Ludwig-Maximilians-Universität München
Institut für Informatik

FORMAL VERIFICATION OF INTERACTIVE VISUAL NOVELS

Analysing the Structure of Interactive Stories to Find Out What Makes Them Fun

Elisabeth Lempa
`elisabeth.lempa@posteo.net`

A thesis presented for the degree of
Master of Science

Supervisor: Prof. Dr. Gidon Ernst
Submission Date: 04.04.2024

ABSTRACT

What makes visual novels fun? In this work, we create metrics to operationalise the contribution of visual novel story structure to video game enjoyment in five different categories. We present a formal verification tool for the Ren'Py language, Ren'Py being one of the most popular visual novel engines. This formal verification tool allows to specify certain properties about a Ren'Py program, and verify whether they hold on every possible path through the game. We give formal specifications of the metrics we created. Finally, we evaluate the metrics' ability to distinguish visual novels with regards to the categories the metrics each were based on, and present additional applications of the tools we developed.

CONTENTS

ACKNOWLEDGEMENTS

A lot of authors write that their work would not have been possible without other people; I feel like nobody has ever meant it as much as I do right now.
First and foremost, I thank my supervisor Dr. Gidon Ernst for his unending patience, gentle persistence, and continuous guidance and support. Not only did he give me the chance to write a thesis about a topic that is so incredibly close to my heart, he also handled it with so much grace and kindness when I went through an entire character arc while I wrote it. I fear that I was not an easy student to supervise, and I think of myself as very lucky to have had a supervisor who was more than up for the task.
I also want to thank my wonderful friend Jonathan for his plentiful advice. He would have made an amazing computer scientist, but sadly, he instead decided to study things that are actually hard, like genders, education and occasionally law.
A lot of the formulas in this work were looked over by one of my oldest and dearest friends Dr. Lukas Miaskiwskyi (mit k), who left academia (and Germany) behind, but came to help every single time when I flashed the “finally” operator sign on the night sky over Utrecht. Lukas, the marvel is in your court.
Dr. Niels Heller and his notes, always adhering to the strict Bry’ian school of writing, were also an essential part of making this thesis make any sense at all. Had my thesis been a cup of yoghurt, Niels would have put it inside a bag at the cash register.
I also want to thank Odin, my pair programming partner in crime, my relentlessly professional consultant, whom I forced to look at so much Maude documentation with me that I am positive that the rail switch software industry will soon pivot to using term rewriting logic. As they should!
I also want to thank my very last minute beta readers: My sister Katharina, and my friends Eva and Thomas – I am deeply lucky to have so many people in my life who care this deeply about precise language –, and my helpful LaTeX fairy Tobias. Every single one of you was crucial. All of you jumped in at such short notice. I know everyone says this, but I’m right: I have the best friends in the world. Thank you.
I would also like to thank the entire Ren’Py community for making and playing visual novel games. I’m glad to be part of such a diverse, vibrant, welcoming community. Literally everyone who was ever written a single line of Ren’Py: I see you, and you’re awesome.
I thank Ann Leckie for writing the Imperial Radch series, which is a series of books I lifted the names “Breq”, “Seivarden” and “Zeiat” from, and which has done more for my mental health than should ever be expected from a trilogy.
And finally, thanks Jan for supporting me through this. Thanks for suggesting I quit all of this and go pursue my other, even sillier dreams instead. I’m glad I didn’t listen. I love you.

SELBSTSTÄNDIGKEITSERKLÄRUNG

Hiermit versichere ich, dass die vorliegende Arbeit von mir selbstständig verfasst wurde und dass keine anderen als die angegebenen Quellen und Hilfsmittel benutzt wurden.

Diese Erklärung erstreckt sich auch auf in der Arbeit enthaltene Graphiken, Zeichnungen, und bildliche Darstellungen.

Insbesondere wurden keinerlei Hilsmittel eingesetzt, die auf generativer AI basieren.

04.04.2024, [illegible]

Datum, Unterschrift

STATEMENT OF ORIGINALITY

I hereby confirm that I have written the accompanying thesis by myself, without contributions from any sources other than those cited in the text and acknowledgements.

This applies also to all graphics, drawings and images included in the thesis.

I did not use any tools based on generative AI.

04.04.2024, [illegible]

Date, Signature

1 INTRODUCTION

Visual Novels are a form of interactive media that blur the lines between literature and computer games. They can tell stories that are various degrees of interactive across all sorts of genres. With this thesis, we want to add to the conversation about what makes video games fun by creating and evaluating formalised operationalised metrics to measure the impact of a visual novel's narrative structure on the enjoyment that is had when playing the visual novel.

This thesis has five chapters, the first of which is an introduction, which you're reading right now.
Chapter 2 provides relevant background information on the nature of visual novels and the Ren'Py engine in Sections 2.1.1 and 2.1.2. It also gives brief a overview of the relevant literature on video game enjoyment in Section 2.1.3, and presents some related work in analysing Ren'Py code at the end of Section 2.1.2. Finally, it summarises the theoretical and mathematical concepts the formalisation of the metrics is based upon, and gives a brief introduction to the Maude rewriting language in Sections 2.2.1 and 2.2.2.
Chapter 3 describes our contribution, beginning with the operationalisation and formalisation of metrics from different video game enjoyment categories in Sections 3.1 and 3.2. It continues by presenting the tools we developed to measure the formalised metrics in Section 3.3, where the Ren'Py-to-Maude translation tool R'Ast is presented, and Section 3.4, where a Maude module specifying an interpreter for a Ren'Py language fragment is presented.
Chapter 4 describes the process of evaltuating the formalised criteria to see whether or not they were able to differentiate on a sample of visual novels. Section 4.1 describes the experimental setup, and Section 4.2 presents its result. Finally in Section 4.3, the results are discussed. We draw conclusions and describe their limitations, and describe additional applications of the metrics we described and the tools we presented. The final chapter, Chapter 5 is the conclusion. It also includes some pointers for future work based on this thesis.

2 BACKGROUND AND RELATED WORK

This chapter will describe the domain of this work (video games, visual novels, and the Ren'Py visual novel engine) and give an overview of related work on some empirical methods of describing video game enjoyment, as well as other tools analysing Ren'Py source code in sections 2.1. In sections 2.2.1 and 2.2.2, we give some background on the mathematical and theoretical foundations of the work: Model Checking, the temporal logic LTL, and the term rewriting system Maude.

2.1 Video Games and What Makes them Fun

This section will will briefly give an overview on the literature on measuring people's fun in video games. It will also describe what the genre of visual novels comprises, and how it differs from other video game genres. It will also provide a brief introduction to the Ren'Py visual novel engine.

2.1.1 Visual Novels

Visual novels are a form of digital interactive fiction. There is some debate about whether or not they really fit into the broader category of video games, as they can often lack certain features that some describe as essential, however, practically, they are sold on gaming platforms like Steam or itch.io, where they are categorised as games.
The genre relies a lot on presenting a narrative in the form of written text to the player. The main form of interaction the player conducts with the game being them simply advancing the text, and occasionally making a choice that can influence the story's progression. (Often, they get to pick the main Character's dialogue or action in certain situations.)
They are distinguished from text-based adventure games of old through the use of graphics, as the term "visual" suggests. Typically, character sprites are shown in front of a background, showcasing their emotion or expression. Important moments in the narrative are sometimes embellished with dedicated *CGs* ("Computer Graphics"), usually hand drawn images that take up the entire screen. They also typically include sound and music, sometimes going as far as full voice acting.
The visual novel genre has historically been very popular in Japan, which has been related to the popularity of Anime. The influence of anime as a medium on the genre of visual novels is undeniable, with many visual novels sharing showing obvious connections in theme and art style. [Cavallaro, 2009]
The visual novel market is still very much a small niche of the huge video game market, but has been growing in popularity in the last few years. [Geoffrey Bunting, 2023]
Because visual novels are often technically simple compared to other video games that attempt to simulate expansive 3d worlds and strive for photorealistic graphics, the production costs of visual novel games are rather low. This makes visual novel

development especially feasible for small teams and even solo developers. [Lebowitz and Klug, 2011a]

VISUAL NOVELS AS SERIOUS GAMES Specialised tools such as the Ren'Py language facilitate development of visual novels without in-depth programming knowledge. Because of this lower barrier of entry and lower production costs compared to other video game genres, visual novels are also highly represented within serious games and commonly used for e-learning applications.[Øygardslia et al., 2020] [Lochman, 2020] This also means that visual novels are commonly created by people who are not game designers or programmers.

INTERACTIVE STORIES As mentioned before, visual novels often allow the player to make choices that influence the progression of the story. Stories that are dependent in such a way on the recipient's collaboration are often called *interactive stories*. In the following, we will call our recipient "the player", but we note that interactive stories can also be present in other forms of media, such as literature and film.

[Lebowitz and Klug, 2011a] further differentiate between *interactive traditional stories* (largely linear stories with small parts that change based on player interaction), *multiple-ending stories*, (stories that are largely linear but have multiple different endings) and *branching path stories* (stories that diverge into branching paths earlier). Visual novels can be any of those.

