ScriptButler: Leveraging Meta-programming Principles to Facilitate the Software Evolution of Digital Games

Clement Julia
clement.julia@student.uva.nl

January 31, 2022, 67 pages

Academic supervisor:	Riemer van Rozen, R.A.van.Rozen@cwi.nl
Second Reader:	Ana Oprescu, a.m.oprescu@uva.nl
Third Reader:	Anders Bouwer, a.j.bouwer@hva.nl
Research group:	Centrum Wiskunde & Informatica, Software Analysis & Transformation, https://www.cwi.nl/research/groups/software-analysis-and-transformation

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Abstract

The digital game industry is a thriving multi-billion dollar industry. Creating interesting and interactive games is a complex process with two major parts: game design and game development. Game design is the art of applying design to create games on a conceptual level. Game development is the process of bringing the game design to life. Part of the game design process is the creation of game mechanics, rules that define how the player interacts with the game. Playtesting is the process of evaluating the impact of these rules on the player experience, with the goal being a net positive impact. However, playtesting has a significant resource and time cost associated with it, as such game designers must sometimes make decisions when evolving their game without the necessary knowledge of the impact on the player experience.

We approach the study of this problem from a meta-programming perspective. We aim to empower game designers with tools and techniques that give feedback about the quality of the games. In particular, we study how dynamic analyses can provide live feedback about a game's rules. We focus our efforts on a concrete problem by studying PuzzleScript and evaluating our approach on a set of published games written using that engine.

We formalize the design of PuzzleScript and implement a redesign of the technical implementation using Rascal, a language workbench designed to facilitate meta-programming. This more extensible and maintainable prototype implementation of PuzzleScript aids us in our initial goal and in future PuzzleScript research. We then extend our implementation with our system of game mechanics analysis and test games for game mechanic errors. Finally, we validate our approach on a set of real-world published games and modify games to test for gameplay decay, the fall in gameplay quality as a result of evolution in-game mechanics.

Contents

Chapter 1

Introduction

Digital gaming is a considerable market, making up 2.7 billion users, worth over $300 billion according to a report by Accenture[1]. Game design is a vital part of the game development process that focuses on the theoretical creation of the game. Game designers aim to create games that are both interesting and interactive aimed at experiences such as entertainment or education.

A key part of the game design is the creation of *game mechanics*, sets of rules that define how the player can interact with the game. Gameplay emerges from game mechanics[1]. The quality of gameplay is defined by the interaction between game mechanics and the player. Since the quality of gameplay is directly tied with the player experience it varies from player based on their personal preferences. Validating the impact of game mechanics on a subjective player experience requires *extensive playtesting*, both generally and within specific control groups. This process is costly in terms of time and money[2] and is subject to human error and inconsistencies.

When game requirements inevitably change, the designers must evolve game mechanics to meet them. To analyze the impact of these changes, designers use techniques like prototyping and playtesting, but gameplay quality degradation my happen nonetheless. When designers attempt to improve a game's rules they take a risk that can end up negatively impacting their game. This problem is exacerbated by the little time designers have for improving games.

Our aim is to empower designers with tools that automate the game design process and enable them to evolve games quickly. To create these tools, we explore the relationship between rules and gameplay by studying PuzzleScript. PuzzleScript is a popular open-source tiny online game engine[3] written in JavaScript by Stephen Lavelle. We analyze the source code of many games created with PuzzleScript to answer our research questions. We explore how languages, tools, and techniques can help designers understand the impact of the changes they make in order to improve gameplay. We study the effect of iterative changes to the source code on the game mechanics.

We automate the game design process by providing rapid live feedback on the state of the game mechanics and the impact of changes. We combine theories from two key areas: automated game design[2, 4–7] and software evolution[8, 9]. PuzzleScript's official implementation is aimed at the portability of games and performance. However, these advantages come at the cost of reduced maintainability and extensibility, this makes it complicated to perform automated dynamic analysis on the code. We reverse engineer PuzzleScript in order to create a formal design document that will help us better understand how we can support developers that use the language. We then re-implement that design and optimize it for our research purposes using Rascal, a meta-programming language and language workbench. Finally, we extend our prototype implementation to perform dynamic analysis on PuzzleScript games and allow a better understanding of the impact of game evolution.

1.1 Problem statement

We now elaborate on the research problem.

Playtesting has a large impact on the quality of the gameplay. Game mechanics that are thoroughly playtested are less likely to negatively impact the quality of the gameplay. Since game mechanics indirectly affect each other, this means that changes to a game mechanic can have unintended side effects when combined with others. As such, game designers must playtest not only how a mechanic can interact

[1]`https://newsroom.accenture.com/news/global-gaming-industry-value-now-exceeds-300-billion-new-accenture-report-finds.htm`

with the game but also how it interact with every other mechanic. With every new mechanic, the amount of playtesting required grows exponentially.

The process of validating game mechanics through QA testing is resource-intensive and time-consuming. It consists of testers, either human or AI-based, playing through the game and reporting on their impressions, this feedback can then be used by game designers to further evolve the game. Because of the time required to playtest, game designers have to make choices on what part of the game they need feedback on. The priority usually goes to the new content, as such, existing content is not always regression tested when new mechanics are introduced[2]

The time constraint exists in both large companies that have financial goals to meet to show growth and in small companies that depend on the revenue to continue functioning. Any game has a labor cost associated with it and most game designers seek to break even. As a result, most game design projects have a sort of deadline, either 'eyeballed' or specifically tailored for the costs of the game to meet the expected revenue. This deadline is the reason that playtesting cannot always be conducted fully.

On a smaller scale, the time constraint also restricts how much game designers can explore the design space of their game. Design space is defined by Schubert in his blog "Understanding Design Space"[2] as "he canvas that the designer can paint on". Understanding the size of that 'canvas' and its growth potential are key to understanding what a game can achieve in terms of content. Game mechanics define the size of the design space. A match-three game will have a much more limited design space than a grand strategy civilization game.

The issue of incomplete playtesting has a significant impact on the quality of games as they begin to shift to a "games as a service" (GaaS) model where game mechanics are intentionally meant to evolve after the initial release. Small digital game companies are especially affected by this as they do not always have the resources or foresight to see the importance of gameplay quality assurance measures[10]. More and more game designers are beginning to rely on their players to test interactions for them and release moderately tested patches that may or may not fix the issues reported[11]. To resolve this we propose a system of solutions based on meta-programming that allow us to conduct Automated Game Design (AGD) through the automation of rule validation. We approach this goal by studying PuzzleScript and the relations between the source code of PuzzleScript, which has an explicit notation for game mechanics and how that code actually impacts gameplay.

This relation between code and game mechanics allows us to test our approach on a concrete problem. However, we run into issues when trying to analyze PuzzleScript:

- PuzzleScript has no formal design document. This makes it challenging to study it for ways to apply our dynamic analysis.
- PuzzleScript's official JavaScript implementation is focused on portability and performance. This makes it challenging to process and analyze the source code.

These two issues are significant obstacles to our project and we cannot progress without either resolving them or selecting another language. We choose to continue forward with PuzzleScript because of its popularity and the relation mentioned above. Additionally, fixing these issues should make it easier to perform research using PuzzleScript as a concrete problem by extending a general purpose tool to fit a specific problem.

1.1.1 Research questions

Our goal is to help reduce the resources required to playtest game mechanics by allowing for automated playtesting to be conducted. We hypothesize that automatically playtesting for simple errors in the game mechanics will allow game designers to focus human playtesting on complex issues. This 'automated playtesting' must be relatively fast so that game designers do not feel reluctant to run it but thorough enough to still pick up on any major flaws in the game mechanic.

We seek to create tools and techniques that will help developers automatically playtest their games not only when they are first created but also as they evolve. The impact game mechanics have on the user experience changes over time, both as a result of direct and indirect changes. As such, we aim to use of meta-programming tools and techniques to: 1) empower developers in their effort to evolve their games; 2) support popular established game engines; 3) mitigate the negative impacts of game mechanics on player experience.

The tools we provide aim to leverage meta-programming principles. However, this goal is very general and therefore hard to concretely apply. We resolve this by specifically studying PuzzleScript, which leads

[2]`https://www.zenofdesign.com/understanding-design-space/`

us to formulate the following questions:

RQ1: How can meta-programming tools and techniques be used to empower PuzzleScript game designers and allow them to freely explore the design space of their game?

- **RQ1.1**: How can PuzzleScript be used for studying game mechanics?
- **RQ1.2**: How can game designers mitigate the impact of iterative game design on the quality of the game mechanics in PuzzleScript games?

One of our goals in this project is to not only test our theory of simple automated mechanics testing but also to provide software contributions to the field of automated game research in PuzzleScript. We aim to make it easier to conduct research into PuzzleScript by creating general-use tools that can be extended to fit individual projects.

1.1.2 Research method

We conduct design science research[12]. This means that we go through an iterative and cyclic process composed of three steps. We gather problems from our environment, propose solutions supported by knowledge bases, and then validate them with field tests to see if they match our requirements. This is a mixed method that is combined with action research[13]. Our action in this case is the tool we propose and we analyze the impact it has on the environment.

We study the existing PuzzleScript's design to understand how code relates to gameplay. We survey common discussion areas to understand what PuzzleScript game designers' needs. We then implement this into our tool to conduct an analysis of the game and provide feedback based on that. We validate our tool by testing it on published games and mimicking software evolution. We can then observe what kind of feedback the tool provides.

1.2 Contributions

The main contribution of this thesis is a prototype tool called ScriptButler, for evolving, analyzing, and running PuzzleScript games. ScriptButler is not intended as a be-all/end-all solution but rather a general-purpose tool that can be extended to meet project-specific needs. At its base, the tool offers an interface to parse, process, and statically analyze PuzzleScript game with the possibility for extension based on the user needs. This can be implementing changes to PuzzleScript design or simply adding new restrictions to enhance the live feedback of a specific project. The more advanced part of the tool provides an engine through which the game is compiled and can then be run. However, Rascal does not have a method to gather user input as such the tool provides an interface for interacting with the game, but a user may also choose to implement their own. An earlier version of this work was previously published as R. van Rozen *et al.*, "First-Person Real-Time Collaborative Meta-Programming Adventures," 2021, To appear.

The tool also provides dynamic analysis for PuzzleScript games. The games are compiled and then run by the engine in the background to generate helpful information on how the various mechanics interact. This dynamic analysis also uses a system of simple solutions to rapidly test out any potential issues with the game. Finally, the tool provides IDE integration within Eclipse for PuzzleScript games. This integration has syntax coloring, shows messages by the checkers, and provides other utilities such as an outline and code folding. We also contribute knowledge to the Automated Game Design area through our theory of rapid feedback allowing for better quality of games. A theory that we test using our tool on PuzzleScript games.

1.3 Outline

In Chapter 2, we elaborate on the research that supports our thesis and go into the related work that we used as a basis. This is different from the Chapter 7 where we discuss works that relate to our work but that are not necessarily used directly to support our work. Rather, that chapter is intended to display what kind of research is being done into PuzzleScript and Automated Game Design.

In Chapter 3, we present the design document we obtained by reverse-engineering PuzzleScript's design. We briefly discuss the design of PuzzleScript and its official JavaScript implementation.

In Chapter 4, we explain the design of our tool and the work that led up to those decisions based on the document presented last chapter. This chapter consists of two sections: re-designing the technical implementation of PuzzleScript and extending our implementation with dynamic analysis to support game designers.

In Chapter 5, we discuss the case study we conduct to validate our tool. We layout the game we choose, with the reason we choose it. Then we go into the method we use for conducting the case study. Finally, we discuss the results of our case studies and whether they successfully validate our tool.

Chapter 6, we reflect on the main results of this thesis. We discuss how our work answers our research questions. We also discuss the threats to validity and how we mitigated them.

Finally, in Chapter 8, we summarize our contributions, give a few concluding remarks, and discuss future work.

Chapter 2

Background

This chapter introduces fundamental concepts and elaborates on the terminology introduced in the previous chapter. This chapter serves to relate practical challenges to research challenges and explains how the two can support each other.

2.1 Game Design

Game design is part of the game development process. Game designers create the game on a conceptual level, including declaring a rule-set which becomes the game mechanics. Game developers implement the rule-set as software to provide players with the affordances they define. Game design is an iterative process. Game mechanics evolve as the requirements for the game change, and as such playtesting those mechanics is required even after their final implementation.

The goal behind game design is to create a certain user experience, whether entertainment-based or educational. The user experience that game mechanics create within the context of a game is called gameplay. Gameplay is an elusive concept that researchers have long tried to specifically define[15]. Crawford[16] broadly defines gameplay as the relation between challenge and player capabilities. In this context, good gameplay is gameplay that seeks to maintain those two notions in balance. Influencing the player's capabilities is complicated but the game designer has complete control over the challenge that the game presents. Game mechanics created by the designers serve to shape that goal. Game mechanics are playtested on whether or not they create that experience. Game designers are already supported by many types of tools and techniques. They can create paper prototypes, rely on player metrics or documentation to judge the impact of mechanics before implementation.

The quality of the game depends on many different factors. These factors include the visual look of the game, its narrative, the performance, and the one that interests us, the game mechanics. Validating the mechanics is done by playtesting. Playtesting can be conducted manually by humans or by AI agents and it can be conducted before or after the mechanics are implemented with the use of techniques like paper prototypes.[17]

Playtesting with AI agents is a more recent innovation that aims to make playtesting faster and more reliable[18]. However, it presents its own set of challenges. As an AI-agent learns much more slowly than a human player since human players can draw on their previous experiences, whereas AI agents usually start from zero. Additionally, while AI can test the mechanics in complex ways, they do not experience them the same way that human players do. In this thesis, we focus specifically on addressing the issues caused by human playtesting. AI-based playtesting has its own set of issues and benefits that fall out of the scope of this project. A workshop study by the HCI Institute[19] has found that student game developers struggled to integrate the playtesting phase within the iterative design process. Instead, the students would frequently leave playtesting to be closer to the release date, often revealing flaws too late. Similar issues occur within the industry where playtesting reveals major issues with certain game mechanics but due to time constraints, the game is released anyways.

Game mechanics interact with each other in various ways, they can complement, oppose, be neutral, be pre-requisite and many other ways. Making sure that mechanics interact in the right way is a question of trade-off. When game designers evolve game mechanics to make them more interesting this usually comes at the cost of opposing game mechanics while complementary mechanics also become indirectly more interesting. Making sure game mechanics exist in an equilibrium is called *balancing.* Balancing game mechanics is a complex part of the process that involves a lot more than just changing mathematical values. The aim of that process is ensuring that different strategies remain competitive even as the rules change and as a result, creating the experience the game designer desires. Schell[20] identifies 13 criteria that contribute to game mechanic balance. These criteria focus on creating an interesting and entertaining experience by balancing challenges and rewards.

Playtesting validates these criteria, but we aim to provide a solution that can provide rapid feedback and as such, we need a quick way to check if a game mechanic meets some of these criteria. To achieve this we reverse the problem, instead of checking if a game matches any of these criteria, we instead focus on checking if any of the game mechanics really go against the criteria. For example, one way to check if a game mechanic is interesting is to solve a level that makes use of it, either manually or using an AI player. This is time-consuming, so instead, we check to see if the game mechanic does not accidentally create an uninteresting solution such as being able to resolve the challenge by walking in a single direction. We seek to identify other such uninteresting solutions and make it possible to conduct these analysis automatically without the intervention of human playtesters.

2.2 Automated Game Design

Automated game design (AGD) is a research area that studies how software automation can help in speeding up and improving iterative game design. AGD shifts the designer's efforts away from error-prone and repetitive tasks. This allows designers to focus on the creative aspect of the process and explore the design space of their game. AGD is often researched from an AI perspective, which means a focus on the development and application of algorithms. We discuss this perspective but our thesis remains focused on AGD from a meta-programming perspective. The meta-programming perspective is a focus on the engineering of user tools for code analysis and transformation. Procedural content generation (PCG) is defined as *the algorithmic creation of game content with limited or indirect user input*[21]. The content in this case is anything that is contained within the game: levels, rules, entities, sounds, sprites, etc. This definition does not include any of the technical sides of the game such as the engine or the source code. Specifically for this thesis, we are interested in *'interaction-bound content'*, content that affects how the player interacts with the game.

Interaction-bound content in the context of PuzzleScript is Objects, Rules, Win Conditions and Levels. This can be summarized as the content that affects the player interactions, also referred to as game mechanics. We make this distinction as opposed to other content such as visuals, sounds, or narrative which do not directly contribute to the interactivity of the game. This is not to say that these elements do not contribute to the quality of the game but simply that it is not in the scope of this thesis to understand how audio-visual content impacts game design. In this thesis, we focus specifically on the rules of a game. Rules provide the player with affordances such as walking, jumping, teleporting, or the ability to push other objects. This is similar to moving a piece in chess or playing a card in Uno, what rules allow players to do differ from game to game. Representing rules as content enables us to change the player experience. Rules can be modified manually by a designer or automatically using a tool that embeds an algorithm.

Game quality is limited by the number of design iterations, producing high-quality games requires a reduction of the iteration time. Overcoming that limitation is an important objective of AGD. We can leverage live programming to provide live (immediate and continous) feedback on the quality of game mechanics to achieve this goal. Live programming provides game designers with feedback without the need for an independent testing phase. There are no 'breaks' in the designing process, only minor adjustments[4]. However, the best way to validate the user experience is still playtesting, so instead of trying to replace playtesters, we simply apply AGD theory to reduce their workload allowing more focus on the complex parts of playtesting.

PCG is not the focus of this thesis but an important part of AGD research and as such, we felt it was necessary to touch on the matter. This thesis focuses more on the use of AGD for automated playtesting, which is more commonly done through the use of AI agents. PCG tools address different challenges based on what they seek to generate. PCG tools aim to generate new levels to leverage existing game mechanics to create an interesting player experience, while others seek to generate the game mechanics themselves. PCG usually requires a level of user input, when the tool and the designer take turns modifying the rules it is called *'mixed-initiative'* design[22]. The designer provides settings or inputs that are interpreted by the tool to generate parts of the game, the designer can then adjust the setting or modify the generated content. Mixed-initiative design tools help designers generate much more specific game content that is more in line with their goals without needing to be trained like an AI.

Using automation such as PCG can have large benefits on the productivity but automation also has

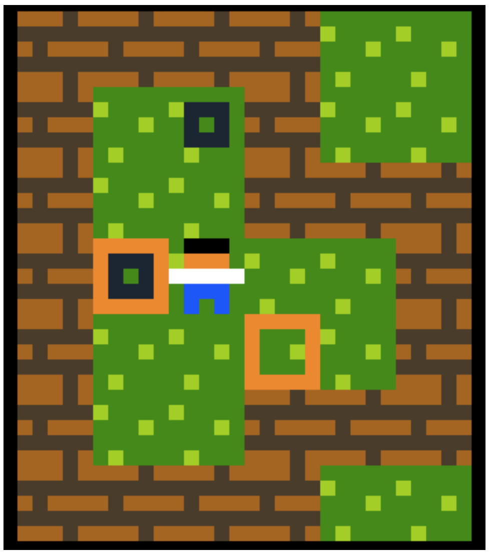


Figure 2.1: A simple PuzzleScript game

a cost. This is especially important as the number of person-months that it takes to create a successful commercial game has increased consistently[23]. As such, game designers are seeking to speed up part through the use of PCG. This can mean a significant reduction in the cost of game development, making it much easier for smaller teams of developers to hit the "successful" milestone. However, automation has a cost, as automation systems become more complex, game designers begin to get pushed out of the game development process.

PCG is a set of techniques that can aid in the development of interesting games but it can also be integrated directly into the game to generate procedural content even as the game is being played. Games leverage PCG to generate randomized levels that greatly increase the replay value of the game. For the player, it means that no two playthroughs are the same even if they may be more or less similar depending on the complexity of the generator. One of the most famous games to use PCG in that way is Minecraft[1], an open-world sandbox. Minecraft generates a world in which the player interacts procedurally, placing rivers, forests, mountains, and caves using a complex algorithm. Such types of PCG usually require a small amount of user input to ensure true randomness in their generation.

Studying automated game design is a complex task. Game are huge projects with many components. To study automated game design, we need a concrete problem that is simple enough to study but still fulfills the technical requirements of a game engine. We select PuzzleScript and explain our reasoning in the next section.

2.3 PuzzleScript

Next, we give a limited overview of related work on PuzzleScript that clarifies what is missing in the state of the art for answering our research questions. PuzzleScript is a tiny online online game engine that was identified in a mapping study by van Rozen[4].

PuzzleScript is an open-source HTML5 puzzle game engine written using JavaScript by Stephen Lavelle. A PuzzleScript game is made up of 5x5 sprites called Objects, which are used to construct tilemaps called Levels. The player uses the affordances provided by the game mechanics to alter the level with the goal of fulfilling an arbitrary victory condition. Game mechanics are defined using patterns that match a level horizontally or vertically called Rules. If a pattern matches, a replacement pattern is applied, transforming the level.

PuzzleScript closely relates game design and game development through its Rule system. Many game mechanics in PuzzleScript games are literally defined with a single line. This makes it considerably easier to understand how alterations in the game code can affect game mechanics and gameplay. Such an example can be seen in Figure 2.2 where a single line allows the player to push an entity in any direction. Reversing the arrow would allow the player to pull the entity, the entity can also be a reference to multiple entities allowing for a single line of code to affect multiple interactions.

[1]`https://www.minecraft.net/`

```
[ >  Player | Crate ] -> [  >  Player | > Crate  ]
```

Figure 2.2: Code for push mechanics

At its core, this rule system is a simple match and replace pattern. The designer defines a pattern on the left side that is either matched horizontally or vertically and then replaced with the pattern on the right side. In Figure 2.2, the patterns can be translated to "if a player tries to move into an adjacent crate, then move both entities in that direction". This simple core makes PuzzleScript very approachable by game designers of all levels and its flexibility still allows them to make relatively complex games. One of the contributions from the bachelor thesis by Vermeulen[24] is a prototype grammar to parse PuzzleScript. This provides support for generating code based on desired player affordances. The prototype parses a significant amount of PuzzleScript rules and serves as the foundation for our own grammar. As a prototype, it fulfills the role of parsing simple PuzzleScript games but it is not a complete grammar. Many of the finer details and corner cases of PuzzleScript's grammar are missing. We use the grammar and design formalization presented in the thesis as a stepping stone.

PuzzleScript's popularity as a game engine and its appeal as a case for researching game design is shown in several papers[25][3][26][27]. The papers provide a proof of PuzzleScript's popularity as a game engine with raw statistics and provide motivation arguments for its use within an automated game design perspective. The papers also discuss procedural content generation within PuzzleScript. The common thread between all these papers is that they demonstrate how the concepts we touch one can be used within PuzzleScript and how PuzzleScript is a legitimate case for realistic game research. In addition to Vermeulen's thesis, we also found other papers with interest in Procedural Content Generation. A paper by Krishnan[28] is an example of people studying PuzzleScript games to research game design.

Several projects that seek to extend or reimplement PuzzleScript in a different language already exist. They provided insights into the processes of the original JavaScript implementation and as such we felt it was important to give a brief overview. The PuzzleScript repository has 127 forks[2] at the time of writing, of those only a few are actively developed as extensions to PuzzleScript, the rest are either inactive or developed with the intent to merge back into the original with a pull request.

Pattern-Script[29] is a major extension of the original PuzzleScript engine by Clement Sparrow. It aims to add a plethora of new features[3] and support for game designers. By providing additional keywords, the engine allows designers to use shortcuts for the creation of tilemaps and grants them a more granular control on the rule matching process. The extension does not modify the engine and does not extend the behavior of rules, instead it allows designers to restrict the conditions of the rules more tightly. This seems to go along our theory that extending PuzzleScript is hard because the engine's complexity.

This fork by Joe Osborn[30] claims to integrate game analysis but does not seem to have got any commits since the fork was made. Another example is the fork by Rikki Prince[31] that attempts to make it possible to generate PuzzleScript games using grammar but does not provide working instructions.

The implementations we found have similar goals to our own. However, unlike our approach, these implementations are all ports of the original PuzzleScript implementation with minor changes. We instead create a complete redesign of PuzzleScript that permits us to create algorithms and tools that can analyze its structure and semantic. This redesign is necessary for answering our research question and the reason we cannot use existing implementations for this project.

- C[32]: Released, runs simple games.
- Rust[33]: Released, feature-complete as far as we are aware, needs games to be formatted in a special JSON-based structure
- C++[34]: Released, missing features

Re-implementations provide insight into how other languages solve the technical complexity of PuzzleScript's rule system. The ones studied here all follow a similar approach to the original JavaScript implementation: hand-crafted parsers that statically check the game at the same time and tightly coupled phases.

Finally, we have works that extend the PuzzleScript implementation in small ways[35][36]. These forks are usually created by programming-inclined PuzzleScript game designers, which have a specific

[2]https://github.com/increpare/PuzzleScript/network/members
[3]https://github.com/ClementSparrow/Pattern-Script/wiki

need that the current implementation does not meet. The additions and modifications they perform provide insight into the needs of PuzzleScript developers and designers.

2.4 Meta-programming

Meta-programming is a programming technique that creates *meta-programs*. Normal programs process and transform data, meta-programs process and transform other programs. Meta-programming techniques and language workbenches help programmers in the activities of meta-programming. Domain specific languages (DSL) are programming languages that perform a very specific set of tasks, compared to general-purpose languages which can be used for a wide range of tasks. PuzzleScript is an external DSL with the specific purpose of creating top-down or 2D puzzle games. One would be unable to use PuzzleScript to create websites or calculate financial data, that is outside the domain.

