels, designers create a level transformation pipeline that generates levels for them by authoring modules, grammars and rewrite rules. The grammar rules work on data structures such as strings, tile maps and graphs, which can be used for generating names, level layouts and missions. These artifacts are step-by-step transformed and combined until a final detailed and fully populated level is generated, with missions, power-ups, challenges, enemies, hidden treasures, secret pathways, encounters, etc. Ideally, each generated level has the intended qualities. Unfortunately, improving the productivity of level designers comes at the cost of quality assurance.

In practice, many small problems arise, such as levers in walls, blocked pathways, missing encounters and lava adjacent to water. A lack of direct manipulation compromises the ability of designers to isolate and improve level qualities, e.g., when authoring bridges, forests or paths. As a result, some generated levels may lack intended goals, challenges and missions.

This chapter was previously published as R. van Rozen and Q. Heijn. "Measuring Quality of Grammars for Procedural Level Generation". In: *Proceedings of the 13th International Conference on Foundations of Digital Games, FDG 2018, as part of the 9th Workshop on Procedural Content Generation, PCG 2018, Malmö, Sweden, August 7–10, 2018*. Ed. by S. Dahlskog, S. Deterding, J. Font, M. Khandaker, C. M. Olsson, S. Risi, and C. Salge. ACM, 2018, pp. 1–8. ISBN: 978-1-4503-6571-0. DOI: 10.1145/3235765.3235821

The qualities of generated levels depend on the composition of grammar rules and how they are combined in sequence. Therefore, potential bugs often remain unknown until they are observed during playtesting. Additionally, the combinatorial explosion resulting from recursive rule expansions complicates forming mental models required for reasoning about intended qualities, and how they are represented in the grammar or intermediate data. Moreover, it is hard to predict how individual rules affect the overall level quality.

Grammars are brittle, i.e. code that is liable to break easily. Designers require special measures to ensure that qualities once introduced, remain intact, preventing successive rewrites from breaking levels. Fixing one level with a rule that prevents an occurrence may introduce new problems in others. In general, there is a lack of tools and techniques for authoring, debugging, testing and improving rules that introduce and preserve design intent. As a result, the full potential of these techniques has not yet been realized.

We aim to improve the quality of grammar-based procedural level generation in general, and focus on grammars that work on tile maps in particular. We motivate our research by studying and improving Ludoscope, a state-of-the-art development environment for generating very diverse game levels. Since its inception, Ludoscope was developed by Ludomotion for indie game development, and successfully applied to a rogue-like dungeon crawler called Unexplored. We address the need of developers for better tools. This paper proposes and contributes two enabling techniques:

1. Metric of Added Detail (MAD), a novel metric that indicates if a grammar rule adds or removes detail to a tile map. We hypothesize that grammars gradually add detail. MAD leverages a detail hierarchy, a binary relation on alphabet symbols indicating which symbol is more detailed, which can easily be derived from transformation pipelines.
2. Specification Analysis Reporting (SAnR), a technique that offers a level property language for expressing level qualities. SAnR analyzes and reports how these properties evolve over time in level generation histories.

We demonstrate the feasibility of MAD and SAnR by implementing Ludoscope Lite (LL), a light-weight version of Ludoscope intended to study level quality. LL is implemented using Rascal, a meta-programming language and language workbench. Our preliminary evaluation shows that SAnR can express and analyze simple level properties, and that MAD raises flags for rules that remove detail. MAD and SAnR augment existing approaches by supporting gradually adding detail and analyzing level generation histories, which ultimately helps designers make better levels and level generators.

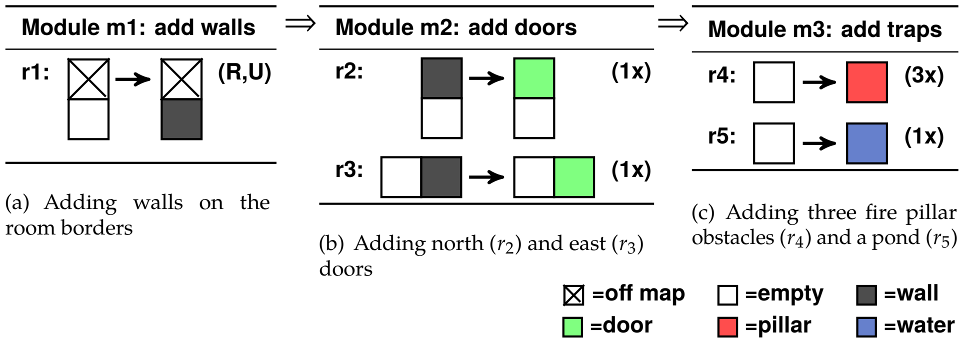
1.2 RELATED WORK

Evaluating content generators and their output is a key open research problem [STN16; SMS+17; YT18]. Generators can be analyzed in terms of generated content, e.g., Summerville et al. evaluate metrics for difficulty, visual aesthetics and enjoyment of platform games [SMS+17].

We take an authoring perspective on level grammars. Our approach stands apart by also taking into account how generated levels are generated. This enables level designers to relate qualities of generated levels back to the source code of the generator (grammar rules) and make targeted improvements.

Level grammars are under-specified, since they also generate many levels that are bad with respect to design constraints. The challenge is authoring a set of rules that efficiently generates varied and well-structured results capturing design intent while limiting the recursion. Smith and Mateas propose explicitly describing design spaces as an answer set programs, and show generators can be sculpted for a variety of content domains [SM11]. Van der Linden et al. focus on improving authoring and controlling level generators by expressing gameplay-related design constraints. They use graph grammars to encode these constraints, and generate action graphs that associate player actions and content for generating complete layouts of game levels [vdLLB13]. We refer to a survey of van der Linden et al. for a wider discussion on techniques for procedural dungeon generation [vdLLB14].