PLAYTHROUGHS AND ENDINGS Because visual novels commonly have multiple distinct paths and often multiple endings, they can be played multiple times with the expectation to see some new content each time. One of these runs through the game from start to finish is often called a "playthrough".

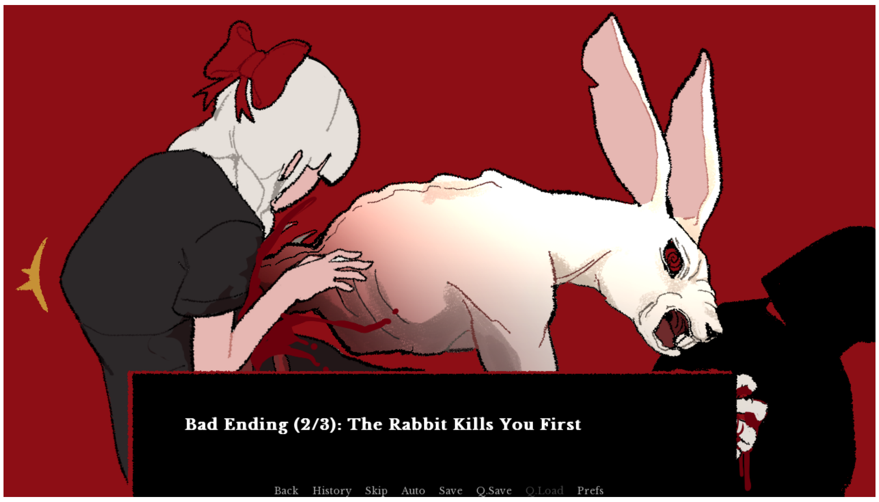
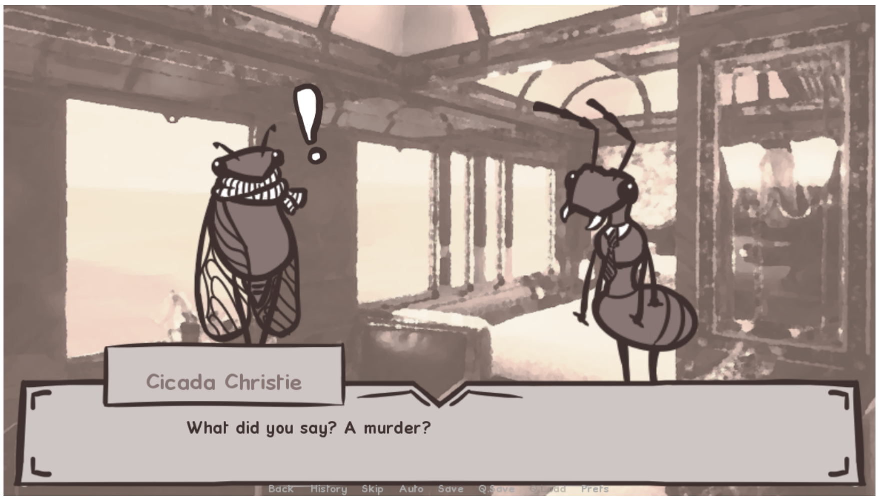

Bad Ending (2/3): The Rabbit Kills You First

Figure 1: Screenshot of the Visual Novel *Kill The Rabbit* [CODI/CLAUDIO, 2024]. The player is explicitly told that they have reached a bad ending.

A lot of interactive stories feature *bad endings* that are explicitly announced to the player. Furthermore, a lot of visual novels have dark or serious subject matters, and there is significant overlap with the horror genre. Popular examples of horror visual novels are *Doki Doki Literature Club* [2017] and *Slay the Princess* [2023].

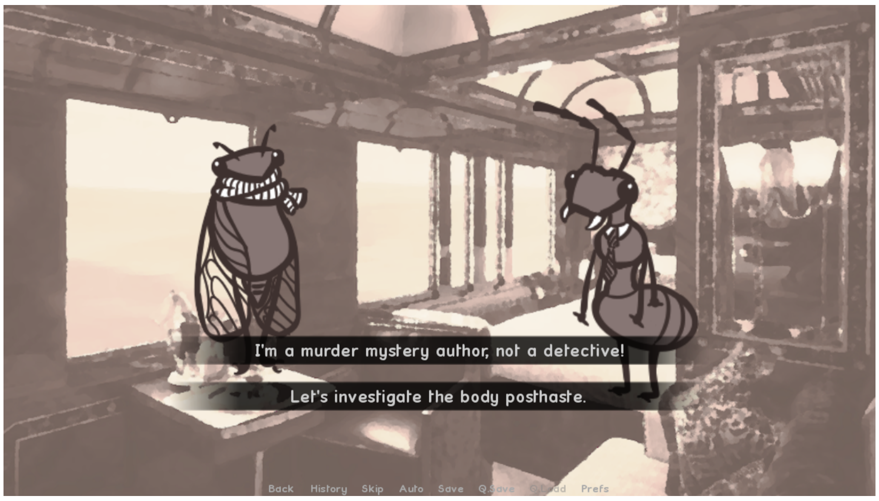
2.1.2 The Ren'Py Engine



What did you say? A murder?

Figure 2: A screenshot from the visual novel *Murder on the Insect Express*.

Ren'Py is a visual novel engine used by visual novel developers to create visual novel games by writing code in the Ren'Py scripting language. It has a fairly low barrier of entry due to a fairly straightforward screenplay-like style that is easy to learn by non-programmers. It also ships with some tools for no-code development such as the interactive director.[1] However, it is built on Python and allows advanced developers to fully utilise basically all features of the Python language, making it a tool suitable for both advanced programmers and non-programmers alike.



I'm a murder mystery author, not a detective!

Let's investigate the body posthaste.

Figure 3: A screenshot from the visual novel *Murder on the Insect Express*, showing an interactive menu offering the player two choices.

1 https://www.renpy.org/doc/html/director.html

REN'PY GAMES When playing a Ren'Py game, the basic user experience consists of reading lines of text being shown one by one on the screen. Typically graphics depicting the scene and characters will be shown, as well. The user can advance to the next line by clicking on the screen or pressing a button.

At certain points in the story, the user will typically be allowed to select one of multiple dialogue options by selecting them from an in-game menu. An example for such an in-game menu can be seen in Figure 3.

TOOLS FOR ANALYSING REN'PY CODE *Ren'Py Graph Viz* [Quimerc'h, 2022] is a tool that can be used to create graphical visualisations of the different branching paths a Ren'Py story can take. However, it does not account for (among other things) Python variables being set, making it incapable of analysing stories that use such variables in the story design. A brief guide on how to use Python Profilers[2] to analyse Ren'Py games was published on the Ren'Py forums [DizzyKa, 2017]. This is useful mainly for performance analysis of python programs in general and does not come with any utility for specific analysis of Ren'Py games with regards to visual novel domain-specific concerns like story strucure.

2.1.3 What makes games fun?

While there is little debate about the fact that enjoyment is an important reason why people play video games, there is no broad consensus on what exactly makes a video game enjoyable. Video game reviews attempting to judge whether or not a game is good are often controversely discussed among gamers [Sharp, 2018], such as the infamous IGN review of the game *Pokémon Alpha Sapphire and Omega Ruby* [Plagge, 2014] where the game was slighted for containing "too much water", which sparked many discussions about that statement's true meaning and critical merit.

PLAYER TYPES Richard Bartle, creator of the early text based massively multiplayer online game *MUD1*, famously created a *taxonomy of players* [Bartle, 1996], categorising players into four distinct groups that describe the motivation that makes the game fun for them. There have been many more attempts to create such categories. [Hamari and Tuunanen, 2014]
When it comes to creating empirical measures for game enjoyment of visual novels, there are two problems with the categorisations mentioned above. Firstly, while the classification of players into the distinct groups is based on empirical research, the underlying model, i.e. the categories themselves are typically based on philosophical contemplation or rhetorical analysis, and not rooted in empirical science. Secondly, Bartle based his category on two axes for game enjoyment, one of which is the relationship to other players. This makes sense because he was specifically speaking about multiplayer games. This focus on the social elements of play is prevalent in a lot of the literature that attempts to find similar categorisations.

2 `https://docs.python.org/2/library/profile.html`

A COMPREHENSIVE MODEL [Schaffer and Fang, 2018] attempt to create a comprehensive list of sources of computer game enjoyment through a broad literature review, and used empirical methods to refine this list into 34 distinct categories.

Because this model presents itself as the most comprehensive of the ones reviewed while still being solidly grounded in empirical research, we chose these 34 categories as a basis for the following work.

2.2 Theoretical Background

We briefly explain temporal logic model checking as well as the term rewriting system Maude, which were used to specify and test the operationalised metrics we construct and formalise in Sections 3.1 and 3.2.

2.2.1 *Temporal Logic Model Checking*

Model checking is a method of formally verifying whether a system adheres to a certain specification. To achieve this, both the system and its specification are modelled in a formal language. In this work, we use a *Kripke structure* to represent the possible configurations of our system and changes thereof, and the Linear Temporal Logic language LTL to specify properties we are interested in verifying. The following explanations of Kripke structures and LTL-model checking was adapted from [Clarke et al., 2018]'s Handbook of Model Checking.