PuzzleScript makes use of meta-programming in its JavaScript implementation to transform rules into JavaScript functions. In this scenario, we see one of the limitations of this specific brand of meta-programming: code obfuscation. A significant part of the code that processes rules of a PuzzleScript game does not exist before runtime, this makes it much harder to maintain and extend. This decision hurts the readability and searchability of the codebase in that a programmer must first compile a game before being able to observe how the rules transform into code. Only having that complete overview during runtime makes it harder for programmers to identify and locate bugs.

Language workbenches can help to facilitate meta-programming. We use Rascal, a meta-programming language and language workbench to parse, run and analyze PuzzleScript games. We selected Rascal from a list of meta-programming languages and language workbenches identified by van Rozen[4] and in a paper on language workbenches for game development[37]. Domain-specific languages have been successfully applied in many domains but their benefits to game development are not yet clear. According to this paper by Klint and van Rozen[37], this is due to the game domain being very diffuse. It is hard to analyze a moving domain where modeling and software reused are limited. The existing approach is game engines which provide the ability for software reuse and optimization, but even these have very specific cases and no silver bullet exists.

In this thesis on the state of the art in workbenches, several other language workbenches are identified. It also identifies several features deemed essential in language workbenches and compares the identified list with the workbenches themselves. From that, we can see Rascal is not the most feature-complete of the workbenches presented, but it provides the ability to make up that difference programmatically.

Rascal is an extensible meta-programming language and IDE for source code analysis and transformation with the goal to merge the SCAM (source code analysis and manipulation) domain into a single language[38]. It provides parsing in the form of its Syntax Definition Formalism that allows developers to create a literal representation of the programs they are trying to parse through the use of lexical and symbols. These representations can then be converted by *imploding* them into equivalent data structures. Rascal also offers the possibility of processing concrete syntax which does not require implosion but has other downsides. Rascal also implements the concept of *visiting*, which allows us to traverse an arbitrarily complex subject and apply a number of cases to all its nodes. We can use this to enrich a parse tree without having to modify its structure by editing or replacing individual nodes.

Finally, Rascal presents a concept called "Abstract Patterns" which allows us to match data structures using regular patterns similar to how one might match text with regular expressions. This concept is very similar to the way that PuzzleScript Rules work on the surface and as such we believe that it is possible to use it to create a more human-readable version of compiled Rules. In the current implementation, Rules are compiled into functions, which adds a layer of complexity as part of the codebase that can only be observed during runtime.

2.5 Reverse Engineering

Our project aims to create a version of PuzzleScript that is well suited for game design research, extension and maintainability. Unfortunately, we cannot simply port the existing JavaScript implementation. We need access to PuzzleScript's design to create our own implementation. Since the design of PuzzleScript is not fully documented, we have to reverse engineer the existing implementation.

Reverse engineering is the method of attempting to understand how a program works by deductive guesses based on the observation of the program's behaviors[39]. Researchers who reverse engineer programs do not always have access to the full source code or architecture documents and as such must rely on their observations to guess the original intentions of the developers. There are two main aspects to reverse engineering: redocumentation and design recovery. Redocumentation is the process of creating a new representation of a program and design recovery is the process of using deduction in relation to one's experience with the product to recover its design[40].

Reverse engineering is commonly used in cases where the source code is inaccessible, however, that is not a requirement. Reverse engineering can also be used when the source code is not properly documented or densely structured to reconstruct the design of the application. This method is not without flaws, it requires a good understanding of the language the program is written in. Knowledge gaps can sometimes be filled by observing the behavior of the application from the user perspective. Reverse engineering is very time consuming, but it is sometimes the only option to be able to separate the design from the implementation.

The process of design recovery involves connecting the codebase to user-facing endpoints to understand how they fit together. Observing the behavior of a program is a relatively safe way of ensuring that the correct intent is understood, but it depends heavily on managing to trigger every corner case. Complementing this is the method of going through the codebase[41]. This method presents its own set of issues depending on how strict the language is with its typing.

The end product of design recovery is a design document based on the observed behaviors. This document is made up of assumptions the person conducting the reverse engineering made therefore it is not a pure version of the original design. Purifying that version to be closer to the original is a task that requires a lot of work but can be supported by specialized techniques such as model-based reverse engineering[42].

Chapter 3

Reverse Engineering PuzzleScript

In this chapter, we take steps to extract PuzzleScript's design from its implementation through reverse-engineering the JavaScript implementation. As we previously mentioned, the reason we need to reverse engineer PuzzleScript is that the existing implementations are not suitable for our research purposes. Therefore, we create our own. However, PuzzleScript does not have a complete design document we can use[1]. We need a design document because we do not want our implementation to have similar pitfalls as the official implementation.

3.1 Approach

We create a comprehensive PuzzleScript design document by performing design recovery on the existing implementation by Lavelle. Design recovery in our case involves going through the codebase and observing the behavior of the implementation. We have access to the codebase through the public GitHub repository and we can access the deployed implementation on the PuzzleScript website[2].

Investigating PuzzleScript's codebase presents challenges. JavaScript is dynamically typed, which means that the type of variables and function returns is not guaranteed. For instance, the code `a + b` could be adding two integers together or concatenating two strings. The only way of guaranteeing either outcome is to run the code and observe it. We can use context clues such as comments and variable names in the early stages but validation requires the code to be observed during runtime. Comments are sparse in PuzzleScript's codebase and the code density is high. As a result, reverse engineering the design of the codebase is challenging. Recovering the design also relies on the software engineer knowledge of that specific language. We complement our approach by observing the program during runtime. We do this by subjecting the system under study to various inputs to observe its behavior. We combine both approaches to create the design document.

Part of PuzzleScript's design is documented in the bachelor thesis by Vermuelen[24]. Their thesis breaks down the design of PuzzleScript and implements that design using Rascal's Syntax Definition. This is similar to our approach, however, their design document and implementation are incomplete. Their goal is to generate code from keywords and their implementation serves that purpose. Our goal is to parse every PuzzleScript game in existence using a new grammar that can be reused in future prototypes. As such, we need a design document that is more comprehensive and documents PuzzleScript's many corner cases.

3.2 PuzzleScript's Design

In this section, we describe the results of our reverse engineering. We frequently reference the PuzzleScript user manual[3]. The user documentation focuses on the specifics of games. We instead focus on the general syntax structure of PuzzleScript and on the contextual constraint that can be used for creating a static analyzer. We aim to provide a document that makes it easier to parse PuzzleScript while the user guide provides specific implementation details.

[1]`https://groups.google.com/g/puzzlescript/c/UBe9M8QP-rk`
[2]`https://www.puzzlescript.net/editor.html`
[3]`https://www.puzzlescript.net/Documentation/prelude.html`

3.2.1 General

A PuzzleScript file stores the contents of exactly one *game*. Anything within that file is considered content for that game and that game only. There exists no way of splitting a game into multiple files or storing more than one game into a single file.

A *game* consists of a *prelude* and multiple *sections*. Sections consist of a header which is the name of the section between two optional lines of equal (=) sign. Sections are delimited by either two headers or a header and the end of the file. In the JavaScript implementation, these sections must be in a specific order but from a design perspective, there is no inherent need for a specific order. However, "define before use" is a natural policy of programming and PuzzleScript may rely on that. The grammar of a section's content depends on the section itself. PuzzleScript is a context-sensitive grammar. The rules of syntax apply differently based on the context, in this case the context is the section. The order of the sections is as follows:

- Objects: Defines all objects to be used in the game
- Legend: Create shorthand links between symbols and objects to allow properties, aggregation of objects, and tilemap creation.
- Sounds: Defines handlers to play sounds when certain events happen
- Layers: How objects interact with each other movement-wise, whether they can stack or if they collide
- Rules: How objects interact, what happens when certain items are next to each other or if a player tries to move into a certain item
- WinConditions: Defines victory conditions for the level
- Levels: List of tilemaps and messages that represent the different rooms the player will explore

Comments in PuzzleScript are delimited by parenthesis (*()*) and can be placed anywhere. It is unclear whether that is a conscious decision or simply a byproduct of the fact that the engine strips all comments before compiling the game.

3.2.2 Prelude

A game's *Prelude* consists of all the lines before the first section. Each line consists of one or two tokens: a *keyword* and a *value*. These pairs allow the developer to modify how the engine works and the visual aspect of the game. Certain keywords do not require a value, they simply need to appear in the prelude to have an effect. Values can be a string (a list of tokens displayed literally), a numeral (int or float), or a member from an enumerator. A full list of possible prelude keywords and their values is documented in PuzzleScript's user guide[4].

3.2.3 Objects

Object are the building blocks of PuzzleScript's games. They make up everything the player can see and interact with. They are used in rules to define game mechanics, they are used in conditions to define how the player achieves victory and they are used in levels to define what kind of challenges they present.

An object consists of a *name*, an optional *legend*, a list of *colors* and an optional *sprite*. The name and the legend are on the same line, the color list on its own line, and the sprite is a 5x5 tilemap. Colors are either an HTML color code or a color keyword from the selected palette. The color palette available is decided by the developer in the prelude from a wide range. The sprite is a 5x5 tilemap of where the `.` represent a transparent pixel and numbers from `0-9` represent a colored pixel based on the index of the colored list. As can be seen in Figure 3.1, `0` represents blue because it is the first color in the list of colors defined for that object.

The name and legend of the object can be almost any character, although choosing certain characters can be an issue in other sections. For example, naming an object using any of the directional arrows is technically allowed. However, that makes it impossible to use the object's name in a rule, where the directional arrows are reserved keywords. In other cases, using reserved keywords for objects still works but confuses the syntax highlighter and results in code that is difficult to read and understand to the programmer.

[4]`https://www.puzzlescript.net/Documentation/prelude.html`

```
Player
black orange white blue
.000.
.111.
22222
.333.
.3.3.
```

Figure 3.1: An object

```
Obstacle = Crate or Wall
P = Player
O = Target And Crate
```

Figure 3.2: Three legends

3.2.4 Legend

Legend allow the developer to create shorthand references to one or more objects. A legend consists of a name and a sequence of references, separated by the equal (=) sign. The sequence of object references, either the object name or another legend, is separated by either 'and' or 'or'. Legend cannot mix separators and cannot reference a legend that makes use of the other separator. *Properties* are object references separated by 'or'. Properties are used to reference multiple objects in the context of rules, victory conditions, and sound. *Aggregates* are object references separated by 'and'. Aggregates represent objects that are stacked on top of each other. Aggregates are used to place multiple items on one pixel in Level tilemaps.

The name of the legend is subject to the same constraints as the name of an object. Aggregates and legends referencing a single object can be used on tilemaps as long as the legend name is only one character long. An example of the different combinations can be seen in Figure 3.2.

3.2.5 Sound

Sounds allow the game designer to define what sound to play in the case certain events happen. A sound consists of either a reference or a sound keyword (SFX[0-10]) followed by a list of conditions (moving left, stationary) and finally a sound seed. The editor generates a limited range of sound seeds for use within games. Examples of sounds are shown in Figure 3.3 and the possible conditions are documented in PuzzleScript's user documentation[5].

3.2.6 Collision Layers

Layers are lines of sequences that consist of either object reference or properties separated by either an optional command (,) or a white space. The purpose of the section is to define whether objects can stack or if they collide. All objects defined in the game must be referenced in a layer. This restriction exists so that the developer does not accidentally spawn an object without collision. Referencing an object in multiple layers is technically allowed but not recommended and it raises a warning. Ignoring

[5] https://www.puzzlescript.net/Documentation/sounds.html

```
player move up 142315
Player Move down 142313
Player Move right 142311
Crate Move 412312
Player CantMove up 41234
Crate CantMove 41234
Crate Create 41234123
CloseMessage 1241234
Sfx0 213424
Sfx3 213424
```

Figure 3.3: Combinations of keywords creating sound events

```
        Crate , Player , Wall
```

Figure 3.4: A Layer

```
        [ >  Player | Crate ] -> [  >  Player | > Crate  ]
        late [ Crate | Crate | Crate ] -> [ | |]
```

Figure 3.5: A few Rules

the warning can have unintended consequences such as layers being ignored, allowing for unintended movement. Figure 3.4 illustrates an example of a layer.

3.2.7 Rules

Rules are at the heart of PuzzleScript. A rule consists of a series of prefixes that affect the whole rule, a left-hand side *"pattern"* and a right-hand side *"replacement"*.

The left-hand side must contain at least one *rule part*, the right-hand side must contain 0 or exactly the same number of rule parts as the left-hand side. Rule parts start with a `[` and end with a `]`. Within a rule part, there are a certain number of *rule sections*, each rule part on the right must have the same number of sections as the equivalent rule part on the left. Sections of a rule part are separated by `|`. Each section consists of object references that can stack, and each object can have one *modifier*. Modifiers allow checking for special matches such as moving objects or the absence of an object. Each section represents an adjacent cell, either vertical or horizontal. Rules can be marked as "late" which means they will run in a second phase after the player movement has been processed. More information on the rules and the modifiers can be found in PuzzleScript's user manual[6]. Figure 3.5 illustrates a few examples of Rules.

3.2.8 Win Conditions

WinCoditions offer a simple way of creating victory conditions that determine when a player has completed (or won) a level. If no conditions are defined, the player will be unable to win a level. If multiple conditions are defined then all of them must be true simultaneously for the player to win. A condition consists of a victory keyword, an object reference or a property, and an optional 'on' condition. An 'on' condition consists of the word 'on' and an object reference. Conditions check whether an object exists on the level (or does not exist). The 'on' condition further requires that the object referenced in the condition be on the same XY-coordinates as the object referenced in the 'on' condition.

Valid keywords for a condition are `Some`, `No` and `All`. A condition with the `Some` keyword becomes true if at least one of the objects referenced is present on the level. If the 'on' condition is defined, it becomes true if at least one of either object is stacked on the other. A condition with the `No` keyword becomes true if zero objects are present on the level. If the 'on' condition is defined, then it becomes true if zero of either object are stacked. A condition with the `All` keyword cannot be used without the 'on' condition being defined. With the 'on' condition defined it becomes true if all objects referenced in the 'on' condition are stacked with an object from the condition. A few examples of these conditions can be seen in Figure 3.6

We note that the syntax of the "Object1 on Object2" syntax can be confusing, and perhaps even misleading. For example, in the case of `All Crate on Target` what this means is that there cannot be a target that is not on the same XY coordinate[7] as a Crate. However, from a natural language point

[6]`https://www.puzzlescript.net/Documentation/rules.html`

[7]We define "stacked" as being on the same XY coordinate as PuzzleScript does not require for a specific item to be on top, as such, an item can still be on another item even if that first item is actually under.

```
        Some target on crate
        No Player
        All oranges on plates
```

Figure 3.6: Three win conditions

```
        message Level 1:  Beginnings

        ####..
        #.O#..
        #..###
        #@P..#
        #..*.#
        #..###
        ####..
```

Figure 3.7: A message and a level

of view, one would assume the contrary, that each Crate must be on the same XY as a Target. This distinction is important when there is more of Object1 than of Object2. For example, if there are four Target and three Crate then the level is impossible to win.

3.2.9 Levels

The level section consists of a combination of any number of *Levels* and *Messages*. A Level is a rectangular 2-D tilemap of variable width and height that consists of symbols representing one or more objects. Only one character long reference can be used, whether they are legend or object names. Aggregation can also be used in levels but not properties.

A Message is a single line consisting of the word 'message' followed by a string (multiple words). When the Level before the message is completed, the engine will display the message before moving on to the next level. Any number of messages can be defined in between two levels, before the first level or after the last level. An example of a Message and Level can be seen in Figure 3.7.

3.3 Implementation of PuzzleScript

Here we describe the design decision of the 'official' implementation of PuzzleScript written by the original developer, Stephen Lavelle, in JavaScript.

The JavaScript implementation of the PuzzleScript engine consists of three separate phases. First, the source code is *parsed* and checked for validity. Second, the *compiler* transforms the parsed code into JavaScript data structures. This also involved transforming rules into JavaScript functions using meta-programming. Further verification is conducted during compilation. Finally, the *engine* runs the game and awaits player input.

The repository containing the source code for PuzzleScript consists of 27 files and adds up to over 15,000 lines of code calculated using *cloc*[8]. Most of the files are used to store code that provides features not directly tied to PuzzleScript's design such as Saving/Exporting games. Those files also contain code for features that are small parts of the design but require large implementations such as sounds. Three files are responsible for the phases we identified above and total 5820 lines of code. The files are:

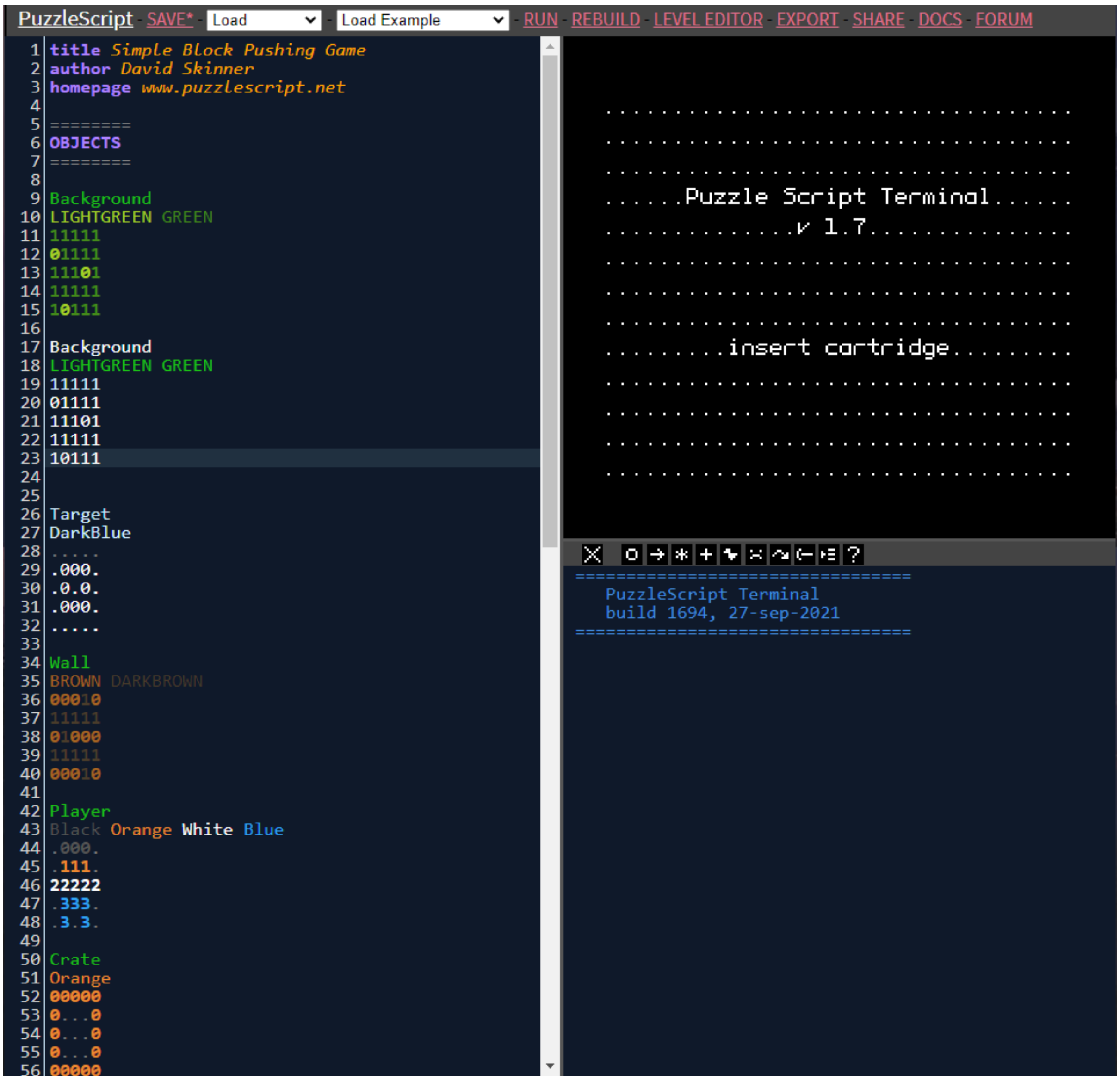
- parser.js: 1065 lines, responsible for the Parsing phase.
- compiler.js: 2350 lines, responsible for the Compilation phase.
- engine.js: 2405 lines, responsible for the Engine phase.

The in-browser IDE, a vital part of the implementation, is implemented using CodeMirror. CodeMirror is a versatile library specifically made for editing code in-browser. The IDE provides comprehensive syntax coloring that adapts to its context. Pixels representing a color will appear as that color in the respective sprites that use them. Syntax coloring will fail if the syntax is incorrect.

Once the user presses "Run" the game compiles, errors appear in the bottom right and the title screen for the game appears in the top right. The top bar of the IDE provides additional options for sharing and exporting the game into a standalone application alongside helpful links for seeking support.

The implementation displays error messages as they are detected and aborts the compilation if a certain threshold is reached. The threshold is necessary because of the possibility of an error cascade. Without the threshold, a high number of ghost errors might be displayed. This would make it harder to fix the true issues. With the threshold, the developer is encouraged to address errors one by one, allowing the static checker to clear up any ghost errors. User inputs are context-sensitive. If the user presses an

[8]http://cloc.sourceforge.net/



```
title Simple Block Pushing Game
author David Skinner
homepage www.puzzlescript.net

========
OBJECTS
========

Background
LIGHTGREEN GREEN
11111
01111
11101
11111
10111

Background
LIGHTGREEN GREEN
11111
01111
11101
11111
10111


Target
DarkBlue
.....
.000.
.0.0.
.000.
.....

Wall
BROWN DARKBROWN
00010
11111
01000
11111
00010

Player
Black Orange White Blue
.000.
.111.
22222
.333.
.3.3.

Crate
Orange
00000
0...0
0...0
0...0
00000
```

```
..................................
..................................
..................................
......Puzzle Script Terminal......
..............v 1.7...............
..................................
..................................
..................................
.........insert cartridge.........
..................................
..................................
..................................
..................................
```

```
==================================
   PuzzleScript Terminal
   build 1694, 27-sep-2021
==================================
```

Figure 3.8: PuzzleScript's browser IDE

arrow key while focused on the editor, it will move their cursor text, if they do the same with a focus on the game, it will attempt to move their character. The IDE does not have a contextual right-click menu.

3.4 Lessons Learned

Here we analyze and discuss design decisions that complicate Puzzlescript's analysis. In particular, we discuss how language features lead to possibly complex situations when writing Puzzlescript due to 'dark corners' in the language semantics.

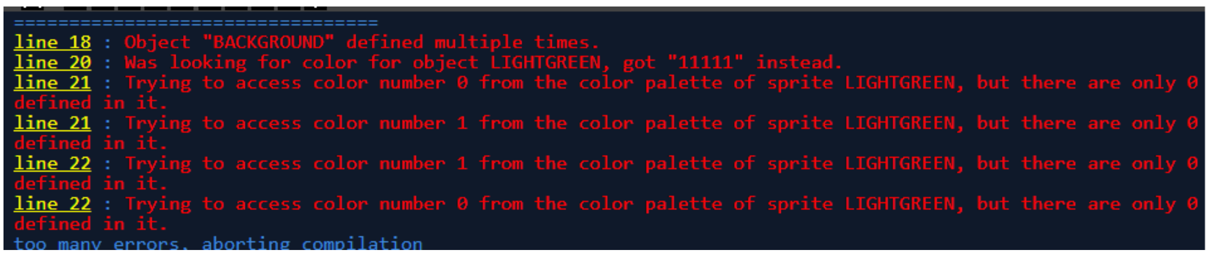
3.4.1 Compilation

PuzzleScript is an incredibly flexible language, both in design and through implementation. However, this flexibility includes corner cases in language semantics that make it hard to formalize the semantics.

For instance, PuzzleScript has section-specific reserved keywords. For instance, specifying the legend for an object as] is allowed. However, the trade-off is that the legend cannot be used to reference that object in the Rules section since] is a reserved keyword in the context of that section. This is made possible by the technical implementation of PuzzleScript's parse which is line-based and handcrafted. The flexibility of PuzzleScript's grammar places it in the category of context-sensitive grammars. Section-

```
Crate,Player, Target
Crate Player Target
Crate, Player Target
```

Figure 3.9: Three examples of the same layer



```
================================
line 18 : Object "BACKGROUND" defined multiple times.
line 20 : Was looking for color for object LIGHTGREEN, got "11111" instead.
line 21 : Trying to access color number 0 from the color palette of sprite LIGHTGREEN, but there are only 0
defined in it.
line 21 : Trying to access color number 1 from the color palette of sprite LIGHTGREEN, but there are only 0
defined in it.
line 22 : Trying to access color number 1 from the color palette of sprite LIGHTGREEN, but there are only 0
defined in it.
line 22 : Trying to access color number 0 from the color palette of sprite LIGHTGREEN, but there are only 0
defined in it.
too many errors, aborting compilation
```

Figure 3.10: A single error causes a cascade of ghost errors

specific keywords are not a common design pattern. PuzzleScript's grammar leveraging that pattern poses a challenge to formalizing it.

The JavaScript implementation has tightly coupled compiler phases. This means that extending the codebase requires a full understanding of the entire process. There are no ”general” functions that can be simply provided as a set of instructions for extensions. PuzzleScript does not provide an interface with helper functions but rather defines functions based on the current needs of the implementation.