We relate our work to other content generators that use grammars. Tracery is a grammar-based tool for authoring stories and art as structured strings that has been used for generating names, descriptions, stories in poetry, Twitter bots and games [Com15; CKM15]. PuzzleScript is a language and authoring environment which uses rewrite rules to express puzzle mechanics [Lav15]. Ludoscope is a visual environment for authoring level transformation pipelines as grammars that builds upon the mission and spaces framework [Dor10; DB11]. Pipelines consist of modules that contain grammars, alphabets and recipes that transform level artifacts such as strings, tile maps, graphs and Voronoi diagrams. In particular, recipes are crucial to control the generation and focus the application of rules for obtaining aesthetically pleasing levels. Recipes parameterize modules with instructions, that determine the ordering of rules and limit how often rules work. Member values annotate tiles with extra information. Both help reducing the generation space, but neither are well-suited to check qualities off-line and independently. Ludoscope is neither extensively documented, nor currently available as open source software. Karavolos et al. report experiences on applying Ludoscope to a platformer and a dungeon crawl game, which require very different transformation pipelines [KBB15]. Our approach closely follows the pipeline structure of Ludoscope, but it improves upon its capabilities for analyzing grammar and level quality.

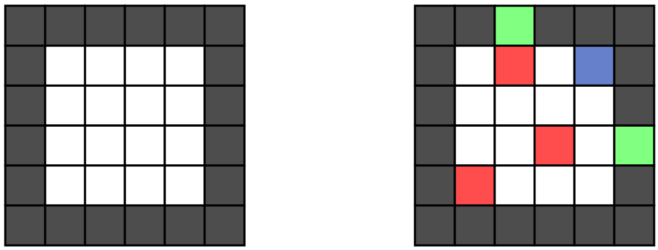


(a) Adding walls on the room borders

(b) Adding north (r_2) and east (r_3) doors

(c) Adding three fire pillar obstacles (r_4) and a pond (r_5)

Figure 1.1: Level transformation pipeline consisting of three modules



(a) Empty room with walls

(b) Example room with content

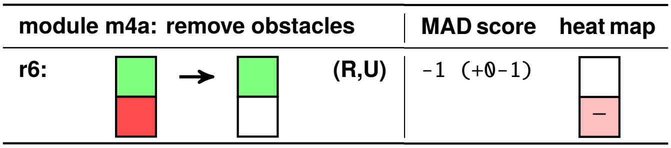
Figure 1.2: Tile maps that are input and output of the pipeline

1.3 GRAMMARS FOR LEVEL GENERATION

Here, we introduce quality issues in grammar-based level design using a simple example that generates a room for a dungeon crawler, which illustrates some of the challenges that arise during authoring grammars. It isolates problems that have larger more complex forms in practice, e.g., in Unexplored. We relate questions designers might have in Section 1.3.2 to technical challenges in Section 1.3.3.

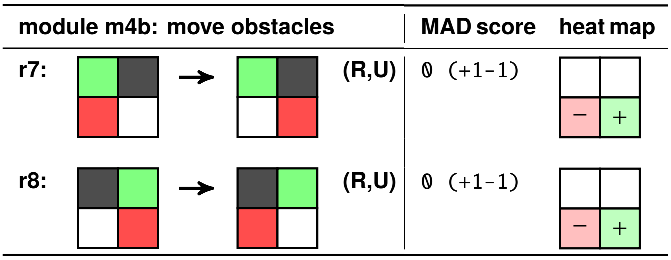
1.3.1 *Introductory Example*

In dungeon crawlers, tile maps often represent rooms connected by pathways. Our level generation pipeline, shown in Figure 1.1, generates rooms with two doors connecting to a larger dungeon. It consists of three modules of grammar rules that represent sequential level transformation phases. The grammar rules rewrite pieces of the tile map matched by their pattern on their left hand side to the pattern on their right hand side. The pipeline takes an empty tile map as input, e.g., of 6x6 tiles. Each



(a) Module removing pillars (b) MAD score and heat map

Figure 1.3: Module for removing pillars that block doors



(a) Moving pillars left (r_7) or right (r_8) (b) MAD score and heat map

Figure 1.4: Module for moving pillars that block doors

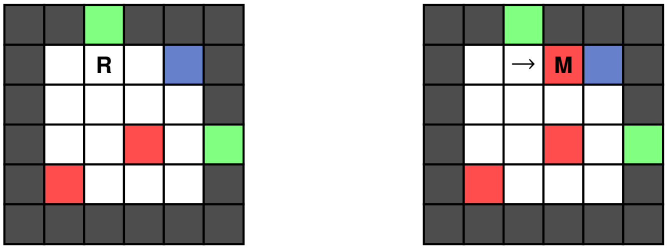
phase randomly selects and applies rules, gradually adding detail. Many levels can result, and as we will see, not all of these are what a designer might deem desirable.

First, module m_1 adds walls on the borders of the tile map (Figure 1.1(a)). It contains one rule called r_1, whose left hand pattern matches on an empty tile on the north edge of the map. Grammar rule r_1 replaces an empty tile on the north edge of the map with a wall. Rules can have modifier symbols to its right. The (U) symbol to the right indicates that rule r_1 is applied as many times as possible. The (R) symbol indicates that rule r_1 is also applied to the east, south and west borders of the map. The result of module m_1 is always a tile map with walls on its borders, e.g., Figure 1.2(a) is the output at 6x6.