KRIPKE STRUCTURES Kripke structures are finite directed graphs whose vertices are labeled with sets of atomic propositions. The vertices of this graph represent "states", the edges represent "transitions" between such states. Formally, a Kripke structure over a set A of atomic propositions is a 3-tuple $K = (S, R, L)$ where S is a finite set of states, $R \subset S \times S$ is the (total) transition relation, and the labeling function $L : S \to 2^A$ maps each state to a set of atomic propositions, signifiyng that those are the propositions that hold true in that particular state.
We can see that the transition relation R shows us all possible state changes the system can possibly undergo. A specific behaviour of the system can therefore be described by a path through the graph, represented by a finite or infinite sequence $\pi = s_0, s_1, s_2, ...$ of states such that $(s_i, s_{i+1}) \in R$ for all $i \geq 0$. Because the transition relation is total, every finite path can be extended to an infinite path.

LINEAR TEMPORAL LOGIC Linear Temporal Logic is a modal temporal logic with modalities referring to points in time in the future. Assuming that $p \in A$ is an atomic proposition and φ_1, φ_2 are two LTL formulas, the set of all LTL-formulas can then be constructed recursively with the following definition:

$$\varphi := \mathit{true} \mid \mathit{false} \mid p \mid \neg\varphi_1 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \bigcirc \varphi_1 \mid \Diamond \varphi_1 \mid \Box \varphi_1 \mid \varphi_1 \mathcal{U} \varphi_2 \mid \varphi_1 \mathcal{W} \varphi_2,$$

where the operators $\wedge, \vee, \neg$ are known from propositional logic. The operators $\bigcirc$ ("next"), $\Diamond$ ("finally"), $\Box$ ("globally"), $\mathcal{U}$ ("until"), and $\mathcal{W}$ ("weak until") are temporal operators.

An LTL formula can now be satisfied by an infinite sequence of $\pi = s_0, s_1, s_2, \ldots$ of states associated with sets of atomic propositions by the labelling function L. Formally, the satisfaction relation $\models$ between such a path and an LTL formula can be defined as follows:

- $\pi \models p$ iff $p \in L(s_0)$
- $\pi \models \neg\varphi$ iff $\pi \not\models \varphi$
- $\pi \models \varphi \vee \psi$ iff $\pi \models \varphi$ or $\pi \models \psi$
- $\pi \models \varphi \wedge \psi$ iff $\pi \models \varphi$ and $\pi \models \psi$
- $\pi \models \bigcirc\varphi$ iff $s_1 \models \varphi$
- $\pi \models \Diamond\,\varphi$ iff there exists an $i \geq 0$ such that $s_i \models \varphi$
- $\pi \models \Box\,\varphi$ iff for all $i \geq 0$ it holds that $s_i \models \varphi$
- $\pi \models \varphi\,\mathcal{U}\,\psi$ iff there exists an $i \geq 0$ such that $s_i \models \varphi$
  and for all $0 \leq j < i$ it holds that $s_j \models \psi$
- $\pi \models \varphi\,\mathcal{W}\,\psi$ iff $\pi \models (\varphi\,\mathcal{U}\,\psi) \vee \Box\,\varphi$

The satisfaction relation between a state s from the state space S and an LTL formula φ is further defined as

$$s \models \varphi \text{ iff for every infinite path } \pi \text{ starting from } s \text{ we have } \pi \models \varphi.$$

And now finally, given a state s over a Kripke structure K and an LTL formula φ, an LTL-model-checking algorithm is a procedure that decides whether $s \models \varphi$. In the case that $s \not\models \varphi$, the model-checking algorithm will usually helpfully provide a counterexample in the shape of a path $\pi = s_0, s_1, \ldots$ of K such that $s = s_0$ and $\pi \not\models \varphi$.

2.2.2 Term Rewriting and Maude

Maude is a specification and programming language based on equational and rewriting logic. It was originally developed by a group of researchers at SRI International, and is currently being maintained and developed by a team of researchers from the US and Spain.[3]
In the follwing section, we provide an overview over the basic functionality of the term rewriting system, and the language features that were used in this thesis specifically. The following section does not in any way shape or form attempt to explain the entire Maude specification, which can be found in [Clavel et al., 2007].

3 https://maude.cs.illinois.edu/w/index.php/The_Maude_System:About

MAUDE MODULES AND SEMANTICS Maude differentiates between *system modules*, (declared with the key words `mod` and `endm`) and *functional modules*, declared with the keywords (`fmod` and `endfm`). System modules are essentially a list of declarations of

1. importation relationships to other modules,
2. sorts and subsorts (which are automatically grouped into equivalence classes called "kinds" by the Maude system),
3. operators,
4. equations and membership statements, and
5. rewriting rules.[4]

Functional modules meanwhile are not allowed to include any rewriting rules, meaning that they are essentially lists of declarations of (1)-(5). This, of course, makes them a special case of system modules. This relationship is meaningful because Maude supports both *membership equational logic* and *rewriting logic*, the former being contained in the latter as a sublogic. [Bruni and Meseguer, 2003] Because we used a system module to specify the main Maude theory for this work (elaborated upon in Section 3.4), we will only give a brief explanation of the mathematical semantics of Maude rewrite theories. This explanation mainly follows the one from Chapter 1.2 of [Clavel et al., 2020].
Mathematically, a Maude system module specifies a *rewrite theory* $\mathcal{R}$:

$$\mathcal{R} = (\Sigma, E \cup A, \varphi, R),$$

where Σ is the *signature* that specifies the type structure (sorts, kinds, operators, etc.), E is the collection of equations and memberships, and A is a collection of equational attributes (such as associativity, commutativity...) declared for the different operators, φ is the function specifying the frozen arguments[5] of each operator in Σ, and R is a collection of rewriting rules. From this theory, we can construct a transition system $\mathcal{T}_{\mathcal{R}}$. The state transitions are the *concurrent rewrites* possible in the system by application of the rules R.
In the following, an example Maude system module representing a deterministic finite automaton will be presented.

REWRITING RULES To make the DFA module model a DFA, we can conveniently describe its transition relation in the form of rewriting rules.
Mathematically, a rewrite rule has the form $l : t \to t'$, where t, t' are terms and l

4 Coming from a nice declarative language like Haskell, one could say that sort declarations are similar to data type declarations and operator definitions are similar to function type signatures. Rules and equations can then be thought of as fulfilling somewhat similar roles as function bodies, however they do it very differently and one should not rely on this comparison too heavily when trying to understand them. And yet, this comparison is one that I would have needed to be pointed out to me in order to understand Maude much quicker, so I feel compelled to give it.

5 This is explained in Section 4.4.9 of [Clavel et al., 2020] but omitted here because no arguments were frozen in the creation of this thesis.

```
mod DFA is
    sort Word .
    ops a b empty : -> Word .
    op _,_ : Word Word -> Word [assoc] .

    sort State .
    ops s0 s1 s2 : -> State .

    sort DFAState .
    op _|_ : State Word -> DFAState .
    op accepting : -> DFAState .
    op rejecting : -> DFAState .

    var X : Word .

    rl [t1] : s0 | a,X => s1 | X .
    rl [t2] : s0 | b,X => s2 | X .
    rl [t3] : s1 | a,X => s1 | X .
    rl [t4] : s1 | b,X => s2 | X .
    rl [t5] : s2 | a,X => s2 | X .
    rl [t6] : s2 | b,X => s2 | X .

    rl [f1] : s0 | empty => rejecting .
    rl [f2] : s1 | empty => rejecting .
    rl [f3] : s2 | empty => accepting .
endm
```

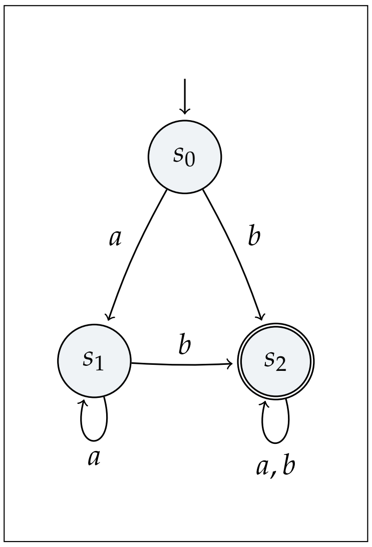


Figure 4: Declaration of a Maude module specifying the DFA pictured on the right. We define three sorts `Word`, `State`, and `DFAState`, several operators on these sorts, a variable `X` and the rewriting rules specifying the DFA's transition relation (t1-t6) as well as the DFA's final state and corresponding acceptance behaviour (f1-f3).

is the label of the rule. Every rule describes a transition in a system, meaning that anywhere in the system state where a substitution instance $\sigma(t)$ of the left hand side t is found, a local transition of that state fragment to the new local state $\sigma(t')$ can take place. In Maude, this is expressed by:

```
rl [<label>] :  <Term1> => <Term2> .
```

where `rl` is the keyword for rules, `label` is a label, and `Term1` and `Term2` are terms of the same kind.