3.4.2 Intentional Ambiguities

The design of PuzzleScript includes intentional ambiguities. Components can be defined in multiple ways with slight differences to their syntax. These multiple ways do not affect the behavior of the component but need to be handled by the parser. These variants of the syntax provide no real benefit and may even confuse the game designers. For instance, objects in collision layers are normally separated by a comma, making the formalization of the grammar appear simple. However, upon further inspection, we discover that a list of space-separated objects is also a valid line. This means that there are three possible separators for a collision layer that can be used interchangeable: 1) a comma with no whitespace; 2) a comma and a white space (in any order) and 3) just a whitespace.

3.4.3 Rule Semantics

Rules, the heart of PuzzleScript, are complex to read and hard to maintain. The instructions of the rules are transformed into code which the compiler interprets as a JavaScript function. PuzzleScript's engine makes use of meta-programming as it generates the JavaScript code that represents the rules during runtime as a function. This means that a great part of the engine does not exist until runtime. Because JavaScript is dynamically typed, it is hard to understand what is being passed to these functions.

In the JavaScript implementation, parsing is one of the two phases and is done simultaneously with error checking. This can very easily cause errors to cascade since any error also throws the parser off as can be seen in Figure 3.10. The graphics and engine are similarly coupled, graphics are generated as rules are applied, and the objects have methods for generating graphics directly attached to them. Additionally, the error messages are duplicated throughout the process. This means that if a developer wants to change the wording of an error message, they need to go through the core files and modify every instance of the message.

3.4.4 Interactive Development Environment

The IDE itself is simple, offering syntax coloring, and printing errors. Its real power comes from the engine itself, which allows running games with great performance directly in the browser. When an error occurs, the syntax coloring breaks on the line where the error occurred. This allows for basic debugging but does not provide any information on what the error is. The debug console provides the necessary information to solve most errors. However, it usually only provides enough information to solve the first error encountered. Both of these aspects are functional and serve their purposes but have room for improvement.

3.5 Conclusion

We conclude that PuzzleScript is a complex language that appears simple to the users. The implementation focuses on performance and portability at the cost of extensibility and maintainability. This specific focus makes it hard to use to answer our research questions, as such, we design our own implementation.

Chapter 4

ScriptButler

Here we present ScriptButler, a prototype tool for analyzing PuzzleScript games. The tool aims to give game designers a better understanding of how code changes affect gameplay. As previously mentioned, we created this prototype because the official PuzzleScript implementation is unsuitable for our research goals. Therefore, to resolve this issue we create our own design and implementation of PuzzleScript. We base our implementation on the design document we reverse engineered in the previous chapter. In this chapter, we discuss the design decisions we take. Our aim with this implementation is to create a general-purpose tool capable of parsing, validating, and running games. Additionally, for the purpose of this project, we also design a dynamic analysis module to provide support for analyzing gameplay. The dynamic analysis module also demonstrates the extensibility of our prototype.

Figure 4.1 shows an overview of our tool's architecture. The game designer interact with the tool through the IDE and the Web GUI. These two components provide user-friendly endpoints to our tool and are responsible for sending inputs from the game designer to the tool. ScriptButler is split up into multiple phases, this is done to reduce issues tied to coupling. The parser is our grammar, it reads PuzzleScript code and converts it into Rascal data structures. The post-process phase cleans up those data structures and annotates them to provide an outline and syntax coloring. The static checker verifies the validity of the code and provides error messages that the IDE can display to the game designer. The compiler further transforms the data structures, removing unnecessary content and making them more efficient to run in our engine. The engine run the game and displays the result in the Web GUI. The user can send inputs to the engine using that GUI and get debugging information. Finally, the dynamic analyzer runs tests on the compiled version of the game and provides feedback on the gameplay as messages that the IDE can display. We explain all the sections in greater detail in the following sections.

4.1 Redesigning PuzzleScript with Rascal

Here we present the design of the PuzzleScript implementation we created using the design document we obtained by reverse-engineering the JavaScript implementation. Design decisions that arise from that understanding and that differ from the design decisions of the original implementation are marked with "$^{[D]}$". When designing the implementation, we use a test-driven approach. We write the tests and their expected results based on the reverse engineering of the original implementation. We then implement PuzzleScript's design and frequently test our implementation against the tests. One of the goals for our implementation is to be compatible with existing games and the tests ensure we keep in line with that goal.

Our second goal is to create a design that is more maintainable than the original JavaScript one. Rascal satisfies part of that goal as a statically typed language. This makes it easier for programmers to understand the data flow. We also leverage good engineering practices such as self-documentation through the use of relevant variables and function names. Our third and final goal is creating an extensible design. To achieve this, we design our implementation with loose coupling between modules. Instead of creating a monolithic structure, we create a collection of plugins that are handled by a general function in charge of passing the data from one plugin to the other. Our approach makes it easy to add, remove or modify plugins. In the rest of this chapter, these plugins are called phases as they mirror, to an extent, the phases of compilation.

We split up the design of our implementation into separate phases$^{[D]}$. Each phase has progressively stricter requirements on what counts as valid code. The phases are the following:

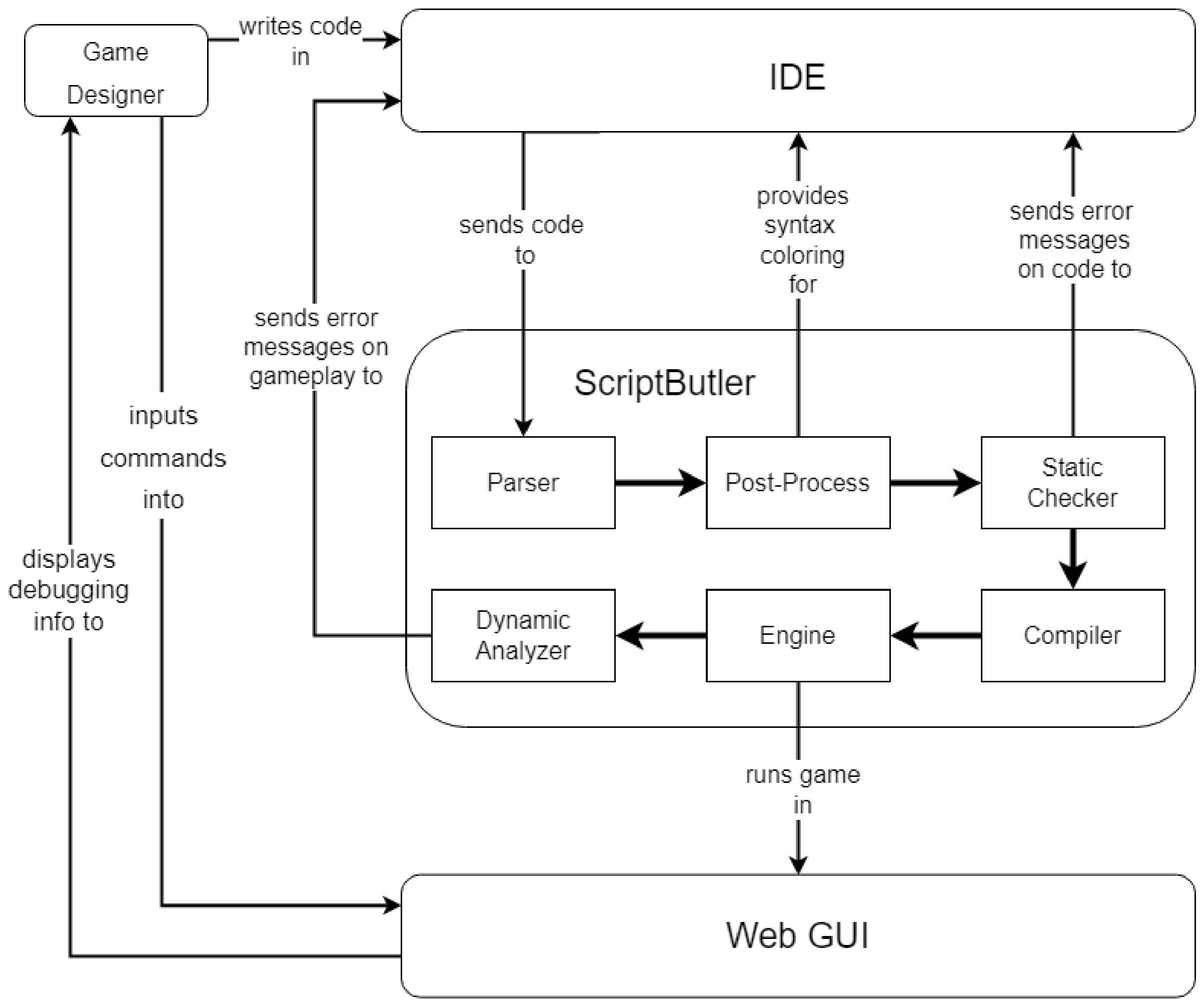


Figure 4.1: The data flow through ScriptButler and its extensions

- Parsing: Using a flexible formalized grammar, a PuzzleScript file is read and transformed into a parsed tree
- Post-process: The parsed tree is cleaned up and enriched with meta-data
- Static Check: Components of the file are checked for validity according to the specifications extracted from the JavaScript implementation.
- Compilation: The game is compiled into a form the engine can run, converting the AST to more efficient Rascal data structures
- Run: The game is running, allowing for the player to interact with it

Splitting the compilation process into separate phases enables us to parse invalid code without the process aborting at the first error. The behavior assists designers in identifying multiple issues simultaneously. We also separate the backend (parsing, checking, compiling) from the frontend (editor, interface)[D]. The backend is designed to accept source code, generate the data structures, and alter those data structures based on user input. The front end acts as a wrapper around the backend, communicating through specific endpoints. The approach allows us to decouple the user interface and engine, making it easier to alter either without impacting the other. Developers can even create their own implementation of the frontend or backend with seamless integration.

The design document of PuzzleScript is subject to errors of misinterpretation because it is based on reverse engineering. We mitigate these interpretation issues by verifying the design of our implementation against the behaviors of the official implementation. We go further into our verification process in Section 4.3.

```
// layout is lists of whitespace characters
layout MyLayout = [\t\n\ \r\f]*;

// identifiers are characters of lowercase alphabet letters,
// not immediately preceded or followed by those (longest match)
// and not any of the reserved keywords
lexical Identifier = [a-z] !<< [a-z]+ !>> [a-z] \ MyKeywords;

// this defines the reserved keywords used in the definition of Identifier
keyword MyKeywords = "if" | "then" | "else" | "fi";

// here is a recursive definition of expressions
// using priority and associativity groups.
syntax Expression
  = id: Identifier id
  | null: "null"
  | left multi: Expression l "*" Expression r
  > left ( add: Expression l "+" Expression r
         | sub: Expression l "-" Expression r
         )
  | bracket "(" Expression ")"
  ;
```

Figure 4.2: Syntax example

4.1.1 Parsing

We address the need for a reusable and maintainable parser by creating a formal syntax definition in the form of a grammar. In our design, we make several important changes to this phase. However, we want our redesign to remain compatible with the PuzzleScript design. As such, it is important that this phase's user-facing endpoints do not change. The main design change in this phase is that we switch from using a hand-crafted parser to using Rascal's Syntax Definition feature[1][D]. This decision allows us to create a declarative representation of a PuzzleScript file that consists of lexicals and symbols. This representation is human-readable and easier to extend than a line-based parser.

A Rascal syntax definition consists of a layout, keywords, lexicals, and symbols. The layout is a pattern that matches the white space and comments, it can be seen at the top of Figure 4.3. Symbols are composed of lexicals, literals, and other symbols. We use lexicals to parse individual tokens and then group them into symbols that represent PuzzleScript components such as Objects, Rules, Sprites and many more. Figure 4.2 shows a simple example using practically all Syntax Definition features.

The main weakness of using Syntax Definition is that we lose control over how the parser handles invalid code. If the parser runs into invalid code it raises an error intended for the tool designer. This error is hard to read for the tool user and does not provide the necessary feedback for a game designer to fix their code. To address this issue, we create a *'flexibile'* grammar. This means that our grammar should parse and accept invalid code, to an extent. We aim to design our grammar so that only severe structural errors can cause it to raise an error. To this end, we define keywords to make our definition more human-readable. However, we do not use them to exclude tokens to avoid a parser error. Our grammar is mostly made up of flexible lexicals grouped up under symbols. The main goal of our grammar is to assign "labels" to tokens so that our static checker knows what rules they should be checked against. Figure 4.3 shows the lexicals, and Figure 4.4 shows a sample of our grammar that parses Objects. The latest version of the full grammar can be accessed on the project repository[2].

The Syntax Definition also offers an opportunity to annotate the file with interesting metadata. This metadata can then be used by the syntax checker or IDE plugins to enrich and support the developer experience. For example, the `@Foldable` tag, allows the developer to collapse any piece of code that matches the symbol in the IDE to temporarily hide it. The design of our grammar does still present weaknesses. PuzzleScript's design presents several ambiguities which a formal grammar cannot handle. We mitigate the effect of these ambiguities by defining certain objects more rigidly. This approach directly counteracts our goal of keeping the grammar flexible so we carefully balance the two.

The result of this section is a grammar that we can apply directly to a PuzzleScript file to generate a parse tree representing that game. The parse tree needs further manipulation to be usable but using a Syntax Definition fulfills our goal of creating a more readable version of PuzzleScript's grammar.

[1]`https://docs.rascal-mpl.org/stable/RascalConcepts/#RascalConcepts-SyntaxDefinitionAndParsing`

[2]`https://github.com/ClementJ18/ScriptButler/blob/main/src/PuzzleScript/Syntax.rsc`

```
lexical LAYOUT
  = [\t\r\ ]
  | ˆ Comment Newlines
  > Comment
  ;
layout LAYOUTLIST = LAYOUT∗ !>> [\t\r\ )];

lexical SectionDelimiter = [=]+ Newlines;
lexical Newlines = Newline+ !>> [\n];
lexical Comment = @Category="Comment" "(" (![()]|Comment)+ ")";
lexical Newline = [\n];
lexical ID = @Category="ID" [a−z0−9.A−Z#_+]+ !>> [a−z0−9.A−Z#_+] \ Keywords;
lexical Pixel = [TRUNCATED];
lexical LegendKey = Pixel+ !>> [TRUNCATED] \ Keywords;
lexical Spriteline = [0−9.]+ !>> [0−9.] \ Keywords;
lexical Levelline = Pixel+ !>> Pixel \ Keywords;
lexical String = ![\n]+ >> [\n];
lexical SoundIndex = [0−9]|'10' !>> [0−9]|'10';
lexical KeywordID = @Category="Key"[a−z0−9.A−Z_]+ !>> [a−z0−9.A−Z_] \ 'message';
lexical IDOrDirectional = @Category="ID" [\>\<ˆva−z0−9.A−Z#_+]+ !>> [\>\<ˆva−z0−9.A−Z#_+] \ Keywords;
```

Figure 4.3: PuzzleScript layout and lexicals

```
syntax Objects = objects: SectionDelimiter? 'OBJECTS' Newlines SectionDelimiter? ObjectData+;
syntax Color = @category="Color" ID;
syntax Colors = Color+;
syntax ObjectName = @category="ObjectName" ID;

syntax ObjectData
  = @Foldable object_data: ObjectName LegendKey? Newline Colors Newline Sprite?
  | object_empty: Newlines
  ;

syntax SpritePixel = @category="SpritePixel" SpriteP;

syntax Sprite
    = @Foldable sprite:
        SpritePixel+ Newline
        SpritePixel+ Newline
        SpritePixel+ Newline
        SpritePixel+ Newline
        SpritePixel+ Newline
    ;
```

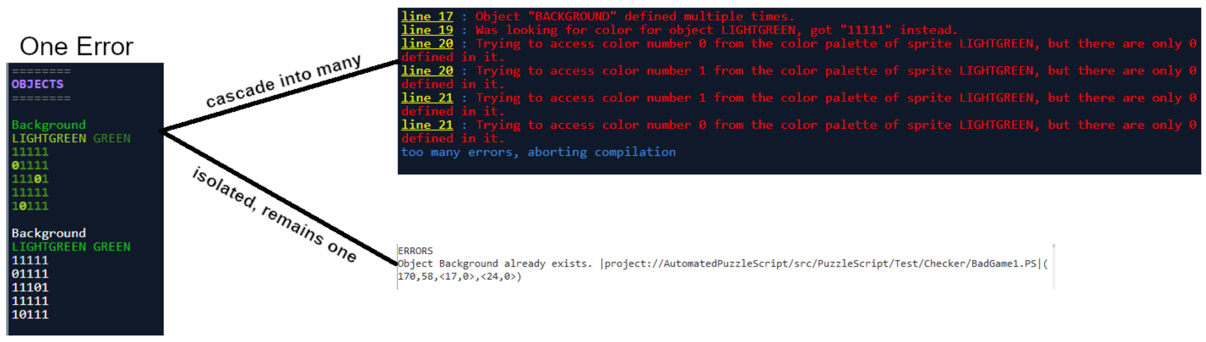
Figure 4.4: Symbols for parsing Objects

4.1.2 Post Processing

The post-processing phase aims to make it easier to manipulate game code programmatically and pad the weaknesses of the grammar. During this phase, we annotate the parsed tree with metadata. This metadata can be used by other phases of our tool to provide feedback or create features to support game designers. For instance, our IDE plugin uses object annotations to provide syntax coloring for the game designer. The reason that we annotate the objects rather than providing direct syntax coloring is part of our design decision to decouple frontend and backend. The benefit is that as long as the annotations are not modified, the syntax coloring will remain functional even if the grammar changes.

As we have previously mentioned, the grammar creates a parse tree that is hard for a compiler or game engine to manipulate. This phase help us address those flaws. We traverse the parse tree and map PuzzleScript objects to more efficient data structures. For instance, in our grammar, Sprites are defined as five individual lines. These lines are considered as individual attributes and cannot be accessed by index. In this phase, we access each of the individual lines and transform them into a list that is easier to traverse. In addition, we attach the original object to the new data structure. This approach guarantees that IDEs always have a copy of the original code to display to the game designer. The trade-off of this approach is that our tool takes up more memory space. Finally, this phase also insures that all sections of the game exist, even as empty sections. This is useful in certain instances when the game designer may omit certain sections. In these cases, our AST is still guaranteed to have an attribute for that section that can be accessed for the tool but will return as an empty section.

The result of this section is a data structure representing the entire game and all its components. This data structure is easier to manipulate than the original AST. This phase also provides simple feedback to game designers in the form features such as syntax coloring. We discuss the many uses of the annotations in Section 4.2.2. We pass the data structures to the static check which is described in the next section.



```
========
OBJECTS
========

Background
LIGHTGREEN GREEN
11111
01111
11101
11111
10111

Background
LIGHTGREEN GREEN
11111
01111
11101
11111
10111
```

```
line 17 : Object "BACKGROUND" defined multiple times.
line 19 : Was looking for color for object LIGHTGREEN, got "11111" instead.
line 20 : Trying to access color number 0 from the color palette of sprite LIGHTGREEN, but there are only 0
defined in it.
line 20 : Trying to access color number 1 from the color palette of sprite LIGHTGREEN, but there are only 0
defined in it.
line 21 : Trying to access color number 1 from the color palette of sprite LIGHTGREEN, but there are only 0
defined in it.
line 21 : Trying to access color number 0 from the color palette of sprite LIGHTGREEN, but there are only 0
defined in it.
too many errors, aborting compilation
```

```
ERRORS
Object Background already exists. |project://AutomatedPuzzleScript/src/PuzzleScript/Test/Checker/BadGame1.PS|(
170,58,<17,0>,<24,0>)
```

Figure 4.5: Isolation of errors

4.1.3 Static Checking

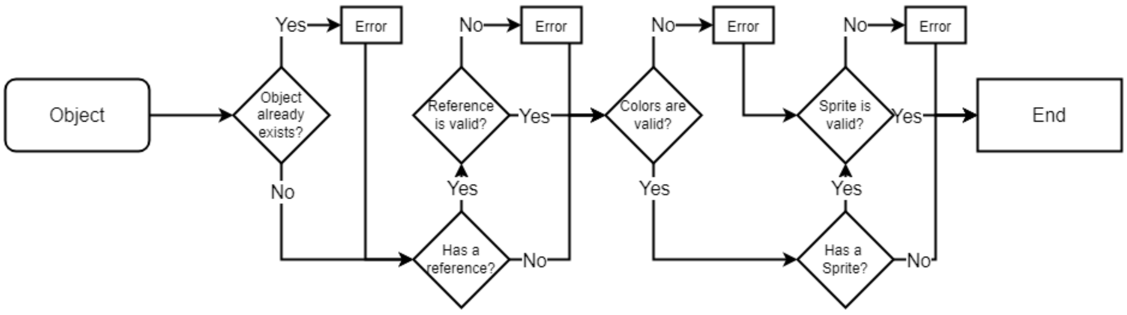
The static checker is in charge of verifying the validity of the code ensuring that the engine can run it. The phase exists so that the code can be checked thoroughly all at once rather than having the game designer debug errors during runtime. In the JavaScript implementation, this is done during the parse and compile phase. In the design of our Rascal implementation, this is done after the post-processing phase and before the compilation phase[D]. Static code analysis is used to generate human-readable error messages before the compiler transforms the code into more complex Rascal data structures.

The static checker verifies the integrity of each component in a specific order. Components that define variables that can be referenced later are checked as soon as possible so that the references can be checked before they are used. Each component is checked for integrity based on the design extracted from the JavaScript implementation. Because the static checker isolates components errors, code cannot cascade and taint the rest of the file[D], this is shown in Figure 4.5. An error can, at most, make it impossible to check the current component. In those cases, the checker will return the current error message and move on to the next component. The only case in which errors 'spill' is when an invalid component is referenced. In this case, the checker will also generate an error stating that the reference to the invalid component does not exist. This amount of 'spill' is acceptable as it only happens in cases where the component has severe syntax errors.

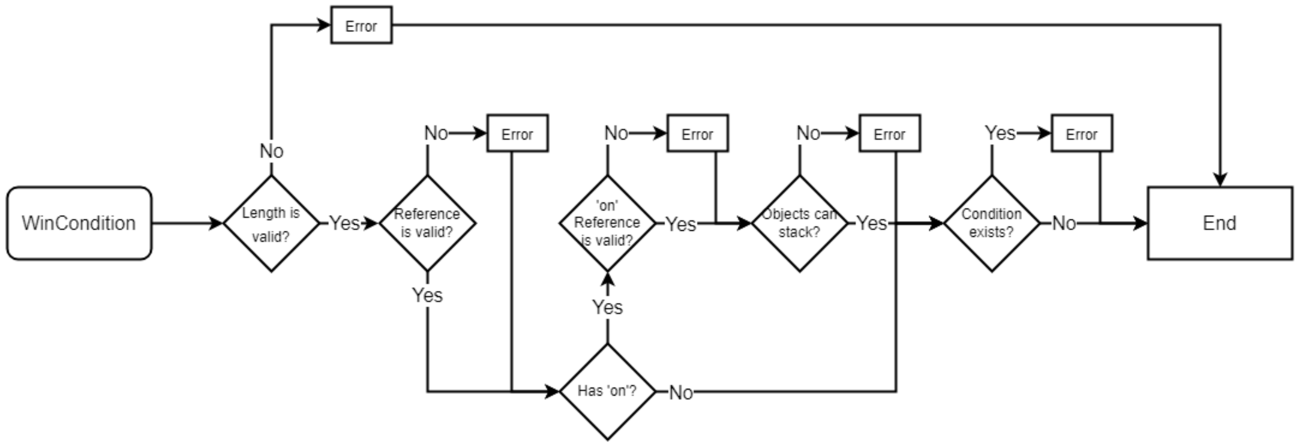
The JavaScript implementation has 117 unique error/warning messages. This number does not account for messages that appear multiple times. Although our design currently checks for only 72 messages, the discrepancy has two reasons. As previously mentioned, using Rascal's Syntax Definition entails a certain loss of control over the feedback provided in exchange for more readability. The first reason is therefore that a part of these messages are now covered by the general parser failure error intended for the tool designer. The second reason is that, in our design, another part of the messages we extracted from the original implementation have been merged[D]. For instance, the original message has two separate messages to report on an error within the rule depending on whether that error was on the left-hand side or the right-hand side. In our design, these errors are merged into one with a keyword to differentiate the side that raised the error.

Figure 4.6a shows an example of how the checker processes a PuzzleScript component, in this case an Object. Diamonds represent tests that the checker submits the code to and rectangles represent an outcome. This is a simplification of the process that omits specifics on what is considered 'valid'. For instance, a valid color is either a HTML color code or a selection from the color palette selected by the game designer in the prelude. Objects that do not pass the requirement generate an error message that is returned by the checker, to be displayed later. In some cases, an early error causes the remaining checks to be skipped as can be seen in Figure 4.6b which illustrates the process for a Win Condition. Figure **??** shows how the static checker validates the remaining components of PuzzleScript. Once detected an error is not immediately displayed but rather stored[D], this serves the dual benefit of making it easy for the IDE to customize how they display the messages and allowing us to centralize our human-readable conversions of these messages. As a result, our redesign is easier to extend in the cases where a tool designers wants to modify the messages. For instance, if a tool designer wants to translate PuzzleScript, all they have to do is go through a single file to have access to the messages.