Next, module m_2 adds doors in the north and east walls that connect the room to other parts of the dungeon (Figure 1.1(b)). The rules r_2 and r_3 respectively add a door in the north and east walls. These rules are applied exactly once (1x).

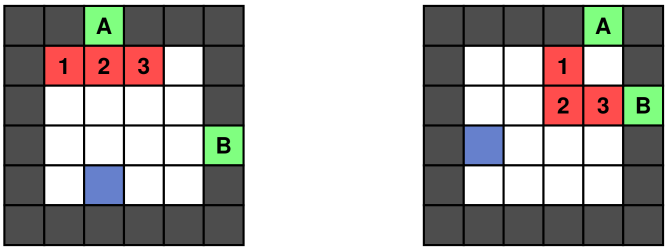
Finally, module m_3 introduces challenge (Figure 1.1(c)). Rule r_4 places three fire pillars, traps that set players on fire if they remain close too long. In addition, rule r_5 adds a pond of water the player can use to extinguish the flames.

1.3.2 *Level designer questions*



(a) Module m4a removed a pillar at R (b) Module m4b moved pillar M

Figure 1.5: Repairing the example level of Figure 1.2(b) in two ways

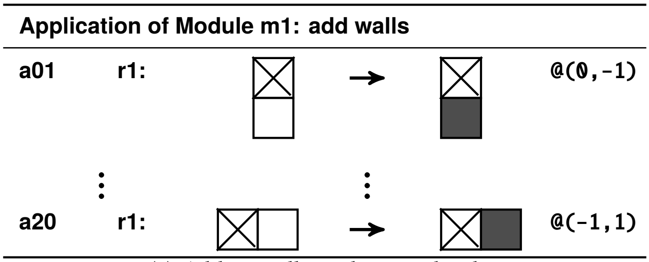


(a) No space to move pillar 2 away from door A (b) Moving pillar 3 can block door A Water remains unreachable

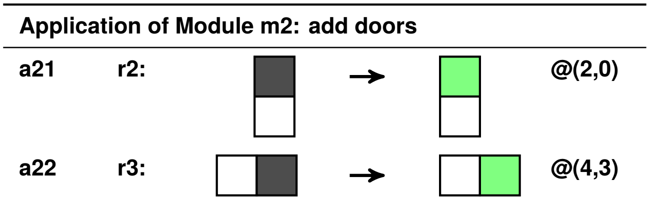
Figure 1.6: Levels that cannot be repaired by Module m4b

The pipeline of Figure 1.1 can also generate problematic levels. For instance, in Figure 1.2(b), a fire pillar in front of the north door prevents players from passing. One way to fix this is to *patch* the level by removing obstacles, as shown in Figure 1.3(a) results in Figure 1.5(a). However, fewer pillars than intended may reduce the difficulty. Another way is moving obstacles away from doors, as shown in Figure 1.4(a), which results in Figure 1.5(b). Unfortunately, other problematic output still exists, e.g., Figure 1.6. Authoring level grammars is hard, even for this tiny example. Questions about quality a designer might have are:

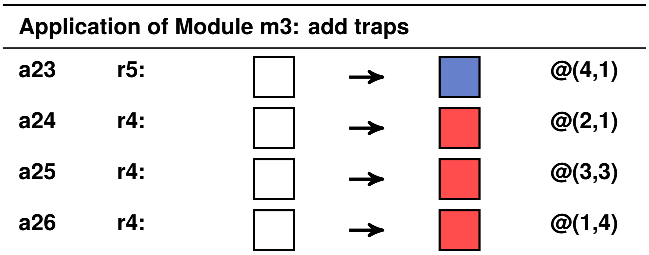
1. **Efficiency.** Do the grammar rules efficiently generate levels, or is time wasted on overwritten dead content?
2. **Effectiveness.** Do the grammar rules effectively generate levels that contain all the intended objects, composite structures, problems and solutions, or are some parts missing?
3. **Root-cause analysis.** Given a level with a problem, by which rules were the affected tiles generated?



(a) Adding walls on the room borders



(b) Adding north (a_{21}) and east (a_{22}) doors



(c) Adding three pillars (a_{24},a_{25},a_{26}) and a pond (a_{23})

Figure 1.7: Level generation history showing how rules generated the example level shown in Figure 1.2(b)

4. **Bug-fixing.** Does changing a rule improve levels, or does it also introduce new problems?
5. **Bug-free.** How can unwanted situations be prevented and removed from the level generation space?

Other relevant questions not further discussed here are, e.g.,

- **Playability.** Are the challenges of all generated levels solvable, or are there ways in which players can get stuck?
- **Challenge.** Are the levels challenging to play?

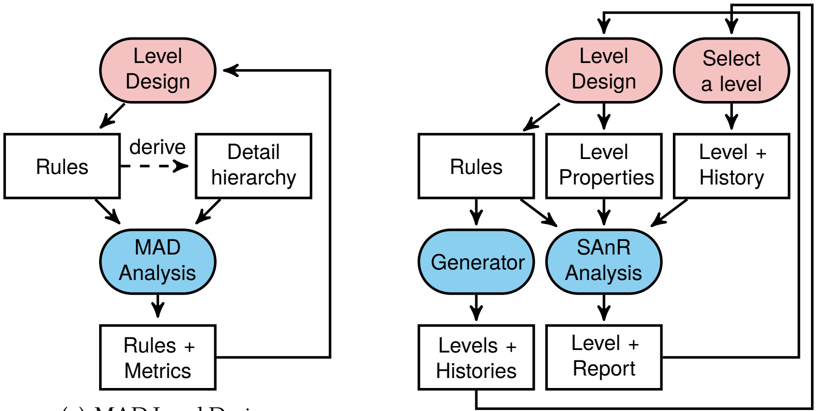
1.3.3 *Challenges*

Here, we identify technical challenges that need to be addressed for answering questions of level designers described in Section 1.3.2.