To see our DFA model in action, we can use the `rewrite` command to simulate our DFA's run on the word *ababab* word by typing: `rewrite s0 | (a,b,a,b,a,b,empty)`.

If we look at the rules defined above, we see that rule `[t1]` is applicable, and that after that application, our new `DFAState` will be `s1 | (b,a,b,a,b,empty)`. Eventually we would get `s2 | empty`, which would be rewritten to `accepting` by rule `[f3]`. If we type the above query in Maude, we will indeed see:

```
Maude> rewrite s0 | (a,b,a,b,a,b,empty) .
rewrite in DFA : s0 | a,b,a,b,a,b,empty .
rewrites: 6 in 0ms cpu (0ms real) (~ rewrites/second)
result DFAState: accepting
```

This shows us that the DFA accepts the word *ababab*. The DFA, of course, lives up to its name: When reading a word, there is only ever one rule applicable at the same time. To showcase the true power of the term rewriting system, lets now define an NFA (Figure 5).

```
mod NFA is
    sort Word .
    ops a b empty : -> Word .
    op _,_ : Word Word -> Word [assoc] .

    sort State .
    ops s0 s1 s2 : -> State .

    sort NFAState .
    op _|_ : State Word -> NFAState .
    op accepting : -> NFAState .
    op rejecting : -> NFAState .

    var X : Word .

    rl [t1] : s0 | a,X => s0 | X .
    rl [t2] : s0 | b,X => s0 | X .
    rl [t3] : s0 | a,X => s1 | X .
    rl [t4] : s1 | b,X => s2 | X .

    rl [f1] : s0 | empty => accepting .
endm
```

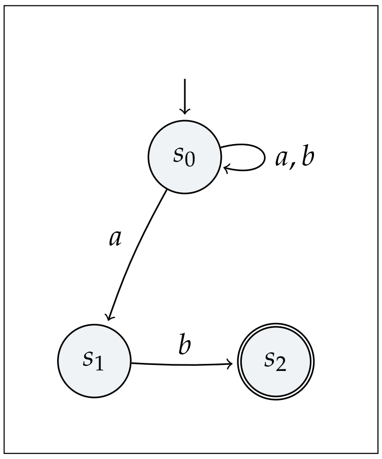


Figure 5: Declaration of a Maude module specifying the NFA pictured on the right.

We cannot simulate the NFA just by using the `rewrite` command. When using `rewrite`, one rule is picked to apply at a time using a specific strategy. However, to know whether or not an NFA accepts, we need to try out all possible transitions to learn about all possible final states a word could end up in. To do this in Maude, we can use the search command:

```
Maude> search in NFA : s0 | a,a,b,empty =>! S:NFAState .
search in NFA : s0 | a,a,b,empty =>! S:NFAState .
```

```
Solution 1 (state 2)
states: 5  rewrites: 4 in 0ms cpu (0ms real) (~ rewrites/second)
S:NFAState --> s1 | a,b,empty

Solution 2 (state 6)
states: 8  rewrites: 7 in 0ms cpu (0ms real) (~ rewrites/second)
S:NFAState --> s2 | empty

Solution 3 (state 7)
states: 8  rewrites: 7 in 0ms cpu (0ms real) (~ rewrites/second)
S:NFAState --> accepting

No more solutions.
states: 8  rewrites: 7 in 0ms cpu (0ms real) (~ rewrites/second)
```

The search command performs a breadth-first search for rewrite proofs starting at the term given at the left hand site of the arrow, looking for a final state that matches the term at the right side of the arrow. In the example above, we used the `=>!` search type, which yields only solutions that cannot be further rewritten. (Mainly to save space on the page.) From this result, we can see that the NFA accepts the word *aab*, because there is a path that ends in an accepting state.

3 CONTRIBUTION

We created several metrics to capture certain aspects of what makes visual novel games fun. Using the categories described by [Schaffer and Fang, 2018] as a starting point, we narrowed our focus on just those categories that apply to visual novels and relate specifically to the formal structure of the visual novel's story. Understanding each of these categories as a criteria for fun visual novels, we then tried to create specific, operationally defined metrics for each of those criteria, such that those metrics can be reasoned about formally by analysing the source code of the game. These metrics and their formalisation into temporal logic are described in 3.1.

Furthermore, we built the *R'Ast* tool to translate the Ren'Py source code into the term rewriting language Maude, which is described in the section 3.3. Finally we created a Maude theory *MiniRenRun*, that interprets the translated terms and can be used to reason about them using an LTL temporal logic model checker, which is described in 3.4.

These tools were then used to analyse Ren'Py code with regards to the criteria described in 3.1. The results of this analysis can be found in Chapter 4.

3.1 Operationalised Metrics for Game Design Principles

These metrics are described with regards to how they relate to the criteria they are intended to measure. Reasons for why they were chosen specifically were given.

We assume that the structure of the interactive story is one aspect of what makes visual novels fun. To investigate in what way this factor contributes, we refer to [Schaffer and Fang, 2018], where 34 different distinct categories of sources of computer game enjoyment are described. 7 of them were deemed not applicable to visual novels at all: 4 relate to interactions with other players (visual novels are almost exclusively single player games), 1 speaks about moving one's body (something that typically requires specific hardware to incorporate into a game), 2 describe some form of skill acquisition or progression (visual novels do not require any skills in the sense most other video games do).

Out of the remaining 27, we chose 5 that specifically relate to the structure of an interactive story. The following section explains these 5 criteria, how they apply to the structure of visual novels, and the metrics we developed to measure them in the visual novels we analysed.

It is important to note that there are many other factors that we would expect to contribute to how fun a visual novel is. [Schaffer and Fang, 2018] for instance also mention the categories *Presence, Role-Playing and Identification with the Character* and *Story*, that one would expect to be very important to visual novel enjoyment. However, this work only focuses on the formal structure of visual novels and how it contributes to visual novel enjoyment, and does not claim to fully capture all facets that make a visual novel enjoyable.

In the following section, each metric will be abbreviated with a symbol such as A_1^+. These symbols always include a little $+$ or $-$ sign denoting whether the metric has positive or negative polarity, i.e. whether this metric is thought to contribute to the enjoyment in this category $(+)$ or reduce it $(-)$.

ACHIEVEMENT AND COMPLETION This category is described in [Schaffer and Fang, 2018] as: *Finishing or completing a major task, and the feeling of closure and accomplishment that finishing the task gives you*. As described in 2.1.1, visual novels typically have multiple endings. Reaching one ending could reasonably be described as completing the visual novel. The metrics in this category describe whether or not an ending is reachable, (which one would argue is necessary to complete it) and how many "steps" in the story, which correspond to user inputs, are necessary to reach it, in relation to the total amount of positions. This second measure is intended to be an approximation of "replayability", i.e. how many times the player might be expected to be able to get the sense of completion from finishing it again after having finished it once by describing what percentage of the story was already seen on the first playthrough.

- A_1^+: Can an ending position be reached? (true/false)
- A_2^-: How long is the shortest path to an ending position? (Numerical, normalised by dividing by total number of positions.)

CONTROL, CHOICE, AUTONOMY In a visual novel, the player enacts control over how the story plays out by selecting specific choices, as described in 2.1.2. We capture the amount of such points where the player gets to make a choice with the metric C_1^+. However, because not all choices actually lead to different outcomes, we also want to measure how many different outcomes the story has. C_2^+ can work as an approximation for a lower boundary for that; It is assumed that two different positions in the source code that both end the game will correspond to somewhat different outcomes of the story. However, it is not a good assumption that one ending position can only correspond to one distinct outcome. C_3^+ in turn aims to capture the upper boundary for that, capturing how many different system states the game can be in while it is in an ending position, assuming that different story outcomes will always correspond with either different variable states, or a different ending position in the story.

- C_1^+: How many different in-game menus specified in the game code are reachable? (Numerical)
- C_2^+: How many different positions in the game that are endings can be reached? (Numerical)

- C_3^+: How many different states can the system be in at the end of a playthrough? (Numerical)

DANGER, UNCERTAIN OUTCOMES, SUSPENSE, SURPRISE, BRAVERY Danger in video games is always simulated, but whereas players of other genres stand to lose their in-game life or currency, in visual novels often the only bad outcomes are those where the story takes a turn for the worse (the main character dies, the love interest rejects them, the murderer gets away, ...). The metrics D_1^+ and D_2^+ in this category attempt to capture the prevalence of bad endings (as described in Section 2.1.1) and moments in the story that are meant to elicit fear, suspense or surprise. The metric D_3^+ aims to capture the uncertainty generated by scary moments in the story that are present, but that do not have to be visited in order to complete the story.