In our design, error messages have several different types and subcategories. The main types are *errors* and *warnings*, support for lower categories exist but are currently unused. Generated messages also fall under one of a few sub-categories:



(a) Flow diagram when validating an Object



(b) Flow diagram when validating a WinCondition

Figure 4.6: Flow diagrams illustrating the validation process done by the checker

- Invalid: The component's syntax is not respected making it unusable
- Undefined: The reference/object with that name is never defined but is used
- Existing: The reference/object with that name already exists, but the code is trying to define it again. Sometimes this is generated as an error and sometimes as a warning depending on whether the engine can handle it.
- Unused: A warning, the code defines a reference/object/sound but never uses it[D].
- Misc: Very specific errors that do not occur in enough numbers to justify a category

We categorize each error with a degree of importance based on whether or not it has a negative impact on the game and its ability to run. Error-level messages can cause issues or unintended side effects in the code that may make the game impossible to resolve, warning-level messages indicate dead or unoptimized code, and information-level messages inform the game designer on gameplay quality and possible best practices. A full list of errors and warnings raised by our tool can be seen in Appendix B.

As we previously mentioned, our tool raises errors and warnings that do not exist in the original implementation. Part of these messages are merges and others are brand news[D]. A sample of these new messages is displayed in Table 4.1. Many of these messages reference existing objects that are never used. Unused objects present two issues. The first issue is that unused objects represent dead code that inflate the file size but add no value. Dead code adds unnecessary complexity to the codebase and makes it more difficult to understand. The second issue is that unused objects may also give a false impression of the game. This false impression of the game makes it hard for other game designers to understand the design of the game. Our tool also raises a warning when redundant components are implemented into the game. For instance, duplicate win conditions can happen when game designers are not fully aware of what object a legend references. This can cause them to create duplicate instances through the use of different legends referencing the same item. Finally, ScriptButler performs additional checks on Win Conditions, an area left relatively untouched by the original implementation. We design checks to ensure that game designers do not accidentally write conditions that are mutually exclusive.

The result of this phase is a list of messages reporting on the state of the game's code. The intent

```
Player
black orange white blue
.000.
.111.
2222222222
.333.
.3.3.
```

(a) Sprite not 5x5

```
####
#.O#..
#..###..
#@P..#
#..*
#..###
####..
```

(b) Uneven level rows

```
Crate
orange green
00000
0...0
0...0
0...0
00000
```

(c) Unused color

```
[Eyeball| ... |Player] -> [> Eyeball|Player]
```

(d) Missing ellipsis in right hand side of rule

```
[> Player|Crate] -> [> Player] [> Crate]
```

(e) Unexpected rule part in right hand side of rule

Figure 4.7: Errors and warnings detected by the checker

Table 4.1: New errors in ScriptButler

Name	Type	Message
existing_sound	Warning	A sound event has already been registered for this object with these actions
existing_condition	Warning	A victory condition similar to this one already exists
existing_rule	Warning	A rule similar to this one already exists
impossible_condition_unstackable	Error	A victory condition requires items existing on the same layer to stack
redundant_prelude_value	Warning	This prelude keyword does not require a value
unused_colors	Warning	This object defines more colors than it uses
unused_object	Warning	This object is defined but never used
unused_legend	Warning	This legend is defined but never used

is for those messages to be passed on to the IDE and displayed to the user as shown in 4.7. The game designer gains a complete overview of the game code and an understanding of its current flaws. Once the designer has resolved the errors, the game code is passed to he compiler. The compiler is discussed in the next section.

4.1.4 Compiler

Game languages require compilers to optimize their performance. Compilers transform the code into lower-level machine code. For PuzzleScript, this means converting the DSL into Rascal data structure. ScriptButler already does part of this task during the parsing, the compilation process completes it. The compiler transforms the existing data structures further, this increases performance but reduces readability. The new data structures do not resemble the original code at all. It may even be impossible to generate code back from the compiler because the structure is lost. For instance, the compiler resolves all references, replacing them with the list of object referenced. As a result, the original reference is lost. Our compiler is separated into three phases, representing the three sections that need to be compiled. Sounds are not compiled because they are not implemented. Objects are not compiled because our tool only requires their name. As such, we only compile three sections: Win Conditions, Levels, and Rules. All three are explained in detail below.

Levels

In PuzzleScript, game designers create 2D representations of their levels with the use of legend and aggregates. However, levels are actually 3-dimension when factoring in the collision layer. As a result, we compile the levels into 3D arrays. The design decision we have to take is in which order to arrange the array and their relation to the coordinates. Each object on a level has an XYZ coordinate. X represents the row, Y the column, and Z the layer they are present on. Each of these triples is unique. We have two choices, we can either store the levels as a XYZ array or as a ZXY array. A XYZ array is an array where the first dimension represents the X coordinate, the second the Y coordinate and the third the Z coordinate. An XYZ array has a similar concept but in a different order. The order matters depending on how we match the rules. We decided on an ZXY[D]design, where the first dimension is the layer. This makes it easier to build the level layer by layer. Using the other method would allow us to build levels one row at a time but makes it harder to check using our rule system.

Levels are designed to be able to function standalone. Each level stores all the data required to function. Only the rules and win conditions are required to run a level. However, to render a level, the engine also provides access to the object sprite even if it does not use it currently. During compilation, references and aggregation are resolved and a background layer is generated for rendering purposes. The data structure makes it possible for the engine to store the contents of a level in a simple structure.

Win Conditions

Compiling win conditions is a simple process. We resolve references and store them in data structure that makes it easier to check against the 3D structures representing our levels. Levels are also compiled in a way that allows the engine to check for their contents efficiently. Because we store the contents of a level in a simple structure, we can easily check the prerequisites of certain win conditions. For instance, if a level does not contain any instance of ObjectX then the condition `'No ObjectX on ObjectY'`. This helps performance as it does not require us to go through the level pixel by pixel. The engine can simply access the list of objects contained in the level and make assumptions based on that. However, if there is an instance of ObjectX present, we then have to conduct the more expensive check.

Rules

Rules are the most complex part of PuzzleScript and we aim to simplify them in ScriptButler. This goal means that we aim to make the system simpler to extend and we make the rules simpler to debug from a game designer perspective. To this end, we leverage a feature of PuzzleScript called Abstract Patterns[3][D]. Abstract Patterns can be used to match the code with patterns. We use it to match data structure representing a level. However, Rascal does not have native support for creating these Abstract Patterns. The patterns are intended to be used statically to match specific code patterns. We need to be able to generate Abstract Patterns from the Rules written by game designers. We achieve this by

[3] `https://tutor.rascal-mpl.org/Rascal/Patterns/Abstract/Abstract.html`

```
1        1 # [
2        2 # [ *layer0 ],
3        3 # [ *layer1 ],
4        4 # [ *prefix_lines2,
5        5 # [ *prefix_objects2,
6        6 # Object player0 : moving_object(str name0_0_2 : /player/, int id0_0_2, str direction0_0_2 : relative_right, Coords coords0_0_2 :
             <xcoord0, ycoord0, zcoord0_0_2>),
7        7 # Object crate0 : object(str name1_0_2 : /crate/, int id1_0_2, Coords coords1_0_2 : <xcoord1, ycoord1, zcoord1_0_2>),
8        8 # *suffix_objects2 ],
9        9 # *suffix_lines2 ]
10       10# ]
```

(a) An abstract pattern of the rule in Figure 4.8b

```
[ >  Player | Crate ] -> [  >  Player | > Crate  ]
```

(b) A simple PuzzleScript Rule allowing pushing Crates

Figure 4.8: Compiling rules

generating Abstract Patterns as code to be interpreted by our engine[D]. This design decision incurs a performance loss, as the Rascal code is not compiled but interpreted.

Figure 4.8b shows a simple Rule written in PuzzleScript. This rule allows the player to push a crate. Using our compiler, we transform the rule into a Rascal Abstract pattern that can be seen in 4.8a. The pattern represents a 3D array similarly to the compiled form of a Level. In this instance, the game has three layers, as such the first dimension of the array is three elements long. However, the objects referenced in the original layer only exist on the third layer. As such, the first two layers (lines 2 and 3) match for anything and then pass the match to the replacement, changing nothing. Line 4 and 5 unpack the second and third dimension of the level, ensuring that whatever comes after and before the match is passed to the replacements pattern. Finally, lines 6 and 7 handle the actual matching. Line 7 matches a player moving in a direction and Line 8 matches a crate standing still. A similar pattern is used to replace the crate with a moving crate.

The compiler requires multiple passes to compile a rule into a list of abstract patterns. However, the end result is more readable from the perspective of a tool designer. Our aim with this design decision is to make it easier to extend PuzzleScript with additional features in rules. Validating Abstract Patterns is easier than validating functions like the original implementation generates.

The result of the compiler is a list of data structures for Win Conditions, Rules and Levels which can be run by the engine. The data structures we compile are easier to extend than the original implementation. This decision does not change anything for the game designer besides a performance impact caused by the new design focus on extensibility and maintainability.

4.1.5 Engine

We divide the game loop into additional phases to avoid unnecessary coupling. We make it so that the phases only require the necessary data to be passed[D], as opposed to passing the entire game. Figure 4.9 shows a full flow diagram of the engine, we briefly explain the different phases below:

- Plan Move: Mark player objects as needing to be moved if the player has given a move command
- Apply Rules: Apply the rules as many times as possible
- Do Move: Move all objects marked as such, if there is an obstacle, see if we can move that obstacle
- Apply Late Rule: Apply the late rules as many times as possible.
- Check Victory: Check if we meet all the victory conditions, if not, do the loop again

The engine provides support to the game loop in the form of individual functions[D]. The game loop we design uses those function to run the game. We made this design decision to make it simple for other developers to create their own version of the loop by assembling the functions in a different order. Tool developers can also override the function to inject their changes without modifying the loop. For avoiding infinite loops, users can specify a limit on the number of times a rule can be applied.

4.2 Extending our implementation

We extend the design of our PuzzleScript implementation with several features to support iterative game design. First, we design a dynamic analyzer that provides the game designers with feedback on their gameplay. Second, we implement a plugin that provides syntax highlighting and error reporting within

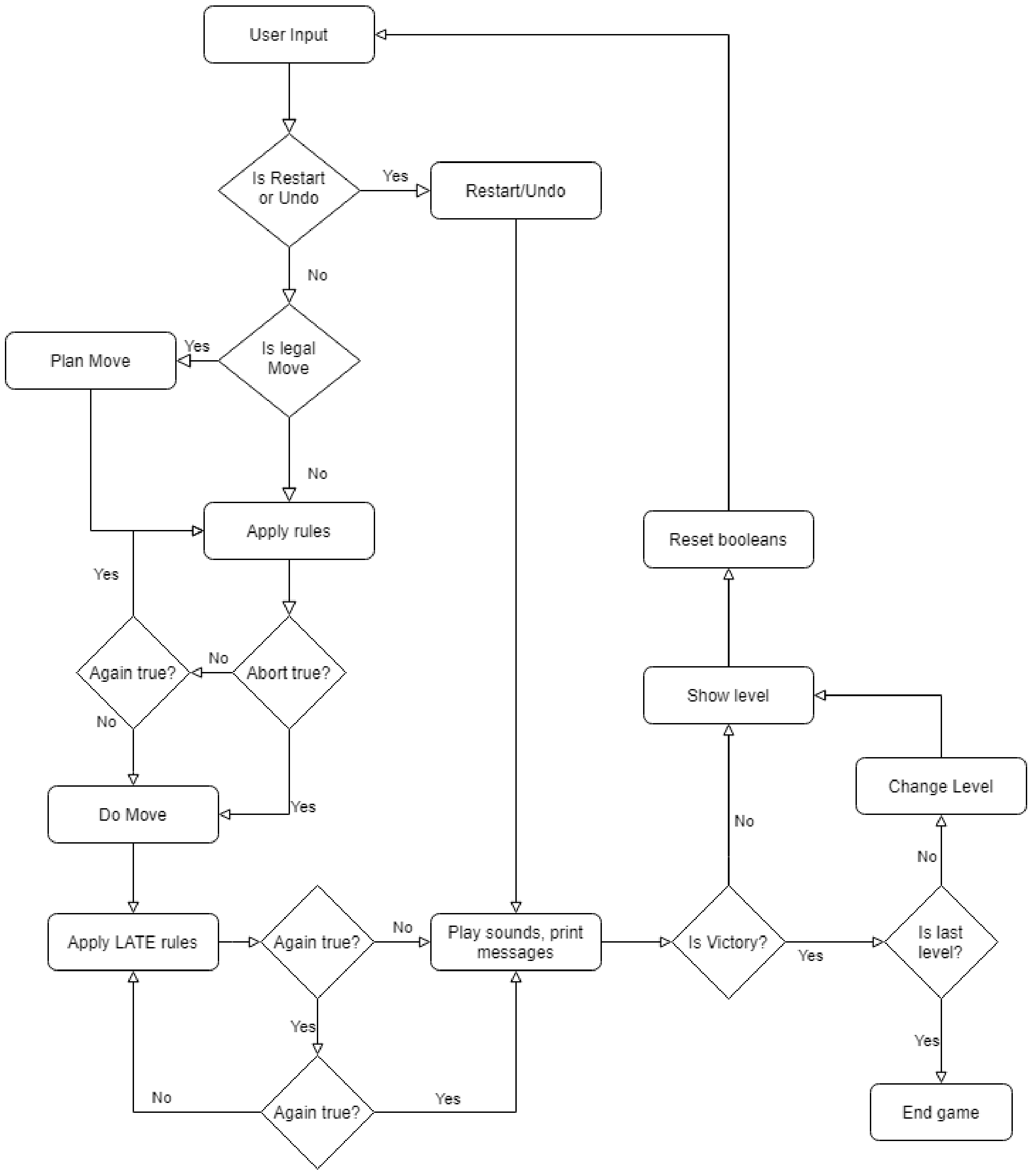


Figure 4.9: Engine flow graph

Eclipse. Finally, we design a user-facing interface using Salix to run games. Below, we detail these functionalities one by one.

4.2.1 Dynamic Analyzer

Our dynamic analyzer phase runs after the engine phase, allowing game designers to gather gameplay-related feedback. The phase runs subsets of the compiled code in the background when the project is built. This process is more expensive than a static check but not required as often. Messages are generated the same way as the static checker and can be manipulated similarly by IDEs.

We perform two types of dynamic analysis, standard dynamic analysis that checks for general gameplay quality and "trivial solutions". The standard dynamic analysis either checks components individually or by seeing how a few different ones interact together. Our "trivial solutions" follow in the line of thinking of our original questions on the complexity of finding fun. Finding a solution to a level is similarly complex, checking if a level can be solved is more of an AI question. Our design makes it possible to implement an AI plugin to solve this problem but that is not the focus of this thesis. Therefore, instead of trying to determine if a game is fun by checking if a level is solvable in an interesting way, we make check whether a game could be unfun by checking if a level is solvable using uninteresting solutions.

Currently, two trivial solutions exist:

- Unidirectional: Is it possible to win the level by only going in one direction
- Unruled: Is it possible to win the level without using any rules.

Unidirectional is done by running a level in the background with the four directional inputs. If after submitting the length of level times two in inputs the level is not won then it is not considered **unidirectional** in that direction. If none of the four inputs come up positive, the level passes. For **unruled**, we use the compiled version of Levels and WinConditions to check whether all objects required are present on the map. This does not, however, check if they are reachable, simply whether they are present or not.

When performing our standard dynamic check, we verify the following:

- Instant Victory: Is the win condition already fulfilled for the level before player interaction
- Impossible Victory: Does the win condition require objects that are not present and not supplied by any rules
- Rule Similarity: A deeper check where we compare the compiled forms of the rules to see if they are similar
- Metrics: We provide various metrics to the game designer that might be useful. This data is provided without analysis to avoid possible misinterpretation.
- Unusable Rule: Similarly to Impossible Victory, we check whether the prerequisites for a rule can be spawned by another rule.

Our dynamic analysis is done similarly to our static checking, it is its own phase, but instead of taking an AST of the game, it takes a compiled engine. The same maintainability and extensibility principles apply as with the rest of the tool. Adding more dynamic checks is a simple matter of writing the logic, adding messages and calling the logic from the main function. The result is also similar, a list of messages to be manipulated as the tool designer desires. The common next step is to display those messages in the IDE.

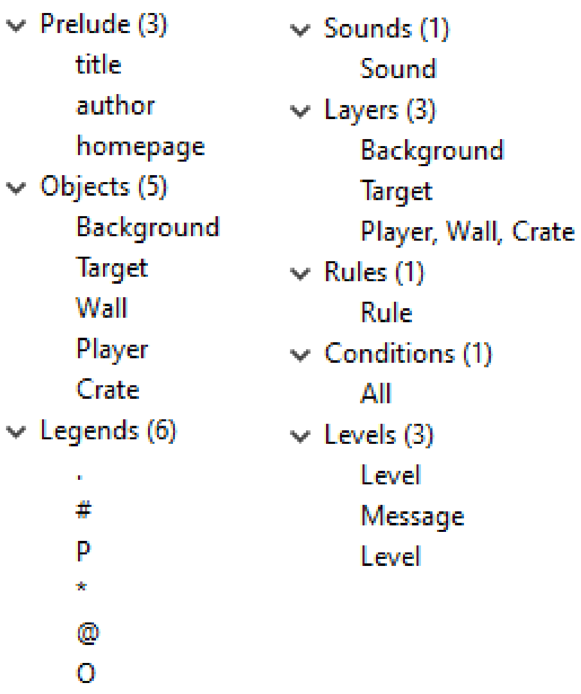
4.2.2 IDE

ScriptButler offers an IDE plugin integrated within Eclipse. This plugin leverages the fact that Rascal's Eclipse integration. Using simple code we add an outline, syntax coloring, a contextual menu, and log messages. We aim to support game designers by providing more features than the IDE from the original implementation. As a reminder, the original implementation provides syntax coloring and partial error reports. Additionally, it also provides features that allow game designers to share and export their games. However, we specifically focus on tools that provide support for game designers in their explanation of the design space.

The first feature that ScriptButler's IDE provides is syntax coloring. This feature already exists within the original implementation. In our design, we tie it to the parse tree annotations[D]. Users can change the coloring without having to change the code of the grammar or mess around with regular patterns. Additionally, this decision is important because the annotations are generated dynamically during the post-processing phase. Syntax coloring improves readability and provides context to the code[43]. Additionally, in the case of PuzzleScript we color the sprites to provide game designers with a preview of the sprite's appearance.

The second feature is an outline view[D]. An outline view is a separate window the game designer can open to gain an overview of their game's contents. This feature allows the game designer to quickly navigate the file based on its contents. For instance, if a designer desires to return to the code for a specific object, they can simply click on that object's name in the outline. The outline improves the efficiency of the designer, especially as the file size grows. The outline we design provides an overview and a navigation shortcut for all major components of PuzzleScript. Tool designers can easily extend this outline to provide an overview of smaller components or to display additional data. Figure4.10 shows this outline view with all branches expanded.

The third feature of the IDE is the display of error messages. In the original implementation, messages



Prelude (3)
- title
- author
- homepage

Objects (5)
- Background
- Target
- Wall
- Player
- Crate

Legends (6)
- .
- #
- P
- *
- @
- O

Sounds (1)
- Sound

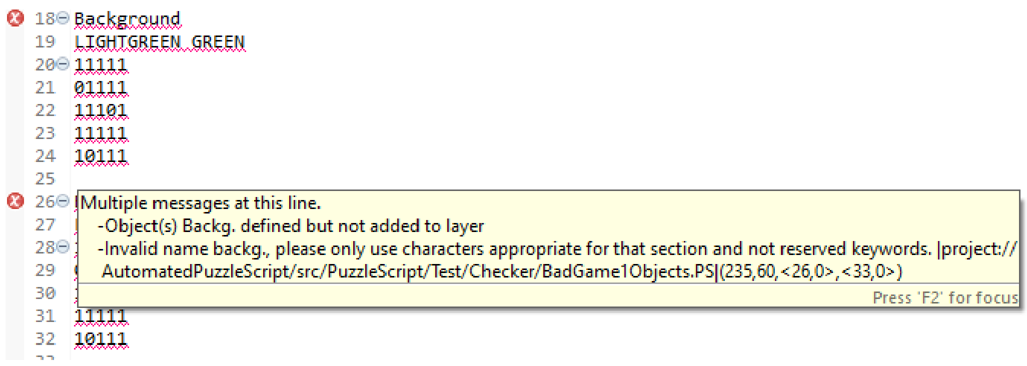
Layers (3)
- Background
- Target
- Player, Wall, Crate

Rules (1)
- Rule

Conditions (1)
- All

Levels (3)
- Level
- Message
- Level

Figure 4.10: IDE Outline



```
Background
LIGHTGREEN GREEN
11111
01111
11101
11111
10111

11111
10111
```

```
Multiple messages at this line.
  -Object(s) Backg. defined but not added to layer
  -Invalid name backg., please only use characters appropriate for that section and not reserved keywords. |project://
   AutomatedPuzzleScript/src/PuzzleScript/Test/Checker/BadGame1Objects.PS|(235,60,<26,0>,<33,0>)
                                                                                              Press 'F2' for focus
```

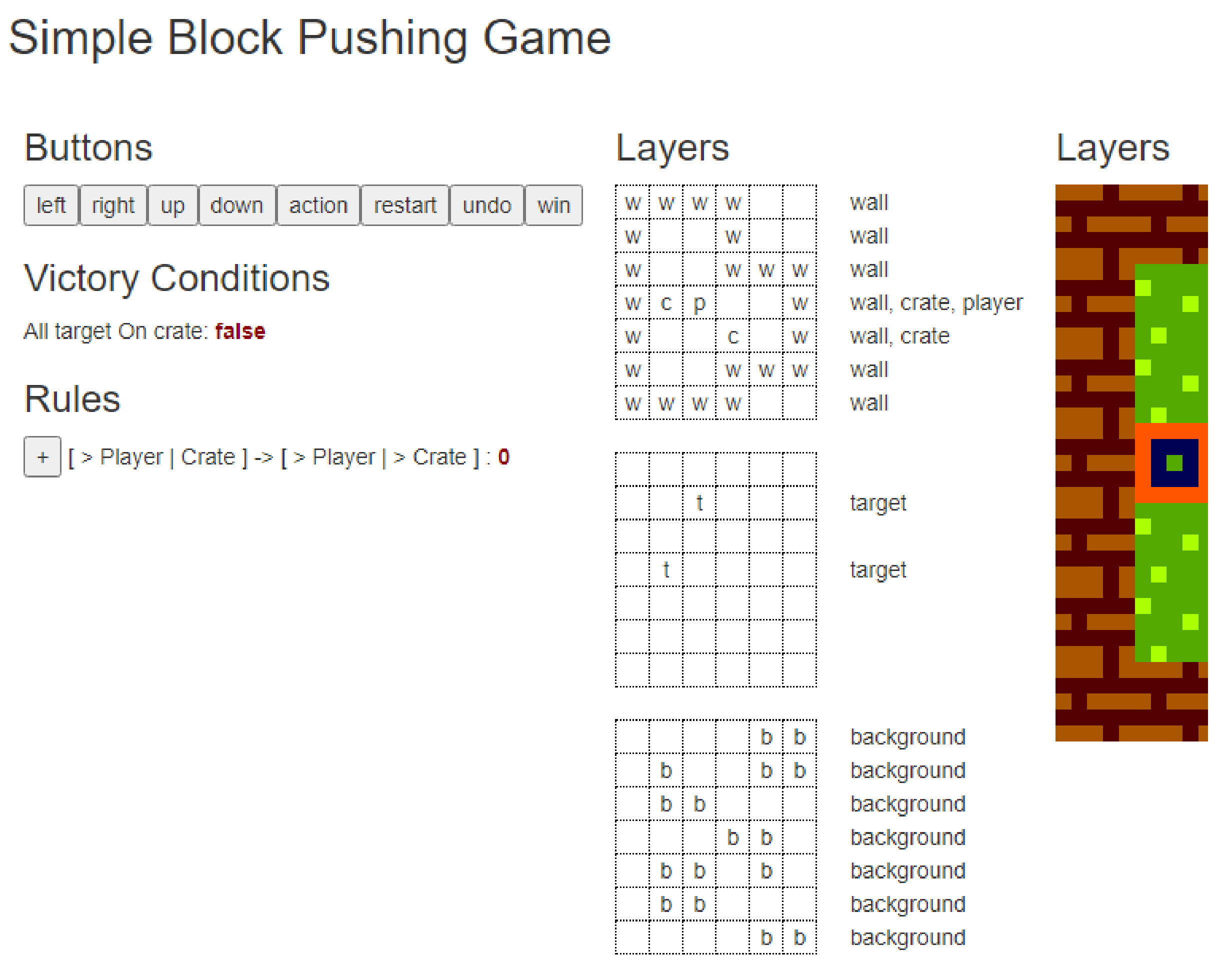
Figure 4.11: IDE Messages

just appear in the console after compilation. However, this kind of implementation does not scale well with high number of messages. As such, we design ScriptButler's IDE so that it display the error messages right next to the code line creating the issue. Figure 4.11 shows this design implemented. The final feature of the IDE is a contextual menu. This contextual menu provides game designers with the ability to start up the Web GUI straight from the editor. However, the feature is very extensible and can be used for a wide variety of actions. For instance, it should be possible to use it to export the game or share it online.

In conclusion, ScriptButler's IDE provides game designers with an editor to interact with the tool itself. This editor supports game designers with a wide variety of features that make programming more efficient.