1. **Static analysis and metrics.** Profiling the applications of rules helps to asses efficiency measuring (relative) times and amounts. However, static analysis may also help predict rule efficiency. Upper bounds on rule applications enable reasoning about worst-case scenarios. Left hand patterns that can never match indicate dead code. In addition, metrics can help assess to which extent rules contribute to generating an intended result, to find bad rules.
2. **Analyzing the level generation space.** Viewed as a state-space exploration problem, rules might rewrite levels to prior states. For a given level, the shorter its trace of rewrites, the more efficient its generation.
3. **Expressing and analyzing level qualities.** Grammar rules lack ways to specify properties at specific points in the pipeline, e.g., if objects are (not) adjacent, contained, intact or missing. Designers need an additional formalism for effectively specifying properties that intuitively capture design intent. To see how qualities evolve, levels can be checked against these properties after each transformation.
4. **History analysis.** Generators produce tile maps by applying grammar rules in sequence, e.g. Figure 1.7. However, these generation histories are usually not stored. For identifying rules that impact tiles, or groups of tiles, designers require an analysis of the level transformation history.
5. **Impact analysis.** Assessing the impact of rules on many generated results requires isolating rule effects. The position in the pipeline scopes the locality of impact, and a dependency analysis can exclude side-effects, but an exhaustive impact analysis requires generating examples.
6. **Test Automation.** Testing the impact of changes on all possible levels is not feasible. As a result, levels may exist that contain bugs. The challenge is devising a test harness that generates representative levels for finding bugs.
7. **Debugging.** Identifying and fixing bugs requires appropriate views and tools for setting break points and making modifications, e.g., selecting one or more adjacent tiles to filter and analyze selected properties.

1.4 GRAMMAR ANALYSIS AND DEBUGGING

We approach the challenges of Section 1.3.3 from a software evolution perspective. We propose two solutions, Metric of Added Detail (MAD) and Specification Analysis Reporting (SAnR). Figure 1.8 schematically shows how designer activities and algorithmic processes (respectively shown as pink and blue rounded rectangles) produce (outgoing arrows) and consume (incoming arrows) artifacts (rectangles). The field of software evolution studies how software evolves over time [MWD+05]. As software ages, it conforms less and less to the changing expectations of its users. In addition,



(a) MAD Level Design

(b) SAnR Level Design

Figure 1.8: Producing MAD and SAnR level design artifacts

for developers it also becomes harder over time to adjust software and maintain its quality. Research includes methods and techniques for analyzing source code and for making changes to improve the software quality. Since game requirements are mainly non-functional and evolve rapidly, these techniques are also vital for game quality.

1.4.1 *Metric of Added Detail*

Metrics have been proposed to analyze how changes to source code impact software quality. Volume (or size) can be measured by counting Lines Of Code (LOC), and branch points in the control flow of methods can be measured using Cyclometric Complexity (CC). At any moment, metrics are just abstract values, but when studied over time they can provide insight into phenomena and quality, in particular when developers have questions regarding the effect of maintenance and new requirements that require programming. Heitlager et al. describe a software maintainability model [HKV07], which requires that measures are 1) technology independent; 2) simply defined; 3) easy to understand and explain; and 4) enablers of root cause-analysis, relating source code properties to system qualities.

Here we introduce the Metric of Added Detail (MAD), a simple metric for grammars operating on tile maps, which is easy to explain and understand. MAD does not directly predict level quality, but instead measures the effect on detail of individual rules by leveraging the assumption that details are gradually added (Figure 1.8(a)).

We define MAD in Figure 1.9, using the concise functional notation of Rascal. MAD requires a *detail hierarchy*, represented as a binary relation on grammar symbols (line 2). Rules are represented as lists of tuples of source and target symbols that abstract from tile map dimensions (line 3). The result of the metric adds a *score*

```
1  module util::mad::Metric
2  alias Detail = rel[str greaterSymbol, str lesserSymbol];
3  alias Rule = lrel[str lhs, str rhs];
4  alias RuleScore = lrel[str lhs, str rhs, int score];
5
6  RuleScore getRuleScore(Rule r, Detail d)
7    = [<lhs, rhs, getTileScore(lhs, rhs, d)> | <lhs, rhs> ←r];
8
9  int getTileScore(str lhs, str rhs, Detail d){//rewriting a tile
10   if(<lhs,rhs> in d) return −1; //removes detail
11   else if(<rhs,lhs> in d) return 1; //adds detail
12   else return 0; //retains detail
13 }
```

Figure 1.9: Metric of Added Detail as a RASCAL program

element to each tuple that records if detail is added (score +1), removed (score -1) or persisted (score 0) (line 4). The function *getRuleScore* specifies the rule metric as a list comprehension (lines 6–7). Given a rule and a detail hierarchy, it calculates for each symbol on the left hand side if the right hand side adds or removes detail using the function *getTileScore* (lines 9–13). Displayed as a heat map, the result is aggregated as a sum of tile detail scores.

1.4.2 *Deriving Detail Hierarchies*

MAD is tool independent and rule parametric, but it requires a detail hierarchy, which needs to be derived. Modules imply a natural hierarchy for tools that use level transformation pipelines, each phase introducing symbols that are more detailed than the last. Using this approach, we derive the following detail hierarchy for the example of Section 1.3.1 Figure 1.1 {water, pillar} > door > wall > empty, or visually { , } > > > .