- D_1^+: How many of the different end states (C_3^+) are in a bad ending position? (Numerical, normalised by dividing by total number of ending states.)
- D_2^+: How many reachable positions are intended to shock, frighten or surprise the player? (Numerical)
- D_3^+: Can an ending position be reached **both with and without** traversing a position that is intended to shock, frighten or surprise the player? (True, False)

MAKING PROGRESS In order to quantify the sense of making progress a player might get from playing a visual novel, remembering the branching nature of interactive stories, we made the assumption that losing progress (i.e. going back to a position that was visited before on the same playthrough) is counterproductive to *making progress*. Therefore, we developed two metrics that relate to the possibility of going back to a position that was already passed before.

- P_1^+: How many positions are "Checkpoints", i.e. once you've passed them, you cannot go back? (Numerical, normalised by dividing by total number of reachable statements.)
- P_2^-: How many positions can be looped infinitely? (Numerical, normalised by dividing by total number of reachable statements.)

OPTIMAL VARIETY & NOVELTY [Schaffer and Fang, 2018] describe this category as: *"An amount of variety and novelty that is neither so low that it is boring nor so high that it is overwhelming"*, emphasizing that there is a range of variety that is best, while less and more variety would both be undesirable.

However, because we analyse visual novels with regards to their story structure, and because visual novels have some characteristics in common with works of literature that they don't share with other video games, we assume that in this case, more variety when it comes to the text being shown is always better. This also seems reasonable because other types of video games bring variety by introducing new, more complicated game systems to the player, something that visual novels typically lack.

Because of this, the metrics we chose are intended to give a sense of the amount of repetition in a visual novel, and thus all contribute negatively to the phenomenon of variety and novelty.

- V_1^-: How many positions can be visited more than once on a path to an ending position? (Numerical, normalised by dividing by total number of statements.)
- V_2^-: How many positions have to be visited at least twice on a path to an ending position? (Numerical, normalised by dividing by total number of statements.)
- V_3^-: What is the maximum number of times any one line needs to be visited in order to reach an ending position? (Numerical)

3.2 Formalisation of Metrics

To formalise the metrics we described in Section 3.1, we describe the state of a Ren'Py program at any point during the execution as a 3-tuple (F, I, p) consisting of

- the mapping $F : V_F \to \mathbb{B}$ representing the current interpretation of all boolean variables (called "flags"),
- the mapping $I : V_I \to \mathbb{Z}$ representing the current interpretation of all integer variables,
- and the program counter $p \in \mathbb{N}$, reflecting the current line number.

for two disjoint sets V_F and V_I of flags and integer variables respectively. We also define the initial state of the system as $s_0 = (F_0, I_0, 1)$, where F_0 sends all values to `false` and I_0 sends all values to 0. (We will find out in Section 3.3 why this is consistent with our other assumptions of the Ren'Py model.)

REACHABILITY Metrics A_1^+, C_1^+, C_2^+ and D_2^+ all relate to the reachability of certain positions. To show reachability of a line, we define the atomic property $\mathit{visting}(n)$ that is satisfied if and only if the system is currently visiting line n:

$$\mathit{visiting}(n) \in L(F, I, p) \text{ iff } n = p$$

Then, to get reachability of each line number n, we must ask:

$$s_0 \models \Box \neg \mathit{visiting}(n)$$

or "can line n never be visited", and then negate that result.[6]

CHECKPOINTS The metric P_1^+ is a stability property that can be expressed in the LTL formula:

$$s_0 \models (\neg \mathit{visiting}(n)) \, \mathcal{W} (\bigcirc \Box \mathit{after}(n))$$

where the atomic property $\mathit{after}(n)$ is defined as

$$\mathit{after}(n) \in L(F, I, p) \text{ iff } p > n$$

This means that once n was visited for the first time, $\bigcirc \Box \mathit{after}(n)$ must hold, i.e. from the next state onwards the program counter must always be greater than n.

6 Remember that that is not the same as simply asking $\neg \Box \neg \mathit{visiting}(n)$, which would be the same as asking $\neg\neg \Diamond \mathit{visiting}(n)$, or even $\Diamond \mathit{visiting}(n)$, because in LTL all formulas are implicitly universally quantified over all paths (see Section 2.2.1).

INFINITE LOOPS To measure P_2^-, we check for lines that **cannot** be looped infinitely by checking for each line n if

$$s_0 \vDash \neg \Box \Diamond \, \mathit{visiting}(n)$$

is true, and then negating the result. This is because the formula above expresses *unloopability*, i.e. it is not the case that n will finally be visited from all positions in the path.

CAN BE VISITED MORE THAN ONCE For the metric V_1^-, we ask for every line n:

$$s_0 \vDash (\neg \mathit{visiting}(n)) \, \mathcal{W} \, (\bigcirc \Box \neg \mathit{visiting}(n))$$

and then negate the result. This is because the above formula is satisfied if and only if line n can only be visited at most once, basically saying that n is not visited until, in the next state, n is never visited again.

MUST BE VISITED AT LEAST K TIMES V_2^-, which describes if a line must be visited twice, is measured by querying

$$s_0 \vDash \Diamond \, (\mathit{visiting}(n) \wedge \bigcirc \Diamond \, \mathit{visiting}(n))$$

because it encodes that, eventually, we will be visiting n while at that same moment looking forward to visiting n at a distinct future timestep. This can then be generalised to the following formula that describes if a line must be visited at least k times which is defined recursively as follows:

$$s_0 \vDash \Diamond \, (\mathit{visiting}(n) \wedge \bigcirc \Diamond \, F_{k-1})$$

Where F_{k-1} is the formula that describes that a line n must be visited at least $k-1$ times. Querying this formula iteratively for larger k can be used to compute V_3^-.

DISTINCT ENDING STATES Finally, to gain insight into all different ending states the system can be in, and whether or not they are *bad*, we do not rely on model checking LTL formulas. Instead we use the `search` command of the Maude system to list all states in which the system is in an ending position (we will see why ending positions correspond to `return` statements in Section 3.3):

```
search :
   initial =>*
   Fs:FlagState ; Is:IntState ; p(N:Nat)
   such that return := line(N:Nat) .
```

And then we filter the result based on whether or not the ending was tagged as *bad*. (See Section 4.1 for details on this tagging process.) Similarly, the shortest path to an ending A_2^- is found by iterating that same search command with increasing search depth until a solution is found. Because every search step corresponds to one Ren'Py statement being executed, it makes sense to use search depth to measure the length of the path through the Ren'Py program, as will become more apparent in Section 3.4. The specific Maude queries that were used to generate every one of the above results can be found in the appendix.

3.3 R'Ast: A Ren'Py to Maude Translator

R'Ast is a translation software that can parse Ren'Py code into an abstract syntax tree and generate corresponding maude terms. These maude terms can then be further reasoned about using the maude interpreter described in 3.4. The following section will give an overview over R'Ast's main components and their functionalities. As can be seen in Figure 6, R'Ast has three main components: The Preprocessor, the Ren'Py Parser, and the Maude Generator.

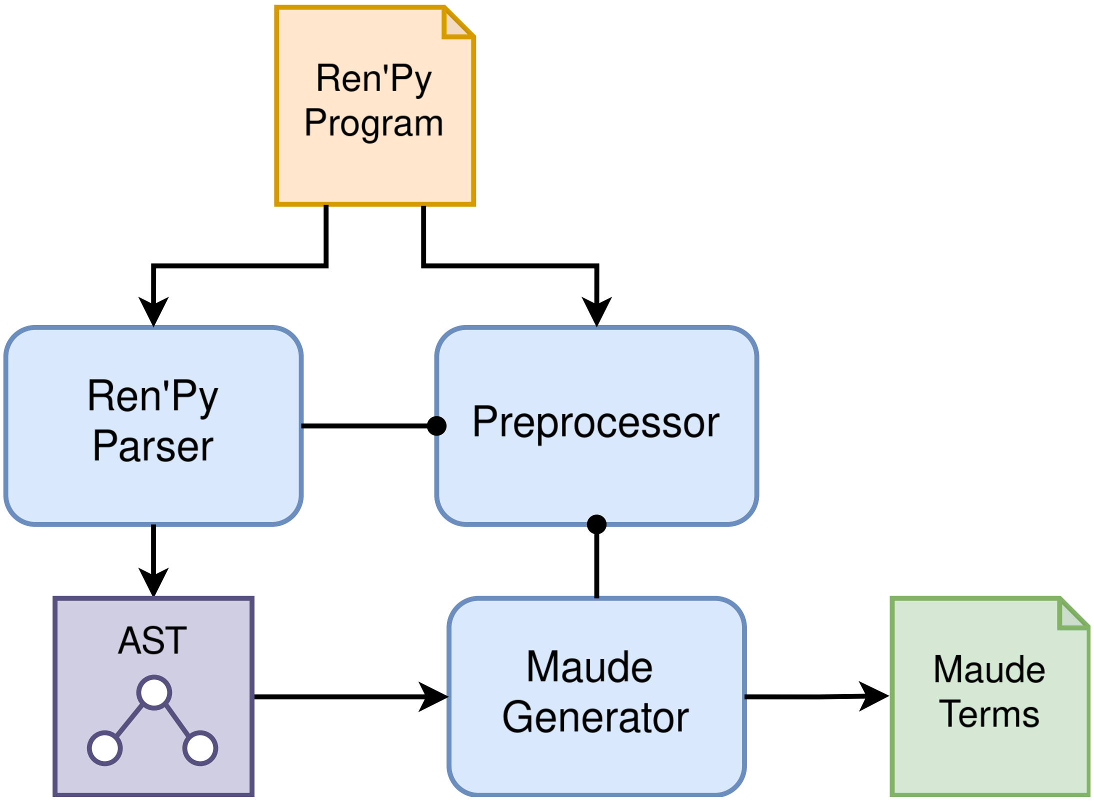


Figure 6: R'Ast main components overview. Arrows show the path of the inputs and outputs through the components. Edges with circles denote that a component holds a reference to another component; the referenced component is the one intersected by the circle. Imagine that the edge is an arm and the circle is a little hand holding onto the other component.