4.2.3 Running Games

ScriptButler offers game designers the ability to run their game in a browser. In the original implementation, this feature is coupled with the editor. However, we separate it and extend its capabilities to provide playtesting support using a *salix* web app. At its heart, the salix web app provides game designers the ability to run their game and to see it graphically represented through a Web GUI. We use the Web GUI because Rascal does not possess the feature to represent the game graphically or to 'wait' for user input. We leverage this weakness as a strength by also extending the Web GUI with useful



Simple Block Pushing Game

Buttons

Victory Conditions

All target On crate: **false**

Rules

+ [> Player | Crate] -> [> Player | > Crate] : **0**

Layers

w	w	w	w			wall
w			w			wall
w			w	w	w	wall
w	c	p			w	wall, crate, player
w			c		w	wall, crate
w			w	w	w	wall
w	w	w	w			wall

		t				target
	t					target

				b	b	background
	b			b	b	background
	b	b				background
			b	b		background
	b	b		b		background
	b	b				background
				b	b	background

Layers

Figure 4.12: Salix web app

playtesting tools.

The GUI provides a list of victory conditions and whether or not they are true in the current level state. Game designers can use this feature to check their assumptions of what their victory conditions are. They can create a level state where they think certain conditions should be true and then check whether or not they actually are. The GUI also provides a list of rules, how many times they were used in the last turn and what they look like when compiled. The last part is probably more useful to engine developers, which is why it's hidden from sight by default. Game designers can use this feature to check whether the rules actually run when they intend them to, and how many times they match. Finally, the GUI provides game designers with a *'layered'* view of their level. A 'layered' view is a representation of the level where objects are separated based on which collision layer they appear on. This feature provides an easy way for users to see all objects on a particular pixel that may otherwise be hidden by objects they are stacked under.

The Web GUI provides tool to support the game designers when they debug their games. It uses salix which means it is, for a majority, written in Rascal. This design decision makes it easier to edit for those seeking to modify or extend the tool. The downside to our approach is that salix is still a relatively new library and suffers in regard to response times. Figure 4.12 shows an example of Web GUI's appearance.

4.3 Testing

Here we describe an extensive test suite that verifies ScriptButtler's implementation works according to its design. We describe how the suite tests different parts of the tool and what the test results are. We validate and verify our tool using two separate methods. We describe validation Chapter 5 and verification in this chapter. The test suite consists of a dozen sections that test each section of our

PuzzleScript implementation. The tests can be found on the GitHub repository[4].

Each section consists of several PuzzleScript files containing mock data and one Rascal file that conducts the tests. The test files are intended to either verify correctness or error handling. Verifying correctness consists of running valid code through the section we are testing. Verifying error consists of running invalid code and testing to see if a controlled error message is raised. The test files consist of mock data which represent games, sections and atomic components of PuzzleScript. The complexity of the mock data varies based on the test requirements.

Parser The test suite verifying the Parsing phase is the most extensive but also has the simplest mocking data. We test individual elements of a game such as objects, rules parts and levels; complete sections and full games. We test for the correctness of this phase by checking whether or not our grammar successfully parses the mock data. Conducting additional tests to verify the integrity of the parsed data during this phase is unnecessary, the data is verified in future phases. In the cases where the parsed data is invalid, then the Abstract Syntax Tree (AST) itself will raise an error, therefore, we verify the parsed data in the phase where we verify the AST. We reuse the mock data from the parsing test suite quite frequently as mock data for our other test suites.

Static Checker The next suite performs verification on the functionality of the static checker by purposefully triggering errors. The mock data consists of sections of PuzzleScript games plus a few additional cases that do not fit in any particular section. Throughout our test suite, we also have mock data which specifically focus on getting a correct result, and this mock data is used both for verification and to guide us during development.

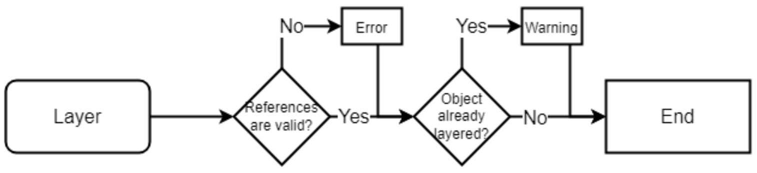
Test-Driven Development Our approach to development is test-driven, so we develop by designing tests first and then writing code that passes those tests. Test-driven development requires the test suite to be designed with intent. We focus on creating tests that provide coverage for both high-level and low-level functions. The test suite grows in complexity as it progresses through the phases of compilation. The time running the suite takes also increases as the tests run increasingly more code. Running tests for the dynamic analysis phase is the most expensive currently.

Results Running our tests gives us feedback on the quality of our tool's source code. ScriptButler is able to parse and statically check the majority of valid source code with an insignificant amount of incorrect error reporting. However, the tool struggles when it has to parse code that has severe structural issues. This causes exception in the runtime of the tool usually resulting in a crash. There is no application with native support for counting of code lines in Rascal. However, we can use *cloc* and count it as if it was Java code to get an estimate. The core of our PuzzleScript implementation totals 9 files and 2317 lines of code. As a reminder, PuzzleScript's official implementation had 3 core files that totaled 5820 lines of code. In Table 4.2 we show the number of tests associated with each section and the percentage of those tests that pass. Note that some tests are used in multiple phases. If a phase does not have any test files, then that means it reuses mock data from previous phases.

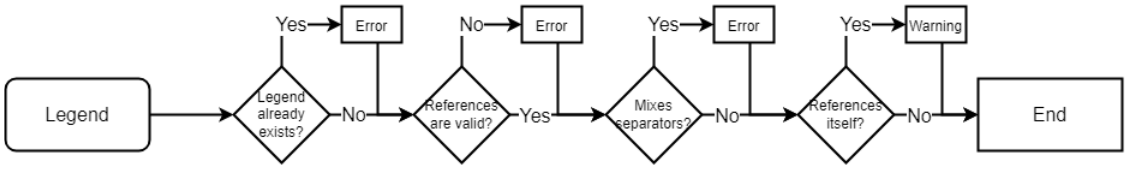
[4]`https://github.com/ClementJ18/Rascal-PuzzleScript/tree/main/src/PuzzleScript/Test`

Table 4.2: Results of our test suite

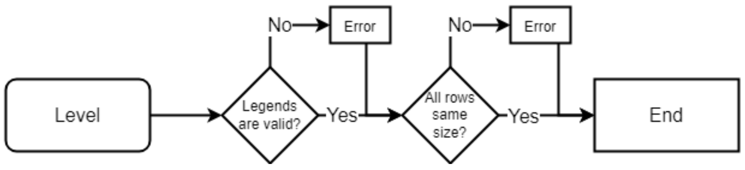
Phase	# of test files	# of tests	Tests passed	Comments
Parser	28	28	24/28	Failing tests all relate to comment parsing
Post-processing	0	16	15/16	Failing test relates to comment parsing
Checker	9	9	9/9	
Compiler	0	0	0/0	Verified through the engine behavior
Engine	11	12	10/12	Advanced games do not properly run
Dynamic Analyser	7	6		
IDE	1	1	1/1	
Interface	0	1	1/1	
Comment Parsing	7	7	6/7	Tests are relatively simple, not representative of all cases
General Parsing	83	1	72/83	Only tests whether games parse, no data validation



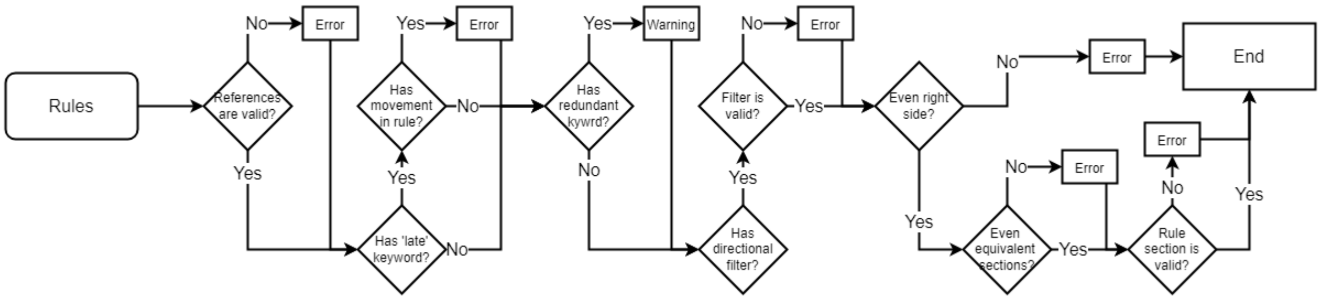
(a) **Flow diagram when validating a Layer**



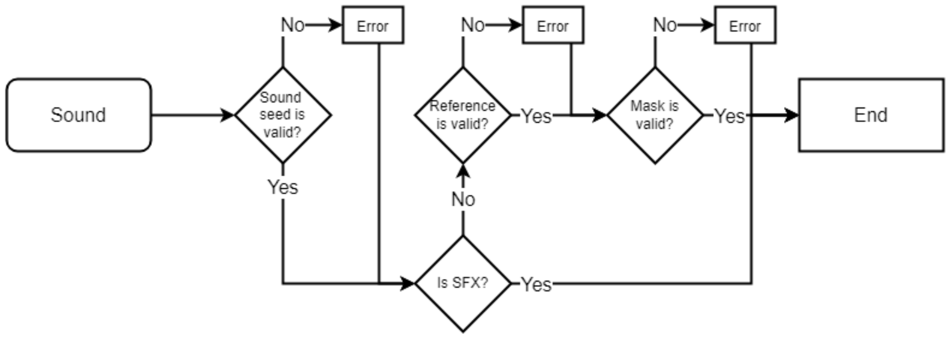
(b) **Flow diagram when validating a Legend**



(c) **Flow diagram when validating a Level**



(d) **Flow diagram when validating a Rule**



(e) **Flow diagram when validating a Sound**

Figure 4.13: Flow diagrams for checker validation.

Chapter 5

Case Study

In this chapter, we discuss the case study we conduct to validate our prototype tool, ScriptButler. The aim of the case study is to demonstrate that the functionality of the prototype tool actually helps designers analyze changes in a realistic evolution scenario. We mimic realistic evolution by taking a complex game designed by a user and removing lines of code to assess their impact. We verify the functionality of the tool with unit tests in Section 4.3 to make sure it fulfills the functional requirements, and now we validate it to make sure that those functional requirements fulfill the goal of the tool. We first describe the case that we study, here a PuzzleScript game, and why it is useful to study it. We then explain the methodology we use when conducting our case study and what results we expect. Finally, we describe the experimental results, reflect on them, and discuss how to what extent they validate our tool.

5.1 Timothy's Adventure

The game we are studying, Timothy's Adventure[44] has a simple concept. Steal the shiny objective, avoid the guards, and escape. You cannot escape until you have stolen the shiny objective. A sample of a level is shown in Figure 5.1.

The victory conditions are expressed in two lines that state that the objective must have been removed and that the player must be on the "exit" tiles. The narrative of this game is that the player is a sort of thief who must grab shiny objects (represented as the objective) and exit the room (by being on the exit objects). As with most PuzzleScript games, many of its mechanics and defeat/victory conditions are implied and must be discovered by the user. Users discover either through the use of common design choices or through messages at the beginning of the level.

For this case study, we extract the exact win conditions and game mechanics by observing the code. There are five key mechanics defined by about a dozen rules. The first mechanic is tied to the *Objective* Win Condition, it removes any Objective the player is adjacent to if the player presses the action key. This removes the object from being an obstacle to the first win condition. The next two mechanics relate to the defeat condition of the game. PuzzleScript does not provide the ability to create defeat conditions the same way it provides for victory conditions. By default, the defeat condition is a *'softlock'*, a softlock is the act of a player progressing to a level state that cannot end with a victory. The player can still modify the level and move around (as opposed to a *'hardlock'*) but will never be able to achieve victory. Sometimes, as with this particular game, the designers create explicit defeat conditions with rules that automatically restart the level if the player reaches a certain state. Two mechanics take care of this defeat condition, the first one states that if a Guard object is on the same X or Y axis as the Player object, then they will move one block in the player's direction and the second states that if a Guard is adjacent to the Player object then the Player object is replaced with another object representing that the player got "caught" and then the level restart at the next loop.

Finally, the game has two mechanics which represent tools that the player can use to solve certain levels. The first mechanic, introduced in level 5, allows the player to step on a Button object which transforms every Door object on the same X and Y axis into a different object which neither the player nor the guards can go through. This is usually used to stop guards from getting to the player but can also softlock the player if used without care. The second mechanic and the most complex one of the game is related to movement. It states that if the player steps on a Portal object, then they will be teleported to the equivalent portal object on the same X or Y axis given that it is not obstructed. This is used to allow

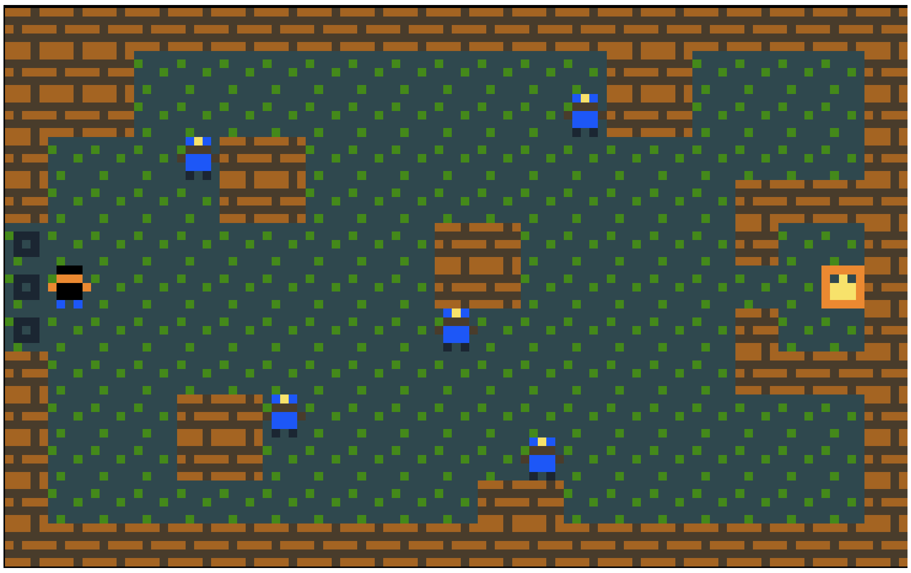


Figure 5.1: A level from the case game showcasing the general layout with the exit on the left, the objective on the right and the guards in blue

the player to move to rooms that are not connected to one another or to escape guards as they cannot use the portal. The game contains 14 levels, each of them created with the goal of gradually increasing the difficulty by introducing new challenges and new mechanics to solve those challenges. These levels are shown in Figure 5.2.

5.2 Approach

This case study is based on the following hypothesis:
"Removing part of an existing high-quality game damages the code and gameplay in such a way that these modifications represent realistic scenarios in the evolution of that game. By studying the effects of these modifications we can learn how the game's mechanics provide affordances to the player."

The damaged code and gameplay we refer to leaves the game technically playable. Players can still play, but their experience has been negatively impacted by the changes. The modifications we perform are intended to mimic the iterative game design. We describe selected modifications, each with a specific purpose. The modifications focus on three specific areas of a game that designers change commonly when evolving their game. We then run ScriptButler with these three modifications and observe what kind of feedback it provides. It is important that each modification is simple enough to be validated manually but complex enough to alter the gameplay in a subtle way. We also run ScriptButler on an unmodified version of the game to see what kind of issues already exist. We do not expect to find any major errors as the game runs and has been playtested by players.

Please note that we do not test our prototype tool on the published version of the game but on our own modified version, even in the case when we test on an "unmodified" version. This modified version includes additional newlines and the removal of redundant white spaces so as to make it simpler for our prototype grammar to parse it efficiently. Line numbers used in this chapter are all references to lines in our modified version which can be found on the repository of the project[1]

5.2.1 Victory Conditions

The first modification is intended to study the effects of changes in victory requirements. When changing victory conditions, the difficulty of a level is impacted, either the difficulty increases (potentially to an impossible level) or it decreases (potentially to a trivial level). We create a scenario that mimics the latter by removing the Win Condition that states that no *Objective* objects must be present in the level (line 210). Removing the condition removes the player's need to reach the Objective objects that are usually the main focus of a level. This alters the route they must take and since they now have fewer

[1] `https://github.com/ClementJ18/ScriptButler/tree/main/src/PuzzleScript/Test/Case`

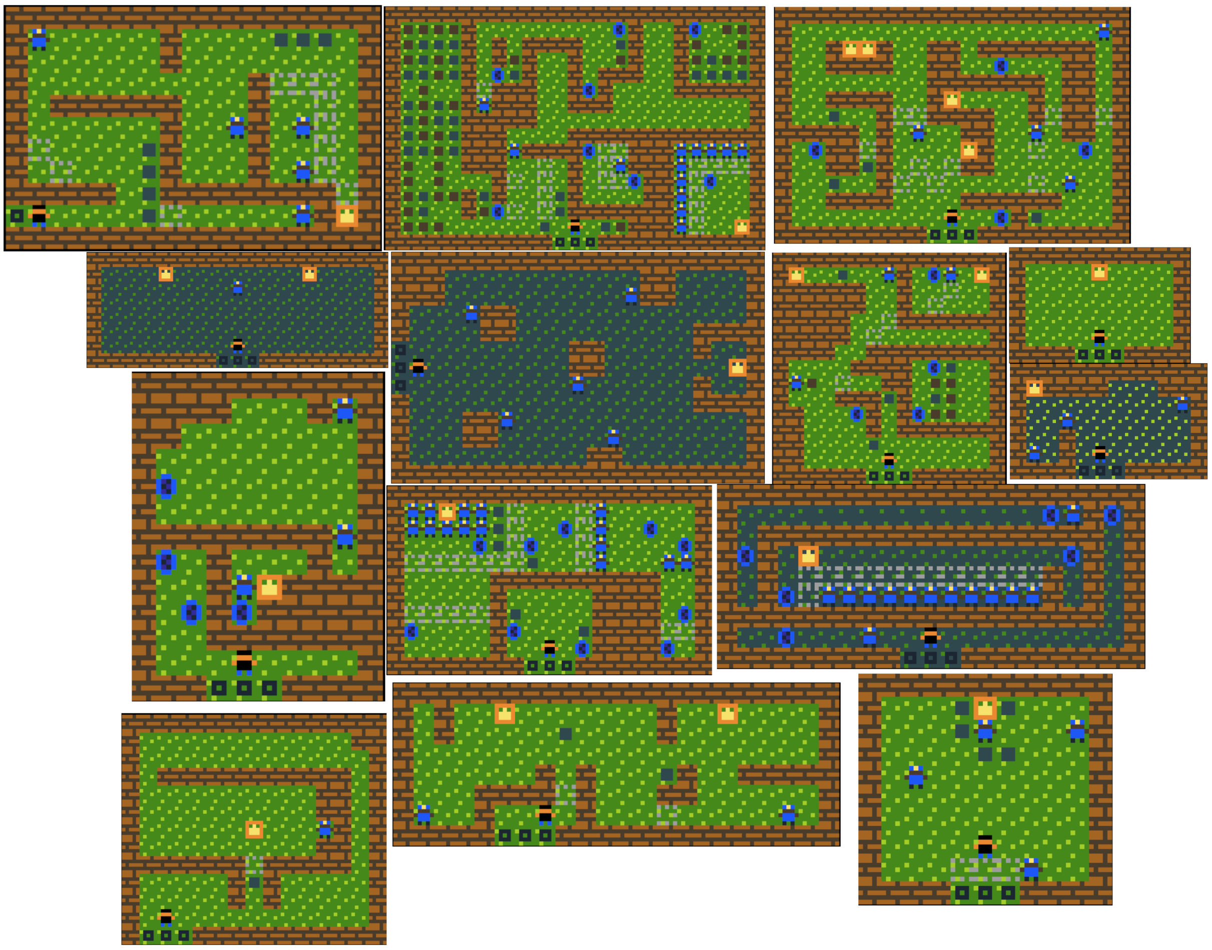


Figure 5.2: A collage of all 14 levels of Timothy's Adventure

obstacles to overcome it lowers the difficulty of the level to a potentially trivial level. Considering the setup of the levels in the case game, we expect this change to make it possible to solve all levels by continually going in one direction.

As we described previously, the case has two Win Conditions. The second Win Condition (line 211) requires that the Player object be stacked with an Exit object to emulate the player "leaving" the room. Removing that condition does not have the same impact since the requirement is mostly focused on pathfinding. However, we can change the object that the Player needs to be stacked with to meet the condition. Therefore, our second change to the victory condition is changing the condition from being `All Player on Exit` to `All Player on Background`. This small change, which in a real scenario could happen by accident, makes it so that every level instantly meets the condition. This effect is because 'Background' objects are the default 'ground' object the player passes over when traversing the level.

Both these modifications intend to trivialize the difficulty of the game. Another realistic scenario would be making the game impossible to solve by creating a victory condition that cannot be fulfilled. We create a similar scenario in the next modification by removing a mechanic of the case. However, we can first simulate an impossible level through the creation of conflicting victory conditions[2].

A conflicting victory condition is a result of software evolution, causing two victory conditions to be mutually exclusive. This case can occur when the mental models game designers have of their references falls out of sync. This human error can happen easily in PuzzleScript due to the fixed placement of components and the vertical space requirement pushing sections of of view easily. We simulate this scenario by adding two lines of code. The first addition is a new reference `'Object = Objective or Guardian or Exit'` (line 167). The first addition mimics a long, generic reference that game designers are more likely to forget. The second addition is a new condition (line 213) that states `'Some Object'`. The second addition mimics a victory condition that does not appear as conflicting due to naming convention but is when resolved.

[2]This modification is stored in Case/Modification4.PS to avoid contamination of other case files.

5.2.2 Game Mechanics

The second modification is intended to study the effects of changes in the mechanics of the game. If the mechanics of a game change, players may no longer have the same play affordances they previously had. Some of these play affordances may have been essential to overcome the obstacles presented to the player. Therefore, the removal of the mechanic may change whether or not the player can overcome the obstacles placed in its way. We recreate that scenario by removing the rule which allows the player to remove the Objective object (line 193). Removing that rule removes one of the core mechanics of the game which is that you can "steal" the shiny objective to satisfy the victory conditions. We expect this change to make it impossible to theoretically solve this level regardless of the setup. We chose specifically this rule over others because this rule does not relate to pathfinding. ScriptButler does not currently have the capacity to evaluate rules that interact with the player's pathfinding. The rule we are removing specifically relate to the player's ability to modify their environment to meet the victory conditions. We expect the result of this modification to be the impossibility to meet the "No Objective" victory condition simply because there is no way to get from a state where there are objects present to one where there are not any present.

The modifications we have done so far impact the game in a retroactive manner. We create scenarios where the designer has an existing, working game and they change it in a way that breaks it. In the next modification, we assume that the player has a working game and existing working levels.

5.2.3 Level Layout

The third modification is intended to study the effects of evolution in level design. When level design changes, this can create new obstacles or remove certain obstacles. This kind of problem can also occur when creating new levels. We focus on how removing objects changes the required player affordances for a level and how the mechanics not meeting those requirements can actively harm the gameplay. If a level requires the player to have a crate to reach an area but the mechanics do not provide a way to get that crate then the gameplay suffers. One possibility to mimic this change is removing the Objective object from the level, but this would be very similar to the previous two modifications. As such, we instead modify the levels by removing the *Exit* objects from certain levels. The expected result is that it will now be impossible to solve the level because there are no mechanics that allow the player to create Exit objects. An example of this modification is illustrated by 5.9.

Creating new levels, or modifying existing ones, is a challenging task. We aim for ScriptButler to provide support for that process so game designers can avoid simple mistakes that result in bad game design.

5.3 Results

Here we discuss the results of running and analyzing the various modifications described above with ScriptButler.

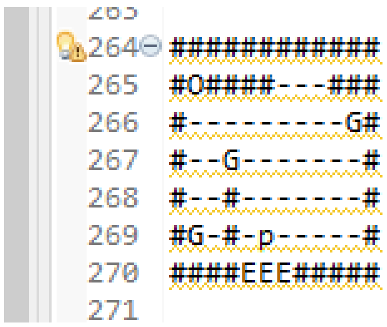
Our first test was to run the game on an unmodified version of the game, this test produced the following warnings:

- WARNING: Line 33 - Color **white** not used in object Background 4
- WARNING: Line 161 - Legend + defined but not used
- WARNING: Line 199 - Right side of the rule similar to rule on line 200
- WARNING: Line 200 - Right side of the rule similar to the rule on line 199
- ERROR: Line 202 - Objects in this section need to be able to stack but appear on the same layer.

We first want to address the error, a false positive as a result of a misinterpretation of PuzzleScript's design. The checker interprets the side of the rule with the error (`Background no Wall no Door_on`) as meaning that the three items should be able to stack. However, the *"no"* modifier means that none of the objects should be present for the rule to match. There is an argument to be made there that this is an unnecessary ambiguity since when objects are aligned in a part of the rule it usually means that they should be able to stack. In addition, there are already tools that can avoid this sort of ambiguity by creating an extra **or** reference that includes both, as can be seen in Figure 5.3. Ultimately, however, the error is caused by the aforementioned flaws of reverse engineering as a way of reconstructing design. We encountered more of such issues and resolved by iteratively testing and updating the design of our implementation. However, because we did not test this particular case, it slipped through our checks.