Competing non-deterministic rules do not sequentially add detail, e.g., r_4 or r_5 adds or first. Therefore, deriving a symbol hierarchy for exposing data generated and overwritten within a module is less straightforward. We see the following alternatives:

1. Allow an explicit user-defined detail hierarchy, or derive it from an explicit rule ordering such as a Ludoscope recipe.
2. Assume detail is sequential to the rules in the module.
3. Add the inverse to the relation for symbols with the same rank in the hierarchy, e.g., > and > . However, this is not very intuitive.

```
start syntax LevelSpec
  = spec: Property*;
syntax Property
  = property: Condition TileSet;
syntax Condition
  = none: "no"            //tile set is empty
  | count: INT size "x";  //tile set is of specific size
syntax TileSet            //defines a set of tiles (now visible)
  = tileSet: ID tileName FilterNow FilterWhere;
syntax FilterNow          //filters the tile set (now visible)
  = nowAny:               //empty alternative, no filter
  | nowAdjacent: "adjacent to" ID tileName;
syntax FilterWhere        //filters a tile set (historically)
  = everAny:              //empty alternative, no filter
  | everRule: "in" ID ruleName; //topographical location
```

Figure 1.10: Syntax of Level Property Language in Rascal

1.4.3 *Analyzing Rules with MAD*

Using the detail hierarchy derived in Section 1.4.2 we calculate MAD scores for rules of modules *m4a* and *m4b* intended to fix broken levels, shown in Figure 1.3(a) and Figure 1.4(a). Rule r_6, which removes fire pillars, has a negative effect on detail, as shown in Figure 1.3(b). The effect of rules r_7 and r_8 that instead move them, shown in Figure 1.4(b), is neutral. MAD helps designers assess if rules contribute to generating intended results, and augments intuitions with facts. Rules that remove details may be fixes, but may also cause dead content or regressions in the level generation space that waste time.

1.4.4 *Expressing and Analyzing Level Properties*

Here we address the challenges of expressing and analyzing level qualities from a Software Language Engineering perspective [Läm18]. We propose Specification Analysis Reporting (SAnR), a technique for analyzing level grammars against level properties. In the mixed-initiative design process shown in Figure 1.8(b), designers author a grammar (rules and modules) and SAnR level properties, a generator generates levels, and the designers selects one level to analyze, for which SAnR generates a report.

SAnR provides a property notation. This is a so-called Domain-Specific Language (DSL), a language that offers appropriate notations and abstractions with expressive power and affordances over a particular problem domain [vDKV00], in this case specifying properties of tile maps as correct outcomes of tile map transformations.

	$a_{01} \ldots a_{21}$	a_{22}	a_{23}	a_{24}	a_{25}	a_{26}	a_{27} (alt.)	a_{27} (alt.)
2**x** door **in** walls		✓						
1**x** water			✓					
3**x** pillar						✓	✗	
no pillar **adjacent to** door	✓			✗			✓	✓
no water **adjacent to** pillar	✓							✗

(a) Level Property Language specification encoding level qualities

(b) Level generation report showing how the level of Figure 1.2(b) evolved over time in Figure 1.7

(c) Certain result of module m4a

(d) Possible result of module m4b

Figure 1.11: Level properties and a level generation report showing two alternatives

We show its syntax in Figure 1.10, and give an informal description of its language semantics. Instead of writing new grammar rules, a SAnR level specification is a set of declarative properties, which refer to names used in the grammar (line 1). Given a level history as a sequence of rule-based model transformations, e.g., Figure 1.7, properties can be evaluated at each point in time, yielding either *true* or *false*. Properties work on tile locations, places on tile maps specified by x and y coordinates denoted as @(x,y), the top left tile being @(0,0). A property is a condition on a set of tile locations visible on a tile map (line 2), which must either be empty (line 4) or of a specific size (line 5). The set is built by collecting tile locations using names from the grammar alphabet, e.g., "door" retrieves a set containing each location of a door. On the example of Figure 1.2(b) this yields {@(2,0), @(4,2)}, which means "2**x** door" is true and "1**x** door" is false. Locations can be filtered in two optional ways.

1. **Adjacency.** The **adjacent to** keyword (lines 8-10), filters locations that do not share at least one side with tiles of another kind, e.g.,"door **adjacent to** pillar", denotes a set of locations of door tiles next to at least one pillar.
2. **Topography.** The **in** keyword (lines 11-13), filters out locations that were never affected by a rule rewrite. In other words, we use rule names to collect sets of tile locations from the level generation history as *"topographical regions"*. The resulting set is the intersection between the left and right hand operands. For example "door **in** walls" gives the set of *door* locations in the region affected by rule *walls*.

1.4.5 *Analyzing Level Generation Histories*

The SAnR analysis uses properties for generating level generation reports that show when properties were valid, and when they became invalid. For example, given the level generation history of Figure 1.7, and the properties of Figure 1.11(a), SAnR evaluates the properties after each transformation step, yielding the report of Figure 1.11(b). From the report we read that at step a_{24} transformation r_4: □ → ■

@(2,1) places a pillar in front of the north door, which invalidates the property "**no** pillar **adjacent to** door".

1.4.6 *Analyzing Rule Impact*

SAnR can also be used to analyze the impact of new rules on existing levels with respect to level properties. For instance, we can spot problems at alternative steps a_{27} in the report of Figure 1.11 caused by modules $m4a$ and $m4b$ intended as fixes, shown in Figure 1.3(a) and Figure 1.4(a). On the one hand, Figure 1.11(c) shows that when module $m4a$ removes the pillar with transformation r_6:  $\rightarrow$  @(2,0) this breaks the property "**3x** pillar". On the other hand, Figure 1.11(d) shows that when module $m4b$ moves the pillar to the east with transformation r_7**:**  $\rightarrow$  @(2,0) this breaks the property "**no** pillar **adjacent to** water".