The Preprocessor receives Ren'Py code as input and provides certain utility functions to the other components. The Parser (to nobody's surprise) parses the Ren'Py code. A successful parse yields the Ren'Py code's representation as an abstract syntax tree (AST). The Maude Generator takes this AST as its input and generates the corresponding maude terms. The Parser and the Maude Generator each hold a reference to the Preprocessor.

REN'PY GRAMMAR AND SEMANTICS The Ren'Py language has a screenplay-like syntax that allows the visual novel developer to write each line of dialogue as a Ren'Py statement. The lines are shown to the user one by one, like described in 2.1.2.

```
define breq = Character("Breq from the Gerentate", color="#121212")
show breq waving_hand
breq "I'm Breq, and this is the first line of dialogue the user would see."
breq "And this is the second line of dialogue the user would see."
hide breq
```

Listing 7: A simple Ren'Py example. The character *Breq from the Gerentate* is defined and bound to the identifier `breq`, and their speaking text colour is set. A picture of this character is then shown, a line of text displayed, and the picture is removed from the screen.

A simple example illustrating this screenplay-like nature can be seen in Listing 7. Because of this basic structure, each line of text corresponds to a distinct line in the source code. In the following, these distinct lines will sometimes also be referred to as "positions" in the story.

For practical reasons, we restricted the Ren'Py grammar to a smaller subset in this work. In the following sections, this smaller fragment will sometimes be referred to as *MiniRen*. Theoretically, the Ren'Py language supports including blocks of Python code, and it would not be feasible to write a full Python parser/interpreter. Furthermore, a lot of the Ren'Py language concerns itself with the "visual" elements of visual novels (i.e. showing and transforming images), which were left out because this work focuses on analysing the narrative structure of visual novels. This leaves us with the simplified grammar for Ren'Py statements shown in Listing 8.[7]

```
<statement> = <say> | <menu> | <setFlag> | <label> | <jump> | <setInt>
<say>       =  <identifier> <String> | <String> <String> | <String>
<menu>      = menu: (<say>)? <choice>+
<choice>    = <string> <condition>?: <statement>+
<condition> = if <identifier>
<setFlag>   = $ <identifier> = (true | false)
<setInt>    = $ <identifier> '=' <intExp> | $ <identifier> '+=' <intExp>
<branch>    = if <condition>: <statement>* (else: <statement>*)?
<label>     = label <identifier>: <statement>*
<jump>      = jump <identifier>
<return>    = return
```

Listing 8: The simplified grammar for Ren'Py statements.

LINES OF TEXT The say-command represents a line of dialogue being displayed. The three alternatives shown in the production rule correspond to three different ways this can be achieved in Ren'Py: The character saying the line can be specified

7 This is an abstract representation that doesn't account for whitespace handling and excludes rules for the boolean and integer expressions, string literals and identifiers, which are as the benevolent reader familiar with statement- and expression-grammars would expect them to be.

by a previously defined identifier (an example of this can be seen in Listing 7), the character can be specified by simply giving their name as a string, or the character can be omitted entirely, usually indicating some sort of description or narration.

INTERACTIVE MENUS Interactive dialogue choices for the player to choose from are expressed in the form of in-game *menus*. A `menu` is comprised of an optional `say` statement that is displayed alongside the menu choices (typically this can be the question the player is supposed to select their character's answer to), followed by multiple *menu choices*.

```
 1 label seivarden_asks_for_tea:
 2 menu:
 3   seiv "Do we have any tea?"
 4   "Use the flask." if flask:
 5       seiv "It doesn't brew right."
 6       breq "Well, suit yourself."
 7   "Go and buy some, if you like.":
 8       seiv "Won't we be late?"
 9       breq "No."
10   "No, I don't think so.":
11       seiv "This is hardly civilised."
12 breq "Let's go to the shuttle."
```

Listing 9: A sample Ren'Py program showing off the **`menu`** statement. In this example, the first option *"Use the flask."* is only selectable if the flag `flask` was set to true earlier. If this is the case, and this choice is selected, lines 5,6,12 will be executed next.

A menu choice is comprised of its choice label, followed by an optional choice condition, and a block of statements associated with this choice.

The player will only be able to select a choice that comes with a specified condition if the condition evaluates to true. If the choice does not have a condition attached to it, the player will always be able to select it.

If the player selects a choice, only the statements in the block attached to this choice will be executed. Afterwards, execution will resume in the first line after the menu statement, i.e. the first line after the last choice's associated block. An example can be seen in Listing 9.

SETTING VARIABLES The `set_flag` and `set_int` statements respectively are a simplification of the full Ren'Py language, which supports multiple ways of setting variables, including via in-line python code using `$`, python blocks, and the `default`-command, which sets a variable only if there isn't a value set for it already; a distinction that only becomes meaningful if changes to the source code between play sessions are permitted, which the Ren'Py engine facilitates to allow developers to update their games after release. In our model, the source code remains unchanged between executions, so the `default`-command is semantically equivalent to setting the variable in the "normal" way.

BRANCHES The branch-statement represents an if-command with an optional else-clause. The following blocks respectively will only be executed if the condition is true (or false). An example can be seen in Listing 10.

```
1  breq "Are you pleased with the selection of fish shaped cakes, translator?"
2  if fishcakes >= 3 :
3      zeiat "Oh yes, Fleet Captain. Very much so."
4      zeiat "Might I ask for some fish sauce, as well?"
5  else:
6      zeiat "How kind of you to ask, Fleet Captain!"
7      zeiat "Indeed, if we might stop by the station soon and get some more..."
8      zeiat "That would be very much appreciated!"
9  breq "That can certainly be arranged."
```

Listing 10: A sample Ren'Py program showing the **branch** statement; depending on the amount of fishcakes, different text is shown to the player. Lines 1 and 9 are shown regardless.

NESTING The production rules for the statements <menu>, <branch> and <label> can generate additional <statement>s, meaning that statements can be nested to arbitrary depth. This nested structure is controlled by indentation, this is omitted from the grammar in Listing 8; an example illustrating nested statements can be seen in Listing 11.

```
1  if bought_tea:
2      breq "Would you like some tea, translator?"
3      zeiat "Oh yes!"
4      if cups > 0:
5          breq "Five will pour you a cup. She also selected this tea set."
6          zeiat "Oh, exquisite!"
7      else:
8          breq "Unfortunately, we don't have any cups just now."
9          breq "Would you mind taking your tea in a bowl, or perhaps a vase?"
10         zeiat "Not at all, Fleet Captain!"
11         zeiat "I've never had tea in a vase before!"
12 else:
13     breq "Good evening, translator."
```

Listing 11: A sample Ren'Py program showing two nested branching statements. The fragment in lines (2-11) will only be executed if bought_tea is true, while lines 5-6 and 8-11 will only be executed if additionally cups is greater than zero, or not, respectively.

UNSTRUCTURED CONTROLFLOW Ren'Py supports unstructured controlflow in the form of the jump-command, which allows jumping to a label-statement anywhere in the program. (An example illustrating use of the jump-command can be found in Listing 12.) This makes it necessary that every label be transparently accessible from anywhere else in the code, meaning that the nested structure needs to either be traversed correctly during or "flattened" before the execution.

```
seiv "Shall I accompany you to the station, Fleet Captain?"
menu:
    "No."
        breq "No, Lieutenant. I will leave you in command of the ship."
        jump ship_ending
    "Yes."
        breq "Yes, Lieutenant. I acquire your assistance."
        jump station_ending
```

Listing 12: A sample Ren'Py program showing off the **jump** statement; depending on the player choice, after displaying one line of text, execution resumes either at the label `ship_ending` or the label `station_ending`.

To achieve this, R'Ast removes the association of `jump`-commands to their target labels in the first translation step between Ren'Py code and AST by resolving all labels to their respective line numbers, eliminating the need to keep track of a dictionary of labels in later steps.

RETURN WITHOUT CALL Additionally, full Ren'Py also includes a *call* statement. It is **not** included in the smaller fragment language this work supports. The `call` statement is similar to `jump`, in that it resumes execution at a target label. Additionally, it pushes the location in the source code from where it was called onto a *callstack*, interoperating with the `return` command to facilitate the creation of subroutines.