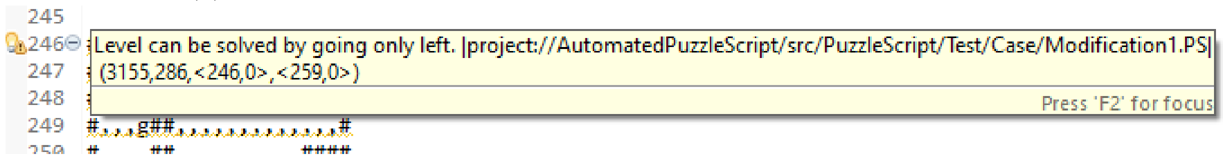
```
Obstacle = Wall or Door_on
[ > Temp Teleport | Background no Obstacle] -> [ Teleport > Player_stealth | Background]
```

Figure 5.3: Removing design ambiguity from the rule



```
############
#O####---###
#---------G#
#--G-------#
#--#-------#
#G-#-p-----#
####EEE#####
```

(a) Game designer is notified there is an issue with this component



Level can be solved by going only left. |project://AutomatedPuzzleScript/src/PuzzleScript/Test/Case/Modification1.PS|(3155,286,<246,0>,<259,0>)

Press 'F2' for focus

```
#,,,g##,,,,,,,,,,,,,#
```

(b) On hover, a message is revealed

Figure 5.4: Warning message caused by the first part of modification 1

The remaining warnings are all valid issues with the game. They outline either dead code or potential optimizations.

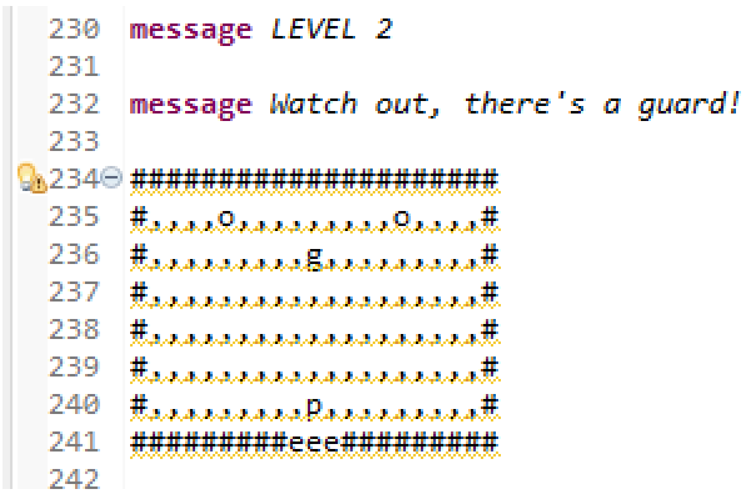
5.3.1 Modifying Victory Conditions

The first part of the modification is the removal of the 'No Objective' win condition. The results match our expectations. Upon running ScriptButler with the modification, we observed error messages relating to the fact that levels could be won by simply going in either to the left or down directions. These arrow messages can be seen in Figure 5.4. The error is part of our set of trivial solutions. Trivial solutions identify ways that a level could be solved that are so easy that they are considered to have a negative impact on the gameplay.

The problem we ran into with this modification is performance-related. To analyze trivial solutions, ScriptButler runs a portion of the game. Because our implementation is only a prototype the performance is not up to the user level yet. However, this performance problem does not impact the theory behind the trivial solution, minimizing its impact on validity.

The second part of the first modification is the replacement of the reference to the 'Exit' object with a reference to the 'Background' object. This change is small from a code perspective, it only modifies one reference, but has a large impact. This demonstrates how ScripButler can assist game designers in cases where they make simple errors with big consequences. The results of this modification can be seen in Figure 5.5. The warning is raised because all objects exist on top of a Background object by default, and since we remove the other victory condition then the player can achieve victory without moving. If we add the 'Objective' victory condition back in, the warning simply alerts us that the modified condition is already met but that the level itself still needs player interaction. The other message that can be seen in Figure 5.5 represents the basic metrics that the tool displays for the convenience of the game designer. The tool does not interpret those metrics but rather displays them raw for the game designer to draw their own conclusion. The reasoning behind this choice is that interpreting metrics is non-trivial and falls outside the scope of this project.

Modifying victory conditions is not the most common of evolution. A victory condition is usually tied very closely to the narrative the game designers create. It is more common for the levels and game mechanics to change. However, modifying victory conditions is still a valid example of game evolution and of its impacts on gameplay. Our tool generates error messages that aim to inform the game designer of the impact of their changes on the difficulty of their level. Game designers that use ScriptButler do not have to playtest their games after changing victory conditions in cases where the changes make certain



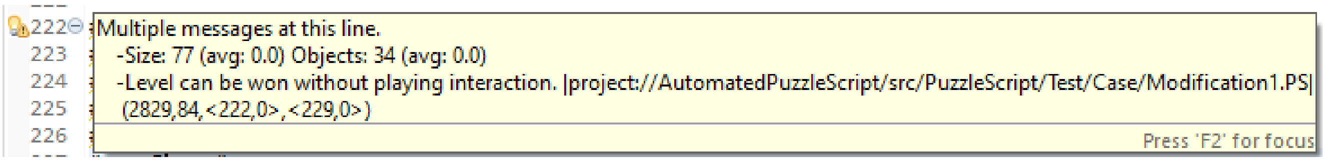
```
message LEVEL 2

message Watch out, there's a guard!

#####################
#....o.........o....#
#.........g.........#
#...................#
#...................#
#...................#
#.........p.........#
#########eee#########

```

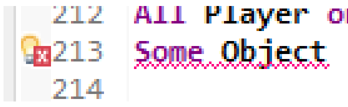
(a) Game designer is notified there is an issue with this component



```
Multiple messages at this line.
  -Size: 77 (avg: 0.0) Objects: 34 (avg: 0.0)
  -Level can be won without playing interaction. |project://AutomatedPuzzleScript/src/PuzzleScript/Test/Case/Modification1.PS|
  (2829,84,<222,0>,<229,0>)
                                                                                                   Press 'F2' for focus
```

(b) On hover, a message is revealed

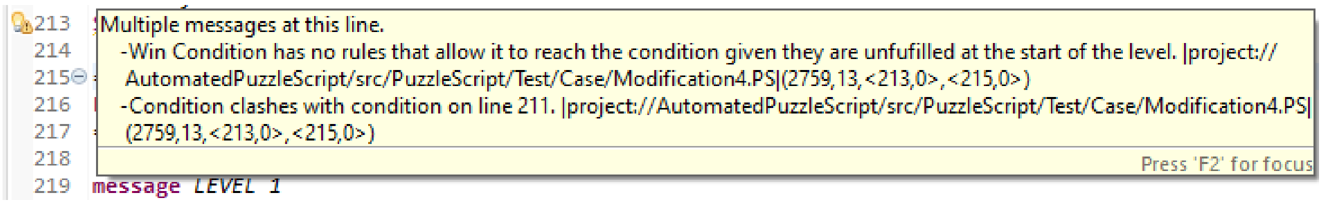
Figure 5.5: Warning messages caused by the second part of modification 1



```
All Player on
Some Object

```

(a) Game designer is notified there is an issue with this component



```
Multiple messages at this line.
  -Win Condition has no rules that allow it to reach the condition given they are unfulfilled at the start of the level. |project://
  AutomatedPuzzleScript/src/PuzzleScript/Test/Case/Modification4.PS|(2759,13,<213,0>,<215,0>)
  -Condition clashes with condition on line 211. |project://AutomatedPuzzleScript/src/PuzzleScript/Test/Case/Modification4.PS|
  (2759,13,<213,0>,<215,0>)
                                                                                                   Press 'F2' for focus
message LEVEL 1
```

(b) On hover, a message is revealed

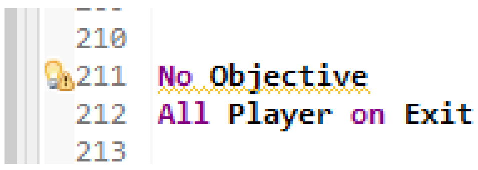
Figure 5.6: Warning messages caused by the third part of modification 1

levels impossible or trivial. Instead, the tool directly informs them and gives them the chance to adjust the code to better meet their design.

Finally, Figure 5.6 shows the results of the third part of modification 1. The third part was the addition of a new reference and victory condition, resulting in mutually exclusive conditions. The results match our expectations. ScriptButler informs the game designer that the conditions are mutually exclusive making the levels unwinnable. This change focuses on showing how ScriptButler can support human developers when they make human errors. The tool does not only support developers that make design decisions with negative impacts but also those that make code decisions with negative impacts. Additionally, this part of the modification also generates another message because there are not game mechanics that support fulfilling it. We discuss these kinds of issues further in the next section.

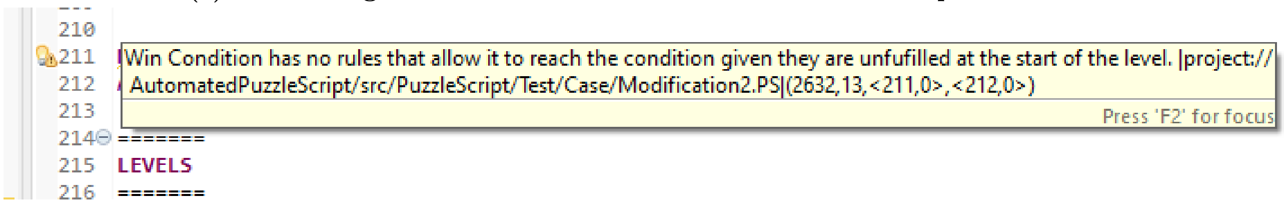
5.3.2 Modifying Game Mechanics

The second modification is the removal of the rule use to remove 'Objective' objects from a level. The results of the second modification match our expectations. Running our prototype with the modification generates warnings next to the 'No Objective' win condition. These warnings state that, assuming the condition to be false at the beginning of a level, there is no way for the condition to become true. These



```
No Objective
All Player on Exit
```

(a) Game designer is notified there is an issue with this component



```
Win Condition has no rules that allow it to reach the condition given they are unfulfilled at the start of the level. |project://
AutomatedPuzzleScript/src/PuzzleScript/Test/Case/Modification2.PS|(2632,13,<211,0>,<212,0>)
                                                                                                    Press 'F2' for focus
=======
LEVELS
=======
```

(b) On hover, a message is revealed

Figure 5.7: Warning message caused by modification 2

messages can be seen in Figure 5.7. ScriptButler achieves this by analyzing the rules and seeing if there are any potential chains that lead from the object being present to the object no longer being present.

In this modification, ScriptButler presents the game designer with a complex problem. The tool does the tedious work of figuring out if there is a problem, but it does not offer a solution. Game designer need to have the necessary knowledge to deal with each message. In addition, game designers will also have a learning curve related to deciphering the meaning of the messages. However, this curve can be softened in future works studying the wording of the messages. Since they are centralized, changing the wording is a trivial task.

Suggesting a solution to each problem is nontrivial as the error can reside in different areas. For instance, in this case, the error can be with either the WinCondition or the Rules. Part of the solution could be simply stating that fact. If the game designer knows where to place their solution they are more likely to find a fix. We do this partially by placing the message near the problematic line. Another possibility is suggesting game mechanics that could potentially fix the issue. The tool can be used to test various fragments of code that could then be suggested to the user. This solution relates more to the realm of PCG and as such, is left as future work.

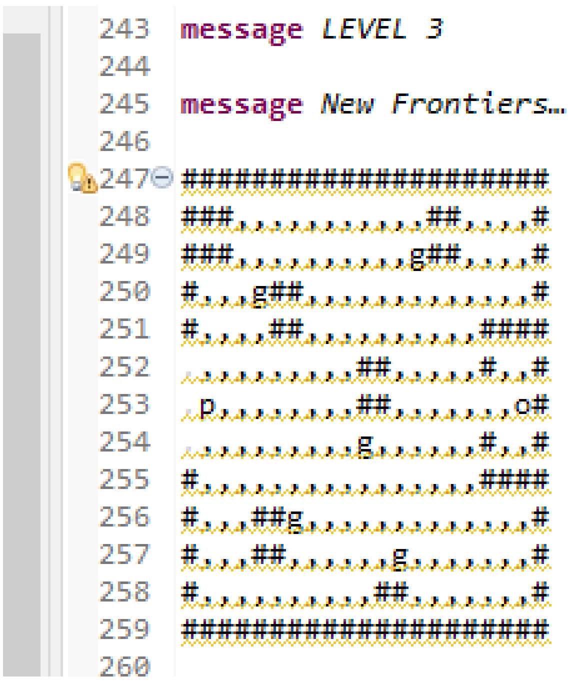
5.3.3 Modifying Level Layouts

The final modification is the remove of 'Exit' Objects from certain levels. The results of the third modification match our expectations. It is a bit trickier than we initially intended because of the *Player* object mentioned in the victory condition. This object has a set of "built-in" rules that are part of the PuzzleScript engine relating to movement. We add an exception to the rules to prevent this issue from occurring. With the issue resolved, ScriptButler checks different rule chains in an attempt to see if any results in an Exit object spawning. As expected, the tool does not find any chain that meets the requirement and as such displays an error message next to the problematic levels as can be seen in Figure 5.8.

We validate this check by temporarily adding a rule to the modification that would allow the player to spawn an exit. This successfully erases all error messages relating to that issue and confirms that the tool works as intended. By conducting this test, we showcase how ScriptButler can be used to support game designers in creating levels. ScriptButler warns the designer when they create levels that are impossible to solve. However, we always assume that this impossibility comes from mechanics rather than pathfinding.

5.4 Conclusion

In our case study, we have demonstrated that ScriptButler can empower designers in realistic evolution scenarios. Using ScriptButler level designers can study the effects of any change on the code quality, and reason about the effects on gameplay. Being able to detect these changes without manually playtesting should help the game designers focus the playtesting on the more complex areas of gameplay. Script-Butler's extensible nature leads us to believe that a similar approach could be applied to many other

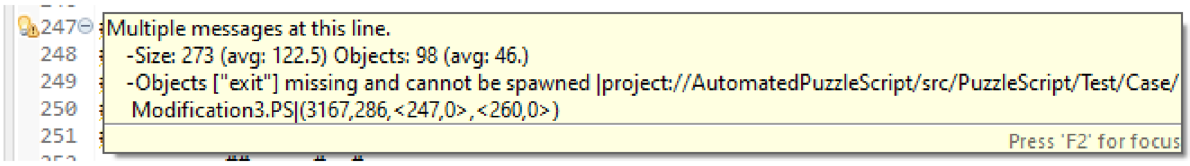


```
message LEVEL 3

message New Frontiers…

#####################
###,,,,,,,,,,,##,,,,#
###,,,,,,,,,,g##,,,,#
#,,,g##,,,,,,,,,,,,,#
#,,,,##,,,,,,,,,,####
,,,,,,,,,,##,,,,,#,,#
,p,,,,,,,,##,,,,,,,o#
,,,,,,,,,,g,,,,,,#,,#
#,,,,,,,,,,,,,,,,####
#,,,##g,,,,,,,,,,,,,#
#,,,##,,,,,,g,,,,,,,#
#,,,,,,,,,,##,,,,,,,#
#####################
```

(a) Game designer is notified there is an issue with this component



```
Multiple messages at this line.
  -Size: 273 (avg: 122.5) Objects: 98 (avg: 46.)
  -Objects ["exit"] missing and cannot be spawned |project://AutomatedPuzzleScript/src/PuzzleScript/Test/Case/
   Modification3.PS|(3167,286,<247,0>,<260,0>)
```

(b) On hover, a message is revealed

Figure 5.8: Warning messages caused by modification 3

```
(BEFORE)
###########
#....O....#
#.........#
#.........#
#.........#
#....P....#
####EEE####

(AFTER)
###########
#....O....#
#.........#
#.........#
#.........#
#....P....#
####...####
```

Figure 5.9: Removing exit objects from level

scenarios. The tool cannot currently detect changes that affect the quality of a gameplay from a pathfinding perspective. Changes that impact alter pathfinding obstacles and impact player movement are not detected in a meaningful way. However, there is potential for pathfinding algorithms to be implemented to allow for automated testing of solutions.

Ultimately, ScriptButler generates error messages that bring potentially harmful changes to the designer's attention. These errors are detected continuously as the design programs their game. This live feedback increases the efficiency of designers and reduces the time consumed by playtesting by reducing the need for playtesting. The extensible nature of the tool allows designers to fine tune the analysis to their specific project.

Chapter 6

Discussion

In this chapter, we discuss the results of our project.

6.1 Design Formalization

Our first contribution is a document outlining the syntax of a PuzzleScript file. We studied PuzzleScript's design by observing how the games behaved under various inputs and by going through the codebase. Based on this we created a document containing details on the design of PuzzleScript and the conclusion of our observations. This document is Chapter 3 of the thesis. The aim of the document was to create documentation on the syntax structure of PuzzleScript so that we could parse it using a formal grammar. The document is complete enough for others to create their own parser and documents the corner cases of PuzzleScript. It lays out the basic constraints that certain objects must abide by. For all the constraints of PuzzleScript, refer to Appendix A.

6.2 ScriptButler

We then used our formalization of PuzzleScript's design to create our own design of an implementation which we then implemented. Our goal was to create a more extensible and maintainable implementation of PuzzleScript. This prototype is not a feature-complete implementation of PuzzleScript as can be seen in Table 6.1. However, it is complete enough to provide feedback on a wide range of games. This range includes published games.

The grammar we created using our design document can parse most games, but it struggles with parsing comments. Currently, we strip the comments before parsing similarly to the JavaScript implementation. In the future, parsing comments could prove an interesting source for metrics. Ultimately, the grammar is a very literal representation of a PuzzleScript file that can be easily extended and modified to alter PuzzleScript's design. The static checker is fully functional but does not currently check for the errors of the original implementation. This is an intentional design decision, we instead focused on implementing new types of errors within our checker that the original implementation did not check for. There are no obstacles to reimplementing all the error messages but they did not provide a lot of value for this project. The compiler and engine work on a wide range of simple games. We do not currently implement complex features such as real-time or rigid bodies. These features are complex to implement and not commonly used in games. We design and implement the engine as a collection of plugins that can easily be exchanged. This design decision should make it easier to optimize in the future.

The final result is a partial reimplementation of the PuzzleScript engine. This implementation can run most simple games, is more extensible and more maintainable than the JavaScript one. We achieve this by reducing the coupling and code density. We leveraged Rascal features to make the compiled Rules more readable and verify the functionality using the extensive test suite discussed in 4.3.

6.3 Supporting Game Designers

To support game designers, we implemented three additional components into our tool: 1) an IDE, 2) an interface for running games and playtesting, and 3) a dynamic analyzer for automated gameplay testing.

Figure 6.1: Feature Comparison between the original implementation and our implementation.

Category	Feature	JavaScript	Rascal
Editor	Text Editor	Yes	Yes
	Syntax Coloring	Yes	Yes
	Messages	No	Yes
	Game frame	Yes	Yes
	Autocomplete	Yes	No
	Export	Yes	No
	Share	Yes	No
	Save states	Yes	No
	Keyboard control	Yes	Yes
	Debug	Yes	Yes, more extensive
Parser	Error Checking	Yes	Yes, generic error
	Grammar	Implicit	Formal
Checker	Error Checking	N/A	Yes
	Errors	101	72
	Warnings	19	11
	Error Isolation	No, cascades	Yes
Compiler	Error Checking	Yes	None
	Prelude	Implemented	Implemented up to Checking
	Objects	Implemented	Implemented
	Legend	Implemented	Implemented
	Sounds	Implemented	Implemented up to Checking
	Layers	Implemented	Implemented
	Rules	Implemented	Implemented, mostly working
	Win Conditions	Implemented	Implemented
	Levels	Implemented	Implemented
Engine	Randomness	Implemented	Implemented
	Rigid Bodies	Implemented	Not Implemented
	Real time	Implemented	Not Implemented
	Rule Loops	Implemented	Not Implemented

The IDE allows game designers to edit their games, similarly to the in-browser editor of the original JavaScript implementation. It provides syntax coloring, an outline, a contextual menu and displays error messages. The IDE allows designers to access the feedback generated by ScriptButler in a user-friendly way as it directly attaches it next to the problematic line. We provide several other features that the original editor does not provide. However, we do not support the extra features that the original editor had such as the ability to share the game on GitHub or to export it as an HTML page. The reason is that does features do not have an impact on game quality. The only feature from the current editor we would potentially look at reimplementing is the 'Rebuild' feature. This feature re-compiles the game while it is running, allowing changes to game mechanics, victory conditions, and object appearance to take effect immediately.

The interface implements the ability for designers to run and play their games. This component was separated from the IDE, unlike the original implementation, so that we could expand it. The interface provides the game designers with tools that can be used to debug their game. An important difference between our implementation and the original's is that our editor does not use keyboard buttons but rather requires the player to click on UI buttons. This error should be fixed in future works when salix supports keyboard events. Future works could also split the interface between the 'game designer view' and the 'tool developer view' to enhance the purposes of both.

Finally, the dynamic analyzer is similar to the static checker. It generates error messages that are then processed by the IDEs and displayed to game designers. However, the messages relate to the quality of the gameplay rather than the quality of the code. The dynamic analyzer provides the game designer with feedback on the playability of their game. In terms of playtesting, this feedback is shallow. However, we purposefully keep the feedback shallow so that we can provide it rapidly, it is an intentional design decision.

6.4 Research Questions

Here we discuss how our work contributes to answering our research questions. We begin by discussing the subquestions and then we describe how they help us answer our main research question.

6.4.1 RQ1.1

As we have discussed previously, PuzzleScript rules are very closely tied to the affordances they provide players. The match and replace patterns almost literally define game mechanics. This simplification allows us to more easily study the effect of code changes on game mechanics. We show this in the dynamic analyzer of ScriptButler. When the game designer removes a piece of code, the tool provides feedback on how it impacts the game mechanics required to solve certain levels or meet certain victory conditions. Despite being relatively simple, PuzzleScript is a powerful and popular game language. This popularity ensures that we study features that meet game designer's need. We could study a programming language, this would make the code easier to analyze. However, studying a programming language does not represent a realistic scenario. Game designers use game engines, so we have to study a game engine.

6.4.2 RQ1.2

When iteratively designing, game developers evolve the software of their game over time, sometimes causing changes with a negative impact on the quality of gameplay. Game designers can reduce the impact of these changes by putting in place mitigation strategies such as playtesting.

These strategies are time-consuming and unreliable. Playtesting requires a human to play through the entire game and actively attempt to break it. Sometimes, AI can be used to automatically play through the game and try to detect gameplay issues. However, this presents the same issue of how incredibly time-consuming this approach is, both in terms of time investment in putting together an AI that can play a game and having the AI test the game. Therefore, the game designers put in place tests to verify gameplay. These tests work similar to unit tests, they provide an input to the game and expect an output. This output is usually a specific game state. These tests allow for automated game design. However, the tests need to be adapted as the requirements change, creating additional dependencies. The tests can only be created within the context of a specific game and cannot be generally applied to a game engine.

6.4.3 RQ1

We now know how PuzzleScript code relates to the code and why that relation is useful. We know strategies that game designers can put in place to mitigate the negative effects of software evolution. Therefore, we now apply meta-programming techniques to implement these mitigation strategies by leveraging the connection between code and game mechanics.

We propose a system of solutions that can be generally applied to any game written using Puzzle-Script. These tests provide rapid feedback in a live programming context. Instead of having to run a long test suite, the tool responds live to changes and provides the designer feedback live. This system of solutions is implemented as a tool, called ScriptButler. The tool is more efficient than a player because it does not require a full game. With only a few lines of code, it can already start providing feedback on the game.

We note that this does not make human testers redundant. Human testers are much better are detecting defects with the visual aspect of the game. Humans are more creative, they can break the game in more interesting ways. Although the extensible nature of our tool means that if new, general ways of breaking game are discovered, they can be incorporated. Finally, humans actually experience the game as a whole and can provide feedback on that experience. Our tool simply tests a few interconnected components. ScriptButler empowers game designers by reducing their dependence on playtesting. The playtesters can then focus on the testing more complex problems that actually relate to the way humans experience a game.

6.5 Threats To Validity

Game Designer Involvement No game designers were actually involved in the process of creating our tools, but we still want our tools to be useful to game designers. This is a threat to validity in that we were not specifically told by game designers that the issues we identified are indeed issues. Instead, we relied on data we gathered from personal experience and from observing the evolution of PuzzleScript games. We mitigate this as much as possible by grounding our research using a case study. The case study mimics scenarios of realistic software evolution. We apply ScriptButler to those scenarios to see to what extent the feedback provided evaluates the impact of those changes. Since the case is simple, we can also validate the errors raised by ScriptButler manually.

Reverse Engineering Reverse engineering is not a perfect method for extracting a design from source code. This is illustrated in our case study where we misinterpreted the observed behavior causing an error that does not actually happen in the JavaScript implementation. This error happens because we did not specifically test for this kind of behavior when reverse-engineering the engine. We partially mitigated this issue by conducting extensive testing on our engine using published games, which we first checked against the original PuzzleScript engine. However, this method is not completely reliable since we also try to test for new types of errors within our implementation.

Incomplete Engine Our implementation of PuzzleScript does not cover all the features of the original implementation. As such, there is a possibility that the features that remain unimplemented cannot be implemented using Rascal. However, we strongly believe that this will not occur as we have successfully implemented rules, the most complex PuzzleScript features.

Academic Prototype ScriptButler is an academic prototype. Additional work is required to make the tool suitable for professional work. However, ScriptButler has sufficient feature coverage to be used within the context of our project. We needed an extendable implementation of PuzzleScript which allowed us to run PuzzleScript games and perform dynamic analysis on them and the tool met that need. If the tool was required to meet the needs of professional game developers, work would have to go into optimizing the engine and covering the remaining corner cases. Additionally, the formal grammar we have created is still unable to parse more complex combinations of lexicals, especially when combined with code that is heavily commented. Future work should look into making it possible to parse comments as they may contain interesting possible metrics to judge code and gameplay quality[45].