1.5 PRELIMINARY EVALUATION

Here, we report on a preliminary evaluation of the use of MAD and SAnR in the implementation of a prototype level generator called Ludoscope Lite.

1.5.1 *Implementation of LudoScope Lite*

Ludoscope Lite (LL) is a light weight version of Ludoscope intended for rapid prototyping, research and experimentation with analysis and generation techniques for making better grammar-based game levels and generators. Its focus is initially on designing and validating approaches for tile maps, which are later implemented and applied in Ludoscope. We use language work bench [EVV+13] and meta-programming language RASCAL* [KvdSV09] to implement MAD and SAnR as separate reusable modules and integrate both in LL†.

Table 1.1 gives an overview of the components of LL and their size in Lines of Code (LOC) relative to Ludoscope. Of course, the user-friendly IDE of Ludoscope has many features LL lacks, explaining the size difference. LL integrates a grammar-based parser that reads the storage format of Ludoscope. The ultimate goal is compatibility, sharing syntax and semantics for generating and analyzing rules. We apply test-driven development, encoding expected behaviors for most of its features in a combination of unit and integration

*https://www.rascal-mpl.org

†https://github.com/visknut/LudoscopeLite

Table 1.1: Source code size of Ludoscope and Ludoscope Lite

Component	**Ludoscope (KLOC)**	**LL (KLOC)**
IDE (features differ)	10.5	0.3
Parser + execution	10	1.7 + 0.4
Test + test data	?	1.5 + 0.7
Metric of Added Detail	not yet	0.1
Level Property Language	not yet	0.3
Extension wrappers	-	0.4
Total	20.5	5.5

Table 1.2: SAnR data on the example the pipeline of Figure 1.1 and its two extensions modules $m4a$ and $m4b$

Data	**Example**	**+ $m4a$**	**+ $m4b$**
Unique histories	9846	9858	9844
Unique tile maps	9171	9014	8775
Broken tile maps	6254	6132	4613
Bugs found	2	2	4

Table 1.3: SAnR level generation reports for 10K random executions. The rules r_n refer to Figure 1.1, Figure 1.3(a) and Figure 1.4(a)

Property	**Example**	**+ $m4a$**	**+ $m4b$**
2**x** door **in** walls	-	-	-
1**x** water	-	-	-
3**x** pillar	-	r6 (3226x)	r7 (111x) r8 (112x)
no pillar **adjacent to** door	r5 (3164x)	-	r5 (438x)
no water **adjacent to** pillar	r5 (5686x)	r5 (5209x)	r5 (5482x)

tests for regression testing. The histories and reports shown in this paper are generated by LL, which currently still generates them as strings. A more user friendly visualization is work in progress.

1.5.2 *Test Automation*

We use LL to evaluate SAnR on the running example of Section 1.3.1[‡] We wish to learn if LL and SAnR can help automate tests, and run 10K random executions (or simulations) on the pipeline Figure 1.1, and its extensions, shown in Figure 1.3(a) and Figure 1.4(a), which makes 30K executions total. For each execution, we record the model transformation history and use SAnR and the properties of Figure 1.11(a) to obtain a report.

Table 1.2 displays an overview of the results, which were obtained in about 10 minutes of run time. The unique number of histories is lower than 10K because some executions yielded the same transformations. In addition, different transformation sequences can produce the same tile map, which explains why there are fewer unique tile maps. We consider a tile map *broken* when not all SAnR property are satisfied. In addition, Table 1.3 shows which rules break properties (in how many histories) for each pipeline version, which helps designers compare and analyze causes.

We gain the following insights. The test automation approach is feasible, and issues can be found in seconds. In addition, by relating the number of unique outputs to the number of broken outputs we can get an idea how serious issues are. Naturally, 10K random executions says nothing about test coverage, but it improves upon random manual testing. We confirm that module *m*4*a* is a bad fix. We note that although extension *m*4*b* increases the number bugs, it also generates fewer broken tile maps. Clearly, the pipeline still requires fixes. Of course, the example is small and not representative of the size and complexity of transformation pipelines of games such as Unexplored. However, our test automation setup is reusable, and enables testing other grammars with larger pipelines too.

1.6 DISCUSSION

MAD and SAnR provide a means for answering designer questions of Section 1.3.2. Here we discuss the befits and limitations of the approaches and threats to validity.

[‡]There is one difference, LL implements Ludoscope recipes for limiting the amount of times that rules are applied. As a side-effect, this limits sequences and reduces the level generation space.

1.6.1 *MAD Level Design*

MAD gives a partial answer to the question if rules generate levels efficiently. The metric helps designers identify rules that remove detail, and possibly waste time on generating cause dead content. It supports the single responsibility principle, exposing modules add many details at once. However, MAD does not address the challenge of analyzing the state space. At best, it can help identify rules that may lead to longer level generation traces. In addition, we do not know if MAD can be used for data structures other than tile maps, e.g., for grammars that work on graphs. Finally, MAD is not yet empirically validated.

1.6.2 *SAnR Level Design*

SAnR properties enable analyzing how effectively rules generate intended levels, e.g. for simple tile adjacency, counting, missing tiles, and topographical inclusion. Properties depend only on the names of rules and tiles, which separates concerns but complicates refactoring grammar rules. SAnR analyzes levels by checking properties against generation histories, and assumes these are correctly generated. Therefore, SAnR reports are only as good as the grammar engine, which may also contain bugs. Of course, our approach is not the first that checks simple invariant conditions. However, to the best of our knowledge, checking properties that use level generation histories and grammar rule names to collect topographical regions of tile locations is new.