The choice to exclude the call command was made because it is not very commonly used and fairly complicated to implement. The `return` command was still included. This makes sense because at the beginning of the execution of every Ren'Py story script, the label `start` is called by the engine, so that (assuming no other additions to the callstack) the first `return` command marks the end of a playthrough, typically passing control back to some sort of main menu. The Ren'Py Engine also passes control back to the main menu if the end of the script file is reached, meaning an explicit `return` statement is actually not needed.

PRESERVING SEMANTICS WITHOUT STRUCTURE In order to preserve the semantics of nested statement structures and allow for unstructured control flow at the same time, some specific steps are necessary in translation. R'Ast first creates an AST that preserves all the structural information of nested structures in a first step. In a second step, the AST gets "flattened" again into a linear sequence, adding additional control flow commands to preserve the semantics of the original nested code. These steps are further elaborated in the following paragraphs with the help of an example.

Consider the Ren'Py example from Listing 9, and find a graphical representation of its AST in figure 13. The structure of the `menu` statement's AST is fully analogous to the corresponging production rule in the grammar shown in Listing 8: A `menu`-node's child nodes include an optional node representing the aforementioned optional say statement, as well as multiple nodes representing the menu choices. A menu choice node then has one child node representing its label, one child node representing the block of statements that is associated with that choice, and finally an optional child node representing the condition that dictates whether this choice is available. The translation from Ren'Py code into this AST structure is therefore straightforward.

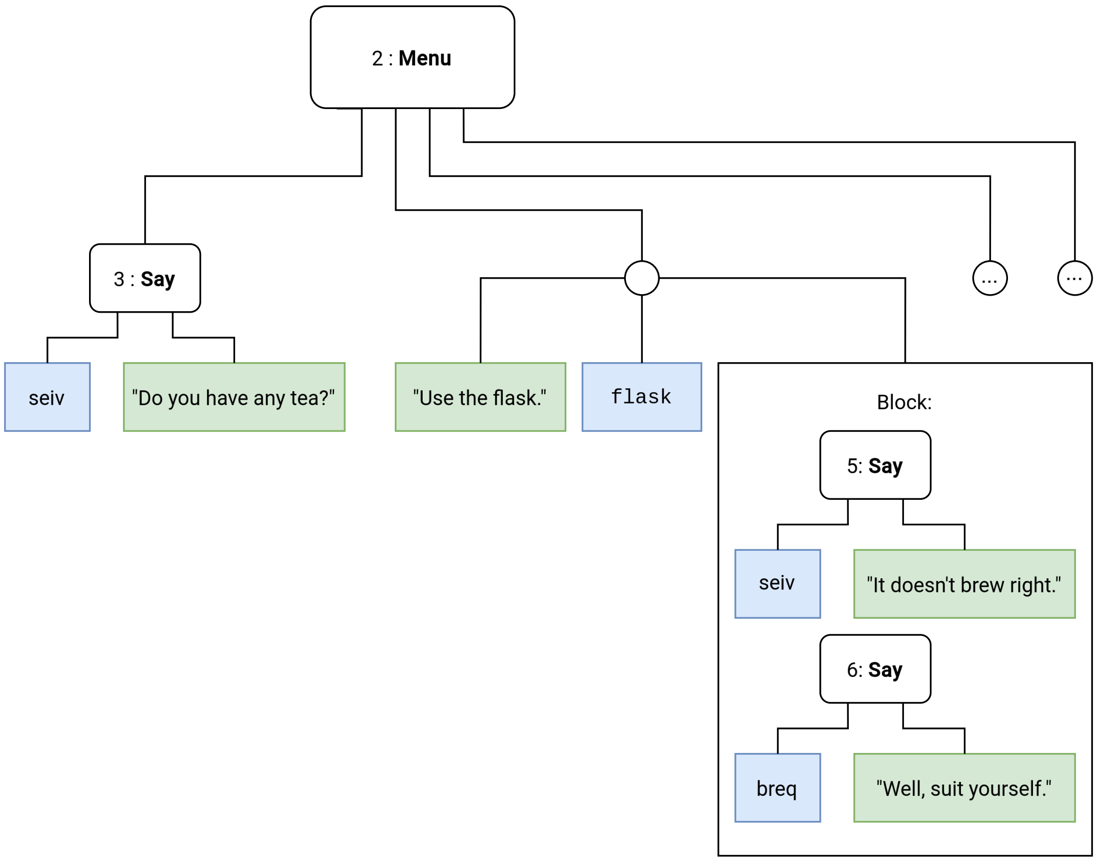


Figure 13: Graphical representation of the AST of the Ren'Py code snippet in Listing 9. Only the first menu option is displayed in full.

To create the unstructured sequence of Maude code, the AST is traversed recursively. Statements only spanning a single line are directly translated into a single line of Maude code. Statements spanning more than one line are translated into multiple lines of Maude code. Note that the AST contains information about source code line numbers; this information is used in this second translation step to add `jump`-statements that facilitate the behaviour expressed by the nested structure. The exact location and target of the `jump`-commands reflect the semantics of the different statements. As an example, when translating the AST representing a `menu`-statement, additional `jump`-commands not explicitly present in the original source code that let execution resume after the `menu`-statement are added at the end of every `choice` block but the very last one.

To illustrate this further, in Listing 14, the original Ren'Py code (the same as in block 9) and the generated Maude code can be compared side by side. (The Maude representations of the different statements will be explained further in 3.4.)

```
1  label seivarden_asks_for_tea:
2  menu:



3    seiv "Do we have any tea?"
4    "Use the flask." if flask:
5      seiv "It doesn't brew right."
6      breq "Well, suit yourself."
7    "Go and buy some, if you like.":
8      seiv "Won't we be late?"
9      breq "No."
10   "No, I don't think so.":
11     seiv "This is hardly civilised."
12 breq "Let's go to the shuttle."
```

```
eq line(1)  = label .
eq line(2)  = menu (
               (if var(flask) cangoto 5),
               (if empty cangoto 8),
               (if empty cangoto 11)) .
eq line(3)  = skip .
eq line(4)  = skip .
eq line(5)  = say .
eq line(6)  = say .
eq line(7)  = jump 12 .
eq line(8)  = say .
eq line(9)  = say .
eq line(10) = jump 12 .
eq line(11) = say .
eq line(12) = say .
```

Listing 14: Comparison of Ren'Py code and generated Maude terms. Every statement that is present in the original Ren'Py code has an equivalent term that is assigned to the original statement's line number. Additional Maude equations, like `eq line(7) = jump 12 .` are added to preserve the semantics of the `menu`-statement without structurally separate blocks for each choice.

PUTTING TOGETHER THE MAUDE PROGRAM Some additional assumptions are made when the Maude program is put together for the interpreter to reason about. While Python is of course a dynamically typed language, allowing the user to assign any sort of data to variables with no regard for health and safety, our model restricts all future variable assignments to the type that variable was first assigned to. The sample we investigated (see 4.1 for details) did not contain any variable assignments that would contradict this limitation.

Furthermore, initially, all integer variables are assigned to 0, and all boolean flags are assigned to false. This assumption does not lead to inconsistent behaviour on correct Ren'Py programs; were a variable ever accessed prior to initialisation, this would lead to a runtime error.

LOST IN TRANSLATION Because our model cares only about structural information, we leave out a lot of information that is present in the original Ren'Py code. This includes the contents of `say` statements, i.e. the actual lines being said and by whom. Display actions like the showing and hiding of images that bear no relevance to the story structure were translated into *skip* statements. Because of this lost information, it was deemed important that the line numbers of the translated program corresponded exactly to the line numbers of the original program to make the process adequately transparent. To achieve this, skip statements were also included wherever there was a line in the original Ren'Py code that did not correspond to a Ren'Py statement (see for example lines 3, 7 and 10 in Listing 14).

3.4 MiniRenRun: A Maude Ren'Py Verification Tool

MiniRenRun is a Maude system module specifying a rewrite theory that simulates an interpreter for the Ren'Py fragment presented in 3.3 and can be used to reason about its state space. The functionality of the interpreter and the way in which the rewrite theory can be used to verify properties of Ren'Py programs will be explained in the following section.

THE STATE SPACE Like we described in 3.1, the state of a MiniRen program at any point during the execution can be described as a 3-tuple (F, I, p). This is represented in Maude by the sort `SystemState`, which is shown in Listing 15. In the following, this Maude syntax will be used to describe the program state.

```
1  sort SystemState .
2  op _;_;_ : FlagState IntState ProgramCounter -> SystemState .
3
4  sort FlagState .
5  op (_=_) : Flag Bool -> FlagState .
6  op __ : FlagState FlagState -> FlagState [assoc comm] .
7
8  sort IntState .
9  op (_=_) : IntVar Int -> IntState .
10 op __ : IntState IntState -> IntState [assoc comm] .
11
12 sort ProgramCounter .
13 op p : Nat -> ProgramCounter .
```

Listing 15: The`SystemState` encompasses the state of the flags, integer variables, the current position of the program counter. The __-operators in lines 6 and 10 allow for multiple assignments to simply be concatenated together.