Chapter 7

Related work

Here we discuss works related to our project.

7.1 Automated Game Design

In this section, we discuss the various works that are related to the automated game design aspect of our project. First, we go over projects that have similar goals to ScriptButler and contrast the differences. In the last section, we delve into the AI perspective of AGD. This perspective is not the focus of our thesis, but it is an important part of AGD.

7.1.1 AI-Based Automation

AI-Based automation is used in two distinct ways in Game Design and we will briefly discuss each. We do not discuss AIs as part of a game but rather AI as a tool for game design and development. Although AI as part of games is an interesting area as they provide additional replayability and randomness to a game. That perspective was briefly covered in our Procedural Content Generation section.

The first way that AI-based automation is used is as a way to support developers in their endeavors through such things as automated mechanics testing. Automated mechanics testing is similar to playtesting in that a player goes through the level insuring that certain actions result in certain level states. However, the method does not provide the ability to analyze the experience. AI-based playtesting is a relatively new area of game design that involves both physical games[46] and digital games. The concept is having an algorithm crawl the game's various obstacles and come up with solutions much more methodically than a human could.

The second way that AI-based automation is used is as a way to support the designers in their endeavors through such things as AI-generated content. This can include things like automatically generating levels based on game mechanics, generating game mechanics based on a preset, or generating a narrative based on rules. In these cases, the AI system influences the mechanics and aesthetics of the game. Game design provides context to the AI which in turn generates affordances for the game based on the domain of the game as can be seen in Figure 7.1.

This concept is well illustrated in the paper by Eladhari[47] which discusses the concepts of context and affordances before relating them to the iterative design loop. Another paper by Eladhari[48] discusses a technical framework called *Mind Module* which aims to model emotion in the world of games. This module generates parts of the game based on the player responses.

7.1.2 Examples of AGD

PlaySpecs A paper by Osborn et al.[25] connects the concepts of automated analysis with software verification. This is similar to our approach where we apply techniques used in software engineering to perform automated game analysis. The author proposes the tool *Playspecs* to test PuzzleScript games. Any puzzle game can define Playspecs conditions that must hold for any found solution on each level. A solution that violates this constraint notifies the developer. However, PlaySpecs requires play traces to analyze. In the paper, this is done by generating solutions using a heuristic search. In other cases, this could be also done manually through playtesting. PlaySpecs' target audience is game engines and engine developers.

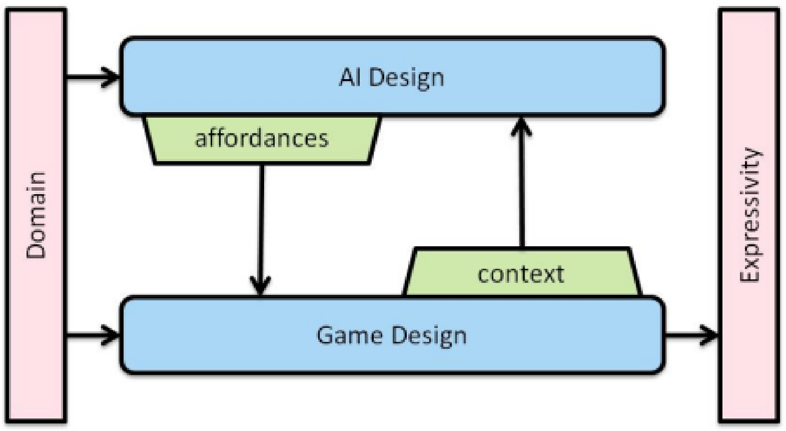


Figure 7.1: Illustrating the process of AI-based game design (adapted from Eladhari[47])

Ludoscope Dormans proposes the tool *Ludoscope* as a way to analyze how level space relates to level missions. They illustrate those concepts in "Adventures in level design"[49] in the context of a *The Legend of Zelda* game. Ludoscope is a general-purpose tool for generating content using transformational grammars. Ludoscope is extensible and in a validation follow-up validation experiment by Dormans[50]. The tool can then be extended based on project-specific needs. Ludoscope touches on the concepts of automated game design and procedural content generation. Game designers define constraints for generation and the tool provides feedback after the generation. ScriptButler does not generate any content but it is similarly extensible.

Micro-Machinations *Micro-Machinations*[51] (MM) is a textual and visual programming language that extends *Machinations*. Van Rozen and Dorman[52] propose a live programming approach as an embeddable MM library. This approach of rapid prototyping is similar to our approach with ScripButler. However, our approach does not focus on providing alternative methods for creating those games, only feedback on existing code.

7.2 PuzzleScript

PuzzleScript's popularity and features generate a lot of interest for developers looking to research game design. Because of that, a few other works have been done relating to it. In this section, we present these different works and how they relate to our project.

We want to mention the project by GitHub user vexorian[53] which implements Breath-First search as a PuzzleScript game. It does not necessarily relate to this project but shows an interesting way of applying Puzzlescript in education. Users can create their own levels to test how breath-first search interacts with it and study the Rules of the game to understand how it works in theory.

The project repository recommends running the project on a fork of PuzzleScript which includes additional features[1] such as the ability to jump to specific levels. It is another of those forks that extend PuzzleScript with very small specific changes to suit their needs[54].

7.2.1 Other Implementations

Here we compare the features of different implementations of PuzzleScript which we previously discussed in Section 2. The comparison can be seen in Table 7.2. The bottom line is that none of these implementations really focus on changing the design process of PuzzleScript game, they are all focused on simply reimplementing PuzzleScript in a different language for convenience. The Rust implementation specifically mentions that it handles rules differently from the original engine but that seems to cause issues. We did not compare PatternScript[29] because it is an implementation of PuzzleScript in JavaScript, something we already knew possible since the original implementation is in JavaScript.

[1] `https://pancelor.com/PuzzleScript/editor.html`

Figure 7.2: Feature Comparison between the original implementation and other implementations.

Category	Feature	JavaScript	C++	Rust	C
Editor	Text Editor	Yes	No	No	No
	Syntax Coloring	Yes	No	No	No
	Messages	No	No	No	No
	Game frame	Yes	No	No	No
	Autocomplete	Yes	No	No	No
	Export	Yes	No	No	No
	Share	Yes	No	No	No
	Save states	Yes	No	No	No
	Keyboard control	Yes	No	Yes	Yes
	Debug	Yes	No	No	No
Parser	Error Checking	Yes	Yes	Yes	Yes
	Grammar	Implicit	Implicit	No, JSON-based	Implicit
Checker	Error Checking	N/A	N/A	No	Yes but no line number
	Errors	101	101	N/A	101
	Warnings	19	19	N/A	19
	Error Isolation	No, cascades	No, cascades	N/A	No
Compiler	Error Checking	Yes	Yes	N/A	Yes
	Prelude	Yes	Mostly missing	Yes	Yes
	Objects	Yes	Implemented	Yes	Yes
	Legend	Yes	Implemented	Yes	Yes
	Sounds	Yes	No	Yes	No
	Layers	Yes	Yes	Yes	Yes
	Rules	Yes	Yes, commands missing	Yes	Yes, mostly working
	Win Conditions	Yes	Yes	Yes	Yes
	Levels	Yes	Yes	Yes	Yes
Engine	Randomness	Yes	No	Yes	Yes
	Rigid Bodies	Yes	No	Yes	No
	Real time	Yes	No	Yes	Yes
	Rule Loops	Yes	No	Yes	No

7.2.2 Solvers

Here we discuss how solving algorithms are used in PuzzleScript. We have found three instances of AI being applied to PuzzleScript with the purpose of automated playthrough.

A paper by Chong-U Lim and D. Fox Harrell[27][2] present a repository with a series of algorithms for solving PuzzleScript levels. However, the most interesting part is that the project is not simply implementing an AI but provides an interface for anyone to implement their own algorithms. The similarity between this paper and our own project is interesting, except that instead of providing a wrapper for PuzzleScript we provide our own implementation with its own endpoints. A project by Lauren Cunningham[55] uses the extension to implement a Monte Carlo Tree Search and shows how exactly one can use such tools.

A project hosted by GitHub user marcosdon[56] is a full PuzzleScript editor and as such is very easy to play around with. The project adds a "Solve" button to the top bar which runs through the

[2]`https://github.com/icelabMIT/PuzzleScriptAI`

level extremely fast and provides a series of movements to solve the level. It only provides one solution, however, so when there are multiple ways to solve a level, it should be interesting to see which one it picks. The interface uses PuzzleScript's code but the algorithm itself is quite simple, as such it gets easily stuck. We tested on the game we selected for our case study and it became essentially stuck at the third level, always getting caught by the guards. Perhaps it needs more PuzzleScript-specific awareness. This is where the previous project is useful as it allows users to extend the algorithms themselves.

Chapter 8

Conclusion

In this thesis, we have posed the question of how we could support game designers by leveraging meta-programming techniques. The solution we have proposed to address this question is a system of solution for relating gameplay and code. To implement this system, we have documented PuzzleScript design in order to simplify the creation of our own implementation. This extensive documentation of PuzzleScript's design is the first contribution of our thesis. The document details the the syntax of a PuzzleScript file enabling the creation of a parser. This document also goes thoroughly into the corner cases of PuzzleScript's syntax.

We have used our documentation of PuzzleScript's design to design our own implementation. We have addressed the technical obstacles that we faced when trying to use an existing implementation by leveraging good software engineering practices to make the implementation more maintainable and extensible. The result is the second contribution of our thesis: a general-use tool, ScriptButler, for parsing, running, and analyzing PuzzleScript games. We have implemented our system of solutions for dynamic analysis into ScriptButler. This implementation allows game developers to get live feedback on the impact of their software changes on gameplay. We have also further extended ScriptButler with IDE support and a debugging interface.

We conclude that relating game code and game mechanics is a valuable way of providing feedback to game designers. We present ScriptButler as a tool for exploring that relation that leverages meta-programming. We believe there is much left to discover and we hope that others can extend the tool to conduct similar research by leveraging PuzzleScript.

8.1 Future work

Of course, our work can be extended and improved in several ways. We discuss several directions for future work.

8.1.1 Level Solver

Our tool has been designed with the goal of leveraging meta-programming principles in the context of Automated Game Design (AGD). The approach we have focused on separating from the current AI-based approaches to AGD. However, our tool is not closed to the possibility of combining meta-programming and AI, as such, future work could look into the possibility of adding level solvers as a way of conducting a more thorough gameplay analysis as a complement to our rapid-but-shallow approach.

ScriptButler already exposes the necessary endpoints for an automated player, so the technical requirements for coupling one with the tool are quite low. This would be mostly a research endeavor, understanding how AI can fit in our concept of rapid feedback. Playtesting the entire game on every build would not be viable, so work would have to be done in understanding how the game can be broken down into parts and then executed based on the quality of feedback provided.

8.1.2 Procedural Content Generation

ScripButler transforms PuzzleScript code into Rascal data structures. Future works could extend the tool to make it possible to generate PuzzleScript code from the Rascal data structures. As such, ScriptButler could be used from a PCG perspective to study PuzzleScript. The goal of such a project could be try and generate individual levels or rules that build up into a working game.

Levels and rules generated by future works could also be evaluated using our existing dynamic analysis module. If a level can be solved trivially, this could detected by the tool. Other trivial solutions might also arise from generating the wide range of content, improving the tool's analysis capabilities.

8.1.3 Audio-Visual Research

Part of a game's quality is how it looks and sound. This a facet of game design that we did not touch upon in this project. However, there is room to apply research in that area to our tool. For instance, some combinations of colors create contrasts that are painful to the eye. What if our tool could detect such an instance and provide a warning.

However, such a project would first have to understand how graphics and sounds interact within game design. With a few lines of code, a plugin could be added to ScriptButler to allow for such analysis.

8.1.4 Automated Game Design for Serious Games

Serious games are more complex to design than regular games because they have an educational component[57]. PuzzleScript games can potentially be used in an educational context. We have found a game that demonstrates how breath-first search behaves using a PuzzleScript game[53]. It is not completely clear whether this is an effective method of teaching as PuzzleScript games require more intuition than most other games to play. This is one of the weaker future works, but we wanted to introduce it anyways.

Acknowledgements

When I first started on this project I was not sure where I was going. I knew I wanted to do something with video games, which have always been a personal interest of mine both from an entertainment perspective and a research perspective.

There was an idea that slowly evolved, helped by the work of others, feedback from those around me and my own tinkering. This project has been one of the biggest undertakings of my life and I would not have managed to finish it if not for the support of my peers.

I am very grateful for the help of my supervisor, Riemer van Rozen, for pushing me into this project when I first approached him. He helped me understand what I wanted from my master project and helped me shape a project that met these desires. He met with me every week, sometimes more, for nearly a year, always asking the hard questions that forced me to leave no stone unturned.

I want to thank my family, and especially my mother who, without knowing anything about meta-programming, never stopped asking me questions on my work in an unorthodox variant of rubber ducking.

I also want to thank Georgia Samaritaki and Youri Reijne, the two other students supervised by Riemer. Their feedback when we discussed our projects was very helpful.

Articles

[2] M. Ostrowski and S. Aroudj, "Automated regression testing within video game development," *GSTF Journal on Computing (JoC)*, vol. 3, no. 2, pp. 1–5, 2013.

[3] J. Warren, "Tiny online game engines," Lecture Notes in Computer Science, vol. 11836, T. Alpcan, Y. Vorobeychik, J. S. Baras, and G. Dán, Eds., pp. 1–7, 2019. DOI: `10.1109/GEM.2019.8901975`. [Online]. Available: `https://doi.org/10.1109/GEM.2019.8901975`.

[4] R. van Rozen, "Languages of games and play: A systematic mapping study," *ACM Comput. Surv.*, vol. 53, no. 6, 123:1–123:37, 2021. DOI: `10.1145/3412843`. [Online]. Available: `https://doi.org/10.1145/3412843`.

[5] E. Butler, E. Torlak, and Z. Popovic, "Synthesizing interpretable strategies for solving puzzle games," S. Deterding, A. Canossa, C. Harteveld, J. Zhu, and M. Sicart, Eds., 10:1–10:10, 2017. DOI: `10.1145/3102071.3102084`. [Online]. Available: `https://doi.org/10.1145/3102071.3102084`.

[6] M. Cook, "A vision for continuous automated game design," AAAI Workshops, vol. WS-17, 2017. [Online]. Available: `https://aaai.org/ocs/index.php/AIIDE/AIIDE17/paper/view/15850`.

[7] ——, "Software engineering for automated game design," pp. 487–494, 2020. DOI: `10.1109/CoG47356.2020.9231750`. [Online]. Available: `https://doi.org/10.1109/CoG47356.2020.9231750`.

[8] T. Nummenmaa, A. Kultima, K. Alha, and T. Mikkonen, "Applying lehman's laws to game evolution," R. Robbes and G. Robles, Eds., pp. 11–17, 2013. DOI: `10.1145/2501543.2501546`. [Online]. Available: `https://doi.org/10.1145/2501543.2501546`.

[9] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri, "Challenges in software evolution," pp. 13–22, 2005. DOI: `10.1109/IWPSE.2005.7`. [Online]. Available: `https://doi.org/10.1109/IWPSE.2005.7`.

[10] P. Mirza-Babaei, N. Moosajee, and B. Drenikow, "Playtesting for indie studios," pp. 366–374, 2016. DOI: `10.1145/2994310.2994364`. [Online]. Available: `https://doi.org/10.1145/2994310.2994364`.

[11] F. Brühlmann, G.-M. Schmid, and E. D. Mekler, "Online playtesting with crowdsourcing: Advantages and challenges," *CHI 2016 Workshop: Lightweight Games User Research for Indies and Non-Profit Organizations*, 2016. DOI: `10.5451/UNIBAS-EP43799`. [Online]. Available: `https://edoc.unibas.ch/43799/`.

[12] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design science in information systems research," *MIS Q.*, vol. 28, no. 1, pp. 75–105, 2004. [Online]. Available: `http://misq.org/design-science-in-information-systems-research.html`.

[13] R. Cole, S. Purao, M. Rossi, and M. K. Sein, "Being proactive: Where action research meets design research," D. E. Avison and D. F. Galletta, Eds., 2005. [Online]. Available: `http://aisel.aisnet.org/icis2005/27`.

[14] R. van Rozen, Y. Reijne, C. Julia, and G. Samaritaki, "First-Person Real-Time Collaborative Meta-Programming Adventures," 2021, To appear.

[15] L. Ermi and F. Mäyrä, "Fundamental components of the gameplay experience: Analysing immersion," 2005. [Online]. Available: `http://www.digra.org/digital-library/publications/fundamental-components-of-the-gameplay-experience-analysing-immersion/`.

[16] C. Crawford, "The art of computer game design," 1984.

[17] T. Fullerton, "Game design workshop. a playcentric approach to creating innovative games," Jan. 2008. DOI: `10.1201/b22309`.

[18] Y. Zhao *et al.*, “Winning isn’t everything: Training human-like agents for playtesting and game ai.,” *CoRR*, vol. abs/1903.10545, 2019. [Online]. Available: `http://arxiv.org/abs/1903.10545`.

[19] J. O. Choi, J. Forlizzi, M. G. Christel, R. Moeller, M. Bates, and J. Hammer, “Playtesting with a purpose,” A. L. Cox, Z. O. Toups, R. L. Mandryk, P. A. Cairns, V. V. Abeele, and D. M. Johnson, Eds., pp. 254–265, 2016. [Online]. Available: `http://dl.acm.org/citation.cfm?id=2968103`.

[20] J. Schell, “The art of game design: A book of lenses,” 2008.

[21] J. Togelius, E. Kastbjerg, D. Schedl, and G. N. Yannakakis, “What is procedural content generation? mario on the borderline,” PCGames ’11, 2011. DOI: `10.1145/2000919.2000922`. [Online]. Available: `https://doi.org/10.1145/2000919.2000922`.

[22] G. Smith, “Understanding procedural content generation: A design-centric analysis of the role of PCG in games,” M. Jones, P. A. Palanque, A. Schmidt, and T. Grossman, Eds., pp. 917–926, 2014. DOI: `10.1145/2556288.2557341`. [Online]. Available: `https://doi.org/10.1145/2556288.2557341`.

[23] N. Shaker, J. Togelius, and M. J. Nelson, “Procedural content generation in games,” Computational Synthesis and Creative Systems, 2016. DOI: `10.1007/978-3-319-42716-4`. [Online]. Available: `https://doi.org/10.1007/978-3-319-42716-4`.

[24] M. Vermeulen, “Automated Game Generation met Gebruik van Meta-Programming,” Aug. 2018.

[25] J. C. Osborn, B. Samuel, M. Mateas, and N. Wardrip-Fruin, “Playspecs: Regular expressions for game play traces,” A. Jhala and N. Sturtevant, Eds., pp. 170–176, 2015. [Online]. Available: `http://www.aaai.org/ocs/index.php/AIIDE/AIIDE15/paper/view/11541`.

[26] A. Khalifa, D. P. Liebana, S. M. Lucas, and J. Togelius, “General video game level generation,” T. Friedrich, F. Neumann, and A. M. Sutton, Eds., pp. 253–259, 2016. DOI: `10.1145/2908812.2908920`. [Online]. Available: `https://doi.org/10.1145/2908812.2908920`.

[27] C. Lim and D. F. Harrell, “An approach to general videogame evaluation and automatic generation using a description language,” pp. 1–8, 2014. DOI: `10.1109/CIG.2014.6932896`. [Online]. Available: `https://doi.org/10.1109/CIG.2014.6932896`.

[28] A. Krishnan, “Level generator for laserverse using asp,” 2018.

[37] R. van Rozen and J. Dormans, “Adapting game mechanics with micro-machinations,” M. Mateas, T. Barnes, and I. Bogost, Eds., 2014. [Online]. Available: `http://www.fdg2014.org/papers/fdg2014%5C_paper%5C_34.pdf`.

[38] P. Klint, T. van der Storm, and J. J. Vinju, “EASY meta-programming with rascal,” Lecture Notes in Computer Science, vol. 6491, J. M. Fernandes, R. Lämmel, J. Visser, and J. Saraiva, Eds., pp. 222–289, 2009. DOI: `10.1007/978-3-642-18023-1\_6`. [Online]. Available: `https://doi.org/10.1007/978-3-642-18023-1%5C_6`.

[39] H. A. Müller, J. H. Jahnke, D. B. Smith, M. D. Storey, S. R. Tilley, and K. Wong, “Reverse engineering: A roadmap,” A. Finkelstein, Ed., pp. 47–60, 2000. DOI: `10.1145/336512.336526`. [Online]. Available: `https://doi.org/10.1145/336512.336526`.

[40] E. J. Chikofsky and J. H. C. II, “Reverse engineering and design recovery: A taxonomy,” *IEEE Softw.*, vol. 7, no. 1, pp. 13–17, 1990. DOI: `10.1109/52.43044`. [Online]. Available: `https://doi.org/10.1109/52.43044`.

[41] E. Eilam, “Reversing: Secrets of reverse engineering,” 2005.

[42] T. J. Biggerstaff, “Design recovery for maintenance and reuse,” *Computer*, vol. 22, no. 7, pp. 36–49, 1989. DOI: `10.1109/2.30731`. [Online]. Available: `https://doi.org/10.1109/2.30731`.

[43] A. Sarkar, “The impact of syntax colouring on program comprehension,” M. Coles and G. Ollis, Eds., p. 8, 2015. [Online]. Available: `http://ppig.org/library/paper/impact-syntax-colouring-program-comprehension`.

[45] L. Tan, “Code comment analysis for improving software quality,” C. Bird, T. Menzies, and T. Zimmermann, Eds., pp. 493–517, 2015. DOI: `10.1016/b978-0-12-411519-4.00017-3`. [Online]. Available: `https://doi.org/10.1016/b978-0-12-411519-4.00017-3`.

[46] F. de Mesentier Silva, S. Lee, J. Togelius, and A. Nealen, “Ai-based playtesting of contemporary board games,” S. Deterding, A. Canossa, C. Harteveld, J. Zhu, and M. Sicart, Eds., 13:1–13:10, 2017. DOI: `10.1145/3102071.3102105`. [Online]. Available: `https://doi.org/10.1145/3102071.3102105`.

[47] M. P. Eladhari, A. Sullivan, G. Smith, and J. McCoy, "Ai-based game design : Enabling new playable experiences," 2011.

[48] M. P. Eladhari and M. Mateas, "Semi-autonomous avatars in world of minds: A case study of ai-based game design," ACM International Conference Proceeding Series, vol. 352, M. Inakage and A. D. Cheok, Eds., pp. 201–208, 2008. DOI: `10.1145/1501750.1501798`. [Online]. Available: `https://doi.org/10.1145/1501750.1501798`.

[49] J. Dormans, "Adventures in level design: Generating missions and spaces for action adventure games," R. Bidarra, Ed., 1:1–1:8, 2010. DOI: `10.1145/1814256.1814257`. [Online]. Available: `https://doi.org/10.1145/1814256.1814257`.

[50] J. Dormans and S. Leijnen, "Combinatorial and exploratory creativity in procedural content generation," 2013.

[51] P. Klint and R. van Rozen, "Micro-machinations - A DSL for game economies," Lecture Notes in Computer Science, vol. 8225, M. Erwig, R. F. Paige, and E. V. Wyk, Eds., pp. 36–55, 2013. DOI: `10.1007/978-3-319-02654-1\_3`. [Online]. Available: `https://doi.org/10.1007/978-3-319-02654-1%5C_3`.

[52] R. van Rozen and J. Dormans, "Adapting game mechanics with micro-machinations," M. Mateas, T. Barnes, and I. Bogost, Eds., 2014. [Online]. Available: `http://www.fdg2014.org/papers/fdg2014%5C_paper%5C_34.pdf`.

[57] E. Braad, G. Zavcer, and A. Sandovar, "Processes and models for serious game design and development," Lecture Notes in Computer Science, vol. 9970, R. Dörner, S. Göbel, M. D. Kickmeier-Rust, M. Masuch, and K. A. Zweig, Eds., pp. 92–118, 2015. DOI: `10.1007/978-3-319-46152-6\_5`. [Online]. Available: `https://doi.org/10.1007/978-3-319-46152-6%5C_5`.

[58] A. R. Gregersen and B. N. Jørgensen, "Dynamic update of java applications - balancing change flexibility vs programming transparency," *J. Softw. Maintenance Res. Pract.*, vol. 21, no. 2, pp. 81–112, 2009. DOI: `10.1002/smr.406`. [Online]. Available: `https://doi.org/10.1002/smr.406`.

[59] A. Ampatzoglou and I. Stamelos, "Software engineering research for computer games: A systematic review," *Inf. Softw. Technol.*, vol. 52, no. 9, pp. 888–901, 2010. DOI: `10.1016/j.infsof.2010.05.004`. [Online]. Available: `https://doi.org/10.1016/j.infsof.2010.05.004`.

[60] U. Tikhonova, M. Manders, M. van den Brand, S. Andova, and T. Verhoeff, "Applying model transformation and event-b for specifying an industrial DSL," CEUR Workshop Proceedings, vol. 1069, F. Boulanger, M. Famelis, and D. Ratiu, Eds., pp. 41–50, 2013. [Online]. Available: `http://ceur-ws.org/Vol-1069/07-paper.pdf`.

[61] A. van Deursen and A. Financial, "Domain-specific languages versus object-oriented frameworks: A financial engineering case study," 1997.