SAnR can help designers analyze quality and remove unwanted situations from the level generation space by identifying transformations and rules that break properties. However, those rules may not be the root cause of the problem, which can originate earlier in the pipeline. In addition, it is hard for developers to analyze the history, since it is not clear where the branch points in the generation process are, and how alternatives would have played out. Finally, the expressive range of properties is currently still rather limited, and a formal semantics relating properties and histories is not yet defined.

1.7 CONCLUSION

This paper proposes two novel techniques that aim to improve the quality of grammar-based procedural level generation for grammars that work on tile maps. The first, is the Metric of Added Detail (MAD), a novel metric that indicates if a grammar rule adds or removes detail to a tile map. The second, is

Specification Analysis Reporting (SAnR), a technique that offers level property language for expressing level qualities. SAnR analyzes and reports how these properties evolve over time in level generation histories. We demonstrated the feasibility of MAD and SAnR with LudoScope Lite, a light-weight version of Ludoscope intended to study level quality.

Our preliminary evaluation shows that SAnR can express and analyze simple level properties, and that MAD is intuitive and raises flags for rules that remove detail. In addition, SAnR can be used in test automation. MAD and SAnR augment existing approaches by supporting gradually adding detail and analyzing level generation histories, which ultimately helps designers make better levels and level generators. Of course, LL is an academic research prototype that is not yet extensively validated in practice.

1.7.1 *Future Work*

Future work includes the following.

- **Validation.** A case study on Boulder Dash is current work. We also plan to study Unexplored to identify which additional SAnR property features are needed to express design intent more fully, e.g., better filters, validity ranges, and for shapes, paths and relative positions. We hope to identify bugs that would otherwise be hard or impossible to find.
- **Analyses.** Additional analyses on rule dependencies, and partial orderings may be identified of different rule orders generating the same levels, e.g., for increasing test coverage and level generation variety. For assessing the variety of generated content, existing metrics can be reused. For instance, Smith and Whitehead assess the expressive range of a generator by comparing metrics for linearity and leniency of platform levels [SW10].
- **Generation.** Here we use SAnR for analyzing level generation histories after they are generated. However, by integrating SAnR into a level generator we could also prune the search space and filter out potential unwanted levels before they are ever produced. A feasibility study can assess the impact on efficiency and scalability of this approach.
- **Formal semantics.** Reproducible dynamic analyses require a formal semantics for the execution of generative grammars, separate from tools and games that interpret them.
- **Parsing.** We observe that ambiguous grammars for parsing and level grammars generating the same tile map with different rule orderings are

related. Given a bugged tile map, how many different rule orderings can reproduce it? When changing the rules, can the new rules produce the tile map with a different generation history?

- **Debugging.** Debugging level grammars requires an interactive debugger, in particular for back in time debugging, exploring what-if scenarios and saving and replaying generated levels while testing new rules. Additional visualizations are needed to see how the generation space unfolds.

Acknowledgements

We thank Paul Klint, Anders Bouwer, Rafael Bidarra and the anonymous reviewers for their insightful comments that helped improve this paper. We thank Joris Dormans for collaborating with us and answering our many questions about the design of Ludoscope.

REFERENCES

[Com15] K. Compton. *Tracery*. 2015. URL: `http://tracery.io` (visited on May 19, 2018) (cit. on p. 3).

[CKM15] K. Compton, B. A. Kybartas, and M. Mateas. "Tracery: An Author-Focused Generative Text Tool". In: *Interactive Storytelling – Proceedings of the 8th International Conference on Interactive Digital Storytelling, ICIDS 2015, Copenhagen, Denmark, November 30–December 4, 2015*. Ed. by H. Schoenau-Fog, L. E. Bruni, S. Louchart, and S. Baceviciute. Vol. 9445. LNCS. Springer, 2015, pp. 154–161. ISBN: 978-3-319-27035-7. DOI: `10.1007/978-3-319-27036-4_14` (cit. on p. 3).

[Dor10] J. Dormans. "Adventures in Level Design: Generating Missions and Spaces for Action Adventure Games". In: *Proceedings of the 1st Workshop on Procedural Content Generation in Games, PCG 2010, Monterey, California, USA, June 18, 2010*. ACM, 2010, pp. 1–8. ISBN: 978-1-4503-0023-0. DOI: `10.1145/1814256.1814257` (cit. on p. 3).

[DB11] J. Dormans and S. Bakkes. "Generating Missions and Spaces for Adaptable Play Experiences". In: *IEEE Transactions on Computational Intelligence and AI in Games* 3.3 (Sept. 2011), pp. 216–228. ISSN: 1943-068X. DOI: `10.1109/TCIAIG.2011.2149523` (cit. on p. 3).

[EVV+13] S. Erdweg, T. Van Der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, et al. "The State of the Art in Language Workbenches – Conclusions from the Language Workbench Challenge". In: *Software Language Engineering – Proceedings of the 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26–28, 2013*. Ed. by M. Erwig, R. F. Paige, and E. Van Wyk. Vol. 8225. LNCS. Springer, 2013, pp. 197–217. ISBN: 978-3-319-02653-4. DOI: `10.1007/978-3-319-02654-1_11` (cit. on p. 13).