[62] R. T. A. Wood, M. D. Griffiths, D. Chappell, and M. N. O. Davies, "The structural characteristics of video games: A psycho-structural analysis," *Cyberpsychology Behav. Soc. Netw.*, vol. 7, no. 1, pp. 1–10, 2004. DOI: `10.1089/109493104322820057`. [Online]. Available: `https://doi.org/10.1089/109493104322820057`.

[63] A. Khalifa and M. Fayek, "Automatic puzzle level generation: A general approach using a description language," 2015.

[64] M. D. Storey, F. D. Fracchia, and H. A. Müller, "Cognitive design elements to support the construction of a mental model during software exploration," *J. Syst. Softw.*, vol. 44, no. 3, pp. 171–185, 1999. DOI: `10.1016/S0164-1212(98)10055-9`. [Online]. Available: `https://doi.org/10.1016/S0164-1212(98)10055-9`.

[65] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "A study of the documentation essential to software maintenance," S. R. Tilley and R. M. Newman, Eds., pp. 68–75, 2005. DOI: `10.1145/1085313.1085331`. [Online]. Available: `https://doi.org/10.1145/1085313.1085331`.

[66] R. van Rozen and Q. Heijn, "Measuring quality of grammars for procedural level generation," S. Dahlskog *et al.*, Eds., 56:1–56:8, 2018. DOI: `10.1145/3235765.3235821`. [Online]. Available: `https://doi.org/10.1145/3235765.3235821`.

[67] Z. Obrenovic, "Design-based research: What we learn when we engage in design of interactive systems," *Interactions*, vol. 18, no. 5, pp. 56–59, 2011. DOI: `10.1145/2008176.2008189`. [Online]. Available: `https://doi.org/10.1145/2008176.2008189`.

[68] D. S. Janzen and H. Saiedian, “Test-driven development: Concepts, taxonomy, and future direction,” *Computer*, vol. 38, no. 9, pp. 43–50, 2005. DOI: `10.1109/MC.2005.314`. [Online]. Available: `https://doi.org/10.1109/MC.2005.314`.

[69] N. Naus and J. Jeuring, “Building a generic feedback system for rule-based problems,” Lecture Notes in Computer Science, vol. 10447, D. V. Horn and J. Hughes, Eds., pp. 172–191, 2016. DOI: `10.1007/978-3-030-14805-8\_10`. [Online]. Available: `https://doi.org/10.1007/978-3-030-14805-8%5C_10`.

[70] A. Ensslin, “The language of gaming,” Dec. 2011.

[71] E. Dickmark, “The use of colour in the game journey: Case study,” 2015.

[72] M. Urbanek, P. Fikar, and F. Güldenpfennig, “About the sound of bananas - anti rules for audio game design,” J. L. Vilaça, T. Grechenig, D. Duque, N. F. Rodrigues, and N. Dias, Eds., pp. 1–7, 2018. DOI: `10.1109/SeGAH.2018.8401361`. [Online]. Available: `https://doi.org/10.1109/SeGAH.2018.8401361`.

[73] V. Sazawal and N. Sudan, “Modeling software evolution with game theory,” Lecture Notes in Computer Science, vol. 5543, Q. Wang, V. Garousi, R. J. Madachy, and D. Pfahl, Eds., pp. 354–365, 2009. DOI: `10.1007/978-3-642-01680-6\_32`. [Online]. Available: `https://doi.org/10.1007/978-3-642-01680-6%5C_32`.

[74] P. Järvinen, “Action research is similar to design science,” *Quality & Quantity*, vol. 41, no. 1, pp. 37–54, 2007.

[75] C. Fabricatore, “Gameplay and game mechanics: A key to quality in videogames,” 2007.

[76] P. Skalski, R. Tamborini, A. K. Shelton, M. Buncher, and P. Lindmark, “Mapping the road to fun: Natural video game controllers, presence, and game enjoyment,” *New Media Soc.*, vol. 13, no. 2, pp. 224–242, 2011. DOI: `10.1177/1461444810370949`. [Online]. Available: `https://doi.org/10.1177/1461444810370949`.

[77] L. Kruh, “Reverse engineering: Eilam, eldad. reversing: Secrets of reverse engineering. wiley publishing, inc., 10475 crosspoint boulevard, indianapolis in 46256 usa. 40.00US51.99 canada. 589 pp. 2005,” eng, *Cryptologia*, vol. 29, no. 3, pp. 282–283, 2005, ISSN: 0161-1194.

[78] G. Canfora, M. D. Penta, and L. Cerulo, “Achievements and challenges in software reverse engineering,” *Commun. ACM*, vol. 54, no. 4, pp. 142–151, 2011. DOI: `10.1145/1924421.1924451`. [Online]. Available: `https://doi.org/10.1145/1924421.1924451`.

[79] I. Pyle, “Software reuse and reverse engineering in practice,” *Computing & Control Engineering Journal*, vol. 4, Jan. 1993. DOI: `10.1049/cce:19930026`.

[80] M. Fowler, “Language workbenches: The killer-app for domain specific languages?,” 2005. [Online]. Available: `http://www.martinfowler.com/articles/languageWorkbench.html`.

[81] M. Treanor *et al.*, “Ai-based game design patterns,” J. P. Zagal, E. MacCallum-Stewart, and J. Togelius, Eds., 2015. [Online]. Available: `http://www.fdg2015.org/papers/fdg2015%5C_paper%5C_23.pdf`.

[82] M. Le, A. S. Sidhu, and N. S. Chaudhari, “Hierarchical pathfinding and ai-based learning approach in strategy game design,” *Int. J. Comput. Games Technol.*, vol. 2008, 873913:1–873913:11, 2008. DOI: `10.1155/2008/873913`. [Online]. Available: `https://doi.org/10.1155/2008/873913`.

Software

[29] ClementSparrow, *Pattern-script*, `https://github.com/ClementSparrow/Pattern-Script`, License: MIT, Commit: 7405d6ae1e7e1cc9d2ec63b1f78a693c6e2491b7, 2021.

[30] Joe Osborn (JoeOsborn), *Puzzlescript*, `https://github.com/JoeOsborn/PuzzleScript`, License: MIT, Commit: c725c60dbb083b278ed55d3b20a801b9320d880b, 2014.

[31] Rikki Prince (rikkiprince), *Puzzlescript-tracery*, `https://github.com/rikkiprince/PuzzleScript-Tracery`, License: MIT, Commit: 53ab1a5a532c4da2bf9d104b9da9fc5322abf277, 2019.

[33] Philip Schatz (philschatz), *Puzzlescript-rust*, `https://github.com/philschatz/puzzlescript-rust`, License: All Rights Reserved, Commit: 3691f45210347904487082df1dbb316d8b67538c, 2019.

[34] KerDelos, *Psionic*, `https://github.com/KerDelos/Psionic`, License: MIT, Commit: ccc3f2c6d82918c8375032b166ab4fcb50e289fe, 2020.

[35] Joseph Mansfield (sftrabbit), *Puzzlescript-smoothscreen*, `https://github.com/sftrabbit/PuzzleScript-smoothscreen`, License: MIT, Commit: 7257c2887f59402a368624d4c2bf78a83c1a61de, 2020.

[36] minotalen, *Puzzlescript-irreversible*, `https://github.com/minotalen/PuzzleScript-irreversible`, License: MIT, Commit: 1803c78df5747e5ea84a3ed4961b0b74e436ef87, 2019.

[44] randomon, *Timothy adventures 0.07*, `https://github.com/randomon/gameDesign`, License: MIT, Commit: b08269ab303ec2438616571fd7947f31c11856c5, 2018.

[53] vexorian, *Puzzlescript-bfs*, `https://github.com/vexorian/puzzlescript-bfs`, License: zlib, Commit: 5d47a2445238646f66fbb1abb95d9fb73bac5d4e, 2021.

[54] pancelor, *Puzzlescript*, `https://github.com/pancelor/PuzzleScript`, License: MIT, Commit: 99fecc0f7e0cf9b234bf01985f7bd895a932f8ca, 2020.

[55] Lauren Cunningham (Moofkin), *Puzzlescript-solver*, `https://github.com/Moofkin/PuzzleScript-Solver`, License: CC BY-NC-SA 3.0, Commit: 2a4fcd7faebf800a424a24d13d9befb41ba049f7, 2015.

[56] marcosdon, *Puzzlescriptwithsolver*, `https://github.com/marcosdon/PuzzleScriptWithSolver`, License: MIT, Commit: 14f47a8d21efe7281ecc09112afc20334a524574, 2018.

[83] Stephen Lavelle (increpare), *Puzzlescript*, `https://github.com/increpare/PuzzleScript`, License: MIT, Commit: f3655f2067540a143e99e06c62607b0d2f1e5964, 2013.

[84] Tijs van der Storm (tvdstorm), *Salix*, `https://github.com/usethesource/salix`, License: BSD-2-Clause License, Commit: 2ec316fca5d5d8a44dcbacc2138e4f9eaff99b52, 2017.

Chapter 9

Appendix

9.1 Appendix A

This is a collection of errors that the JavaScript implementation of PuzzleScript raises. These were used in our project to reverse-engineering PuzzleScript's design. The messages that appear below are ripped straight from the JavaScript code. We have, however, removed duplicate messages. We wanted to centralize the messages for future works and to have an easy reference to them.

9.1.1 Errors

1. "'random' cannot be matched on the left-hand side, it can only appear on the right", rule.lineNumber
2. "A rule has to have an arrow in it. There's no arrow here! Consider reading up about rules - you're clearly doing something weird", lineNumber
3. "An ellipsis on the left must be matched by one in the corresponding place on the right.", rule.lineNumber
4. "An ellipsis on the right must be matched by one in the corresponding place on the left.", rule.lineNumber
5. "Background must be in a layer by itself.",state.lineNumber
6. "Cannot define a property (using 'or') in terms of aggregates (something that uses 'and').", state.lineNumber
7. "Cannot define an aggregate (using 'and') in terms of properties (something that uses 'or').", state.lineNumber
8. "Commands should only appear at the end of rules, not in or before the pattern-detection/-replacement sections.", lineNumber
9. "Error, an
10. can only have one direction/action at a time, but you're looking for several at once!", lineNumber
11. "Expecting sfx data, instead found "" + seed + "".", lineNumber
12. "Got caught looping lots in a rule group :O",ruleGroup[0].lineNumber,true
13. "Movements cannot appear in late rules.", lineNumber
14. "Need have to have matching number of 'startLoop' and 'endLoop' loop points."
15. "No collision layers defined. All objects need to be in collision layers."
16. "Oops! " + object_name.toUpperCase() + " not assigned to a layer.", rule.lineNumber
17. "Something bad's happening in the LEGEND", state.lineNumber
18. "The start of a rule must consist of some number of directions (possibly 0), before the first bracket, specifying in what directions to look (with no direction specified, it applies in all four directions). It seems you've just entered "" + token.toUpperCase() + "'.', lineNumber
19. "There can't be more than 30 rigid groups (rule groups containing rigid members).", rules[0][0][3]
20. "There's no point in putting an ellipsis at the very start or the end of a rule", rule.lineNumber
21. "Trying to access color number "+n+" from the color palette of sprite " +state.objects_candname.toUpperCase()+", but there are only "+o.colors.length+" defined in it.",state.lineNumber
22. "You can't define object " + candname.toUpperCase() + " in terms of itself!", state.lineNumber
23. "You can't have anything in with an ellipsis. Sorry.", rule.lineNumber

24. ”You can’t use two ellipses in a single cell match pattern. If you really want to, please implement it yourself and send me a patch :) ”, oldrule.lineNumber
25. ”You cannot use ’no’ to exclude the aggregate object ” + c.toUpperCase() + ” (defined using ’AND’), only regular objects, or properties (objects defined using ’OR’). If you want to do this, you’ll have to write it out yourself the long way.”, lineNumber
26. ”You have an totally empty pattern on the left-hand side. This will match *everything*. You certainly don’t want this.”
27. ”a sprite cannot have more than 10 colors. Why you would want more than 10 is beyond me.”, o.lineNumber + 1
28. ”background cannot be an aggregate (declared with ’and’), it has to be a simple type, or property (declared in terms of others using ’or’).”
29. ”error detected - unexpected character ” + stream.peek(),state.lineNumber
30. ”error, didn’t find any object called player, either in the objects section, or the legends section. there must be a player!”
31. ”got caught in an endless startloop...endloop vortex, escaping!”, ruleGroup[0].lineNumber,true
32. ”have to have matching number of ’startLoop’ and ’endLoop’ loop points.”
33. ”no layers found.”,state.lineNumber
34. ”spooky ellipsis found! (should never hit this)”
35. ”too much stuff to define a sound event.”, lineNumber
36. ”you have to define something to be the background”
37. ”you have to have a background layer”
38. ”’’+n+’” is an aggregate (defined using ”and”), and cannot be added to a single layer because its constituent objects must be able to coexist.’, state.lineNumber
39. ’Cannot add ”’ + candname.toUpperCase() + ’” to a collision layer; it has not been declared.’, state.lineNumber
40. ’Cannot end a rule with ellipses.’, lineNumber
41. ’Cannot reference ”’ + candname.toUpperCase() + ’” in the LEGEND section; it has not been defined yet.’, state.lineNumber
42. ’Encountered an unexpected ”-¿” inside square brackets. It’s used to separate states, it has no place inside them ¿:— .’, lineNumber
43. ’Error in win condition: ”’ + candword.toUpperCase() + ’” is not a valid object name.’, state.lineNumber
44. ’Error, malformed cell rule - encountered a ”[”” before previous bracket was closed’, lineNumber
45. ’Error, malformed cell rule - was looking for cell contents, but found ”’ + token + ’”. What am I supposed to do with this, eh, please tell me that.’, lineNumber
46. ’Error, symbol ”’ + ch + ’” is defined using ’or’, and therefore ambiguous - it cannot be used in a map. Did you mean to define it in terms of ’and’?’, level[0] + j
47. ’Error, symbol ”’ + ch + ’”, used in map, not found.’, level[0] + j
48. ’Error, when specifying a rule, the number of matches (square bracketed bits) on the left hand side of the arrow must equal the number on the right’, lineNumber
49. ’Error, you can only use ”-¿” once in a rule; it’s used to separate before and after states.’, lineNumber
50. ’Expecting the word ”ON” but got ”’+candword.toUpperCase()+”’.”, state.lineNumber
51. ’Identifiers cannot appear outside of square brackets in rules, only directions can.’, state.lineNumber
52. ’In a rule, each pattern to match on the left must have a corresponding pattern on the right of equal length (number of cells).’, lineNumber
53. ’In a rule, if you specify a force, it has to act on an object.’, lineNumber
54. ’Invalid character ”’ + ch + ’” in sprite for ’ + state.objects_candname.toUpperCase(), state.lineNumber
55. ’Invalid color specified for object ”’ + n + ’”, namely ”’ + o.colors[i] + ’”.’, o.lineNumber + 1
56. ’Key ”’ + ch.toUpperCase() + ’” not found. Do you need to add it to the legend, or define a new object?’, state.lineNumber
57. ’MetaData ”’+token+’” has no parameters.’,state.lineNumber
58. ’MetaData ”’+token+’” needs a value.’,state.lineNumber
59. ’Name ”’ + candname.toUpperCase() + ’” already in use.’, state.lineNumber
60. ’Name ”’ + m + ’”, referred to in a rule, does not exist.’, state.lineNumber
61. ’No levels found. Add some levels!’, undefined, true

62. 'Object "' + candname.toUpperCase() + '" defined multiple times.', state.lineNumber
63. 'Object "' + n.toUpperCase() + '" has been defined, but not assigned to a layer.', dat.lineNumber
64. 'Object "' + target + '" not found.', lineNumber
65. 'Object "'+n+'" has been declared to be multiple different things',state.legend_aggregates[i].lineNumber
66. 'Object not found with name ' + n, state.lineNumber
67. 'Palette "' + val + '" not found, defaulting to arnecolors.', 0
68. 'Rule matches object types that can't overlap: "' + object_name.toUpperCase() + '" and "' + existingname.toUpperCase() + '".', rule.lineNumber
69. 'Spooky error! Empty line passed to rule function.', lineNumber
70. 'Sprites must be 5 wide and 5 high.', state.lineNumber
71. 'The "+" symbol, for joining a rule with the group of the previous rule, must be the first symbol on the line '
72. 'The "+" symbol, for joining a rule with the group of the previous rule, needs a previous rule to be applied to.'
73. 'This rule has a property on the right-hand side, "' + rhsPropertyRemains.toUpperCase() + "", that can't be inferred from the left-hand side. (either for every property on the right there has to be a corresponding one on the left in the same cell, OR, if there's a single occurrence of a particular property name on the left, all properties of the same name on the right are assumed to be the same).", lineNumber
74. 'This rule has an ambiguous movement on the right-hand side, "' + rhsAmbiguousMovementsRemain + "", that can't be inferred from the left-hand side. (either for every ambiguous movement associated to an entity on the right there has to be a corresponding one on the left attached to the same entity, OR, if there's a single occurrence of a particular ambiguous movement on the left, all properties of the same movement attached to the same object on the right are assumed to be the same (or something like that)).", lineNumber
75. 'This rule has some content of the form "X no X" which can never match and so the rule is getting removed during compilation.', rule.lineNumber
76. 'This rule refers to nothing. What the heck? :O', lineNumber
77. 'Trying to define aggregate "' + n.toUpperCase() + '" in terms of property "' + value.toUpperCase() + '".'
78. 'Trying to define property "' + n.toUpperCase() + '" in terms of aggregate "' + value.toUpperCase() + '".'
79. 'Two "+"s ("append to previous rule group" symbol) applied to the same rule.', lineNumber
80. 'Unknown junk in spritematrix for object ' + state.objects_candname.toUpperCase() + '.', state.lineNumber
81. 'Unrecognised stuff in the prelude.', state.lineNumber
82. 'Unwelcome term "' + n1 + '" found in win condition. Win conditions objects have to be objects or properties (defined using "or", in terms of other properties)', lineNumber
83. 'Was expecting a direction, instead found "' + direction + '".', lineNumber
84. 'Was looking for color for object ' + state.objects_candname.toUpperCase() + ', got "' + str + '" instead.', state.lineNumber
85. 'You cannot use relative directions ("v¡¿") to indicate in which direction(s) a rule applies. Use absolute directions indicators (Up, Down, Left, Right, Horizontal, or Vertical, for instance), or, if you want the rule to apply in all four directions, do not specify directions', lineNumber
86. 'You need to have some objects defined'
87. 'You want to spawn a random "' + object_name.toUpperCase() + '", but I don't know how to do that', rule.lineNumber
88. 'cannot assign sound events to aggregate objects (declared with "and"), only to regular objects, or properties, things defined in terms of "or" ("' + target + '").', lineNumber
89. 'cannot duplicate sections (you tried to duplicate "' + state.section.toUpperCase() + '").', state.lineNumber
90. 'color not specified for object "' + n + '".', o.lineNumber
91. 'incorrect format of legend - should be one of A = B, A = B or C (or D ...), A = B and C (and D ...)', state.lineNumber
92. 'incorrect format of win condition.', state.lineNumber
93. 'incorrect sound declaration.', lineNumber
94. 'must start with section "OBJECTS"', state.lineNumber

95. 'names cannot end with the letter "v", because it's is used as a direction.', state.lineNumber
96. 'no such section as "' + state.section.toUpperCase() + '".', state.lineNumber
97. 'section "' + state.section.toUpperCase() + '" is out of order, must follow "' + sectionNames[sectionIndex - 1].toUpperCase() + '" (or it could be that the section "'+sectionNames[sectionIndex - 1].toUpperCase()+'"is just missing totally. You have to include all section headings, even if the section itself is empty).', state.lineNumber
98. 'section "' + state.section.toUpperCase() + '" must be the first section', state.lineNumber
99. 'unexpected sound token "'+candname+'".' , state.lineNumber
100. example[0] + ' and ' + example[1] + ' can never overlap, but this rule requires that to happen.', rule.lineNumber

9.1.2 Warnings

1. "Author list too long to fit on screen, truncating to three lines.",undefined,true
2. "Commands should only appear at the end of rules, not in or before the pattern-detection/-replacement sections.", lineNumber
3. "Game title is too long to fit on screen, truncating to five lines.",undefined,true
4. "Invalid syntax, ellipses should only be used within cells (square brackets).", lineNumber
5. "Invalid syntax. Directions should be placed at the start of a rule.", lineNumber
6. "Invalid token " + token.toUpperCase() + ". Object names should only be used within cells (square brackets).", lineNumber
7. "Maps must be rectangular, yo (In a level, the length of each row must be the same).",state.lineNumber
8. "Multiple closing brackets without corresponding opening brackets. Something fishy here. Every '[' has to be closed by a ']', and you can't nest them.", lineNumber
9. "Sprite graphics must be 5 wide and 5 high exactly.", o.lineNumber
10. "This rule may try to spawn a " + o1 + " with random, but also requires a " + o2 + " be here, which is on the same layer - they shouldn't be able to coexist!", rule.lineNumber
11. "You probably meant to put a space after 'message' innit. That's ok, I'll still interpret it as a message, but you probably want to put a space there.",state.lineNumber
12. "throttle_movement is designed for use in conjunction with realtime_interval. Using it in other situations makes games gross and unresponsive, broadly speaking. Please don't."
13. 'A "restart" command is being triggered in the "run_rules_on_level_start" section of level creation, which would cause an infinite loop if it was actually triggered, but it's being ignored, so it's not.'
14. 'Janky syntax. "—" should only be used inside cell rows (the square brackety bits).', lineNumber
15. 'Message too long to fit on screen.', level[2]
16. 'Unknown junk in object section (possibly: sprites have to be 5 pixels wide and 5 pixels high exactly. Or maybe: the main names for objects have to be words containing only the letters a-z0.9 - if you want to call them something like ",", do it in the legend section).',state.lineNumber
17. 'You named an object "' + candname.toUpperCase() + '", but this is a keyword. Don't do that!', state.lineNumber

9.2 Appendix B

A list of figures related to all the errors currently implemented within ScriptButler. Some of these errors are new and some are translations from the original PuzzleScript implementation.

Table 9.1: Invalid errors detected by the analyzer

Name	Description
invalid_name	Name does not respect the pattern rules defined for it
invalid_color	Color not found in the chosen palette
invalid_sound_seed	Sound seed is not a valid positive int
invalid_sound_verb	Sound verbs not recognized
invalid_sprite	Sprite is not 5*5
invalid_level_row	Level rows are not all equal in length
invalid_sound_length	Sound definition has too many/not enough verbs
invalid_condition_length	Condition definition has too many/not enough verbs
invalid_condition	Combination of condition verbs not recognized
invalid_condition_verb	Condition verbs not recognized
invalid_object_type	Mixed properties and aggregates in legend definition
invalid_prelude_key	Prelude keyword not recognized
invalid_prelude_value	Prelude value invalid for prelude keyword
invalid_rule_prefix	Verb not valid as rule prefix
invalid_rule_command	Verb is not valid as a rule command
invalid_sound	Verb is not a valid sound
invalid_ellipsis_placement	Ruleparts cannot start or end with ellipsis
invalid_ellipsis	Ellipsis must be alone in the section
invalid_rule_part_size	Rules must have the same number of parts on each sides
invalid_rule_content_size	Rule parts must have the same number of sections on each part
invalid_rule_keyword_amount	Each force in a section must be applied to exactly one object
invalid_rule_keyword_placement	Each force in a section must be applied to exactly one object
invalid_rule_ellipsis_size	Ellipsis must be present in both parts at the same place
invalid_rule_movement_late	Late rules cannot match for movements
invalid_rule_random	Random keywords cannot be used on the left side

Table 9.2: Undefined errors detected by the analyzer

Name	Description
undefined_reference	Reference used but never defined
undefined_object	Object used but never defined
undefined_sound_seed	Sound seed does not exist
undefined_sound_mask	Sound mask used but never defined
undefined_sound_objects	Object used but never defined
undefined_sound	Sound event used but never defined

Table 9.3: Existing errors detected by the analyzer

Name	Description
existing_object	Object has already been defined
existing_legend	Legend has already been defined
existing_section	Section has already been defined
existing_mask	Mask has already been defined for this sound
existing_sound_seed	Seed has already been defined for this sound
existing_sound_object	Object has already been defined for this sound
existing_prelude_key	Prelude keyword already defined

Table 9.4: Misc errors detected by the analyzer

Name	Description
mixed_legend	Legend has both 'or' and 'and'
mixed_legend	Property cannot use aggregates/Aggregates cannot use properties
unlayered_objects	Object defined but not added to any collision layers
ambiguous_pixel	Properties cannot be used in levels
reserved_keyword	Cannot use keyword there
self_reference	Legend references self
mask_not_directional	Direction keywords used but mask is not directional
impossible_condition_duplicates	Duplicate conditions found
impossible_condition_unstackable	Conditions require items that exist on the same layer to stack
missing_prelude_value	Prelude keyword requires a value

Table 9.5: Warnings detected by the analyzer

Name	Description
unused_colors	Color is defined for the object but never used in sprite
no_levels	No levels have been defined for the game
message_too_long	The message might be too long to display
existing_sound	Sound event already registered, registering a new one
existing_condition	Similar win condition already exists
existing_rule	Similar rule already exists
redundant_prelude_value	Prelude keyword does not require a value
multilayered_object	Objects appear in multiple layers
semantic_warning	Object that needs to be on top for victory condition present in inferior layer
win_keyword	Win keyword in the rule makes other commands and rule parts useless