[HKV07] I. Heitlager, T. Kuipers, and J. Visser. "A Practical Model for Measuring Maintainability". In: *Proceedings of the 6th International Conference on the Quality of Information and Communications Technology, QUATIC 2007, Lisbon, Portugal, September 12–14, 2007*. Ed. by R. J. Machado, F. B. e Abreu, and P. R. da Cunha. IEEE, 2007, pp. 30–39. ISBN: 0-7695-2948-8. DOI: `10.1109/QUATIC.2007.8` (cit. on p. 9).

[KBB15] D. Karavolos, A. Bouwer, and R. Bidarra. "Mixed-Initiative Design of Game Levels: Integrating Mission and Space into Level Generation". In: *Proceedings of the 10th International Conference on the Foundations of Digital Games, FDG 2015, Pacific Grove, CA, USA, June 22–25, 2015*. Ed. by J. P. Zagal, E. MacCallum-Stewart, and J. Togelius. Society for the Advancement of the Science of Digital Games, 2015 (cit. on p. 3).

[KvdSV09] P. Klint, T. van der Storm, and J. J. Vinju. "Rascal: A Domain Specific Language for Source Code Analysis and Manipulation". In: *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009, Edmonton, Alberta, Canada, September 20–21, 2009*. IEEE, 2009, pp. 168–177. ISBN: 978-0-7695-3793-1. DOI: `10.1109/SCAM.2009.28` (cit. on p. 13).

[Läm18] R. Lämmel. *Software Languages: Syntax, Semantics, and Metaprogramming*. Springer, 2018. ISBN: 978-3-319-90798-7. DOI: `10.1007/978-3-319-90800-7`. URL: `http://www.softlang.org/book` (cit. on p. 11).

[Lav15] S. Lavelle. *PuzzleScript*. 2015. URL: `https://www.puzzlescript.net` (visited on May 20, 2018) (cit. on p. 3).

[MWD+05] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri. "Challenges in Software Evolution". In: *Proceedings of the 8th International Workshop on Principles of Software Evolution, IWPSE 2005, Lisbon, Portugal, September 5–7, 2005*. IEEE, 2005, pp. 13–22. ISBN: 0-7695-2349-8. DOI: `10.1109/IWPSE.2005.7` (cit. on p. 8).

[STN16] N. Shaker, J. Togelius, and M. J. Nelson. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Computational Synthesis and Creative Systems. Springer, 2016. ISBN: 978-3-319-42714-0. DOI: 10.1007/978-3-319-42716-4. URL: http://pcgbook.com (cit. on p. 3).

[SM11] A. M. Smith and M. Mateas. "Answer Set Programming for Procedural Content Generation: A Design Space Approach". In: *IEEE Transactions on Computational Intelligence and AI in Games* 3.3 (Sept. 2011), pp. 187–200. ISSN: 1943-068X. DOI: 10.1109/TCIAIG.2011.2158545 (cit. on p. 3).

[SW10] G. Smith and J. Whitehead. "Analyzing the Expressive Range of a Level Generator". In: *Proceedings of the 1st Workshop on Procedural Content Generation in Games, PCG 2010, Monterey, California, USA, June 18, 2010*. ACM, 2010, pp. 1–7. ISBN: 978-1-4503-0023-0. DOI: 10.1145/1814256.1814260 (cit. on p. 17).

[SMS+17] A. Summerville, J. R. H. Mariño, S. Snodgrass, S. Ontañón, and L. H. S. Lelis. "Understanding Mario: An Evaluation of Design Metrics for Platformers". In: *Proceedings of the International Conference on the Foundations of Digital Games, FDG 2017, Hyannis, MA, USA, August 14-17, 2017*. Ed. by S. Deterding, A. Canossa, C. Harteveld, J. Zhu, and M. Sicart. ACM, 2017, pp. 1–10. ISBN: 978-1-4503-5319-9. DOI: 10.1145/3102071.3102080 (cit. on p. 3).

[vdLLB13] R. van der Linden, R. Lopes, and R. Bidarra. "Designing Procedurally Generated Levels". In: *Proceedings of the the 2nd workshop on Artificial Intelligence in the Game Design Process*. AAAI, 2013 (cit. on p. 3).

[vdLLB14] R. van der Linden, R. Lopes, and R. Bidarra. "Procedural Generation of Dungeons". In: *IEEE Transactions on Computational Intelligence and AI in Games* 6.1 (Mar. 2014), pp. 78–89. ISSN: 1943-068X. DOI: 10.1109/TCIAIG.2013.2290371 (cit. on p. 3).

[vDKV00] A. van Deursen, P. Klint, and J. Visser. "Domain-Specific Languages: An Annotated Bibliography". In: *ACM SIGPLAN Notices* 35.6 (June 2000), pp. 26–36. ISSN: 0362-1340. DOI: 10.1145/352029.352035 (cit. on p. 11).

[vRH18] R. van Rozen and Q. Heijn. "Measuring Quality of Grammars for Procedural Level Generation". In: *Proceedings of the 13th International Conference on Foundations of Digital Games, FDG 2018, as part of the 9th Workshop on Procedural Content Generation, PCG 2018, Malmö, Sweden, August 7–10, 2018*. Ed. by S. Dahlskog, S. Deterding, J. Font, M. Khandaker, C. M. Olsson, S. Risi, and C. Salge. ACM, 2018, pp. 1–8. ISBN: 978-1-4503-6571-0. DOI: 10.1145/3235765.3235821 (cit. on p. 1).

[YT18] G. N. Yannakakis and J. Togelius. *Artificial Intelligence and Games*. Springer, 2018. ISBN: 978-3-319-63518-7. DOI: 10.1007/978-3-319-63519-4. URL: http://gameaibook.org (cit. on p. 3).