INPUT MODULES MiniRenRun does not come with a parser; the "source code" has to be present in the form of a Maude module. A sample MiniRen input module that can be run by the interpreter can be found in Listing 16. Such an input module must `include` the `MINIRENRUN` module. This is because it describes specific operators that implement sorts defined in that module. To be able to use the theory for its intended verification purposes, this module also needs to include the `MINIREN-PROP` and `MODEL-CHECKER` modules. `MODEL-CHECKER` is an extra module implemented for core Maude. [Bae, 2014] `MINIREN-PROP` contains property definitions for the model checker to reason about and will be explained in the following section. All flags and integer variable names must be introduced as (0-ary/constant) operators of their respective sorts. The actual "source code" must be given as a sequence of equations using the `line` operator to effectively giving a mapping of every line number to the respective line's statement.

```
 1 mod MINIREN-EXAMPLE is
 2
 3     protecting MINIREN .
 4     including MINIREN-PROP .
 5     including MODEL-CHECKER .
 6
 7     op initial : -> SystemState .
 8     op flask : -> Flag .
 9
10     eq line(1) = label .
11     eq line(2) = menu((if var(flask) cangoto 5),
12                       (if empty cangoto 8),
13                       (if empty cangoto 11)) .
14     eq line(3) = skip .
15     eq line(4) = skip .
16     eq line(5) = say .
17     eq line(6) = say .
18     eq line(7) = jump 12 .
19     eq line(8) = say .
20     eq line(9) = say .
21     eq line(10) = jump 12 .
22     eq line(11) = say .
23     eq line(12) = say .
24
25     eq initial = (flask = false) ;
26                  () ;
27                  p(1) .
28 endm
```

Listing 16: Representation of the Ren'Py program in Listing 9 as a Maude module. Lines 3-5 are imports that are explained further in the following section. Lines 7 and 25-27 concern the specification of the initial state of the program. The flag `flask` (recall that flags are variables that appear in the Ren'Py source code and are thus specific to this code example) is set to `false` initially, and no integer variables are referenced in this example. The program counter starts at 1. Lines 10-23 define the lines of "source code" the interpreter will run.

SEMANTIC RULES As briefly explained in Section 2.2.2, Maude supports specification through term rewriting rules. The interpreter is built from a set of such rewriting rules, transforming one `SystemState` into another `SystemState`, according to the semantic rules of the Ren'Py fragment specified in Section 3.3. Every rule is made up of a left hand side, describing the current `SystemState`, and a right hand side describing the `SystemState` after its application. Every rule is also a *conditional rule*, meaning that it can only be applied if the given condition is satisfied.

The simplest of these rules is the rule concerning the `say`-command, given in Listing 18. To understand this rule, it is helpful to remember that the purpose of the `say` command in Ren'Py is to display text. However, for our purposes of structural

```
    sort Statement .
    op say : -> Statement .
    op return : -> Statement .
    op label : -> Statement .
    op jump_ : Nat -> Statement .
    op menu_ : Choice -> Statement .
    op set_to_ : Flag Bool -> Statement .
    op set_to_ : IntVar IntExpr -> Statement .
    op if_proceedelsegoto_ :  Condition Nat -> Statement .
```

Listing 17: MiniRen fragment showing the different operators of the `Statement` sort. Each operator represents a different kind of Ren'Py statement.

analysis, the content of the displayed text is not relevant and is omitted, as pointed out in Section 3.3. The `say` command effectively does nothing; when this rule is applied to a `SystemState`, the state of the flags `Fs` and Integer variables `Is` remains unchanged. The program counter is incremented by one, so that the next line will be executed.

```
crl [say]   :   Fs           ; Is ; p(X)
            =>  Fs           ; Is ; p(X + 1)
            if (say) := line(X) .
```

Listing 18: MiniRen interpretation rule representing the say statement.

Like all the rewriting rules making up the MiniRen interpreter, it is a *conditional rule*. The `say` rule comes with the condition `if (say) := line(X)`. This is a so-called *matching equation*. It serves the purpose of guaranteeing the rule only gets applied if the line at the current position of the program counter `X` is in fact a `say`-statement; i.e. the shape of the `say`-statement *matches* the current line `X`.
The use of a matching equation over an ordinary equation allows for instantiation of new variables that do not appear in the left side of the rewriting rule. The `say` statement does not introduce any such new variables, and the condition could therefore be rewritten to an ordinary equation. It is merely given as a matching equation to remain consistent with other rules.

The rules implementing the `set-flag` statement and the `set-int` statements respectively are shown in Listing 19. (As the rules get more complicated, it may be helpful to read them bottom to top, beginning with the condition that describes the statement.) On application of the `set-flag` rule, assuming the statement is specifically of the form `set F to Bnew`, the flag state `Fs` is changed. The old tuple `(F = Bold)` is replaced by `(F = Bnew)`.
It is worth noting that our Ren'Py language fragment MiniRen only allows for flags to be set to true or false directly, therefore there is no need to evaluate the boolean

expression Bnew.
The set-int rule works similarly, however, because integer variables can be set to more complicated integer expressions, – or more accurately: can be set to the evaluation result of an integer expression (given the current variable bindings) – it is necessary to evaluate the integer expression Inew before writing it into the state.

```
crl [set-flag]    :  (F = Bold) Fs ; Is ; p(X)
                 =>  (F = Bnew) Fs ; Is ; p(X + 1)
                     if (set F to Bnew) := line(X) .


crl [set-int]     :  Fs ; (IVar = Iold) Is ; p(X)
                 =>  Fs ; (IVar = inteval(Iexpr, (IVar = Iold) Is)) Is ; p(X + 1)
                     if (set IVar to Iexpr) := line(X) .
```

Listing 19: MiniRen interpretation rules representing the set-flag and set-int statements.

The rule implementing the jump statement fairly unsurprisingly sets the program counter to the jump target. Because the jump operator's argument is a line number and not a label, the label statement is vestigial at this point in the translation process. The rule is identical to the say rule. The choice to keep labels as a distinct type of statement was made to make the translated code more readable and the correspondence to the original Ren'Py code more obvious.

```
crl [jump]  :   Fs             ; Is ; p(X)
            =>  Fs             ; Is ; p(Y)
            if (jump Y) := line(X) .

crl [label] :  Fs             ; Is ; p(X)
            => Fs             ; Is ; p(X + 1)
            if (label) := line (X).
```

Listing 20: MiniRen interpretation rules representing the jump and label statements.

The rules implementing the branch statement are shown in Listing 21. For additional clarity, another example comparing the original Ren'Py code and the Maude code is provided in Listing 22.
Assuming both branches of the if-then-else-statement are present: If the condition supplied in the first argument of the if_proceedelsegoto_ operator evaluates to true, the [if] rewriting rule can be applied, (lines 1-4 of Listing 21). This means that the program counter is simply incremented by one ("proceed"). If it is false, the [else] rule can be applied (lines 6-9 of Listing 21). In that case, the program counter is set to Y, which is the second argument of the if_proceedelsegoto_ operator. There is also an additional jump command created in the line where the head of the else-branch sits in the original Ren'Py code (we see this in Listing 22 in line 5.) The target of this jump is always the next line after the branch statement concludes.

```
crl [if]    :   Fs           ; Is ; p(X)
            =>  Fs           ; Is ; p(X + 1)
            if (if P proceedelsegoto Y) := line(X)
            /\ booleval(P,Fs,Is) .

crl [else]  :   Fs            ; Is ; p(X)
            =>  Fs            ; Is ; p(Y)
            if (if P proceedelsegoto Y) := line(X)
            /\ not booleval(P,Fs,Is) .
```

Listing 21: MiniRen interpretation rules representing the branch statement.

```
breq "Are you pleased with the cakes?"
if fishcakes >= 3 :

    zeiat "Oh yes, Fleet Captain."
    zeiat "Can I have fish sauce?"
else:
    zeiat "How kind of you to ask!"
    zeiat "Could we perhaps get more?"
breq "That can certainly be arranged."
```

```
    eq line(1) = say .
    eq line(2) = if var(fishcakes) \>=
        const(3) proceedelsegoto 6 .
    eq line(3) = say .
    eq line(4) = say .
    eq line(5) = jump 8 .
    eq line(6) = say .
    eq line(7) = say .
    eq line(8) = say .
```

Listing 22: Another comparison of Ren'Py code (similar to the snippet in Listing 10) and corresponding generated Maude terms, illustrating the semantics of the `if_proceedelsegoto_` operator.

Assuming only the if-branch is present (which is also valid, recall the grammar in Listing 8 in Section 3.3), the second argument of `if_proceedelsegoto_` is the number of the first line after the if-block concludes, no additional jump is created in this case. Note that so far, it was never possible for more than one rule (or more than one version of a rule with different variable interpretations) to be applied at the same time.

This will change now. Recall first that the argument of the `Menu` operator was specified above to be of sort `Choice`. Review the `Choice` sort declarations and associated operator declarations in Listing 23 to find that terms of sort `Choice` can be of the form `if_cangoto_`, and specify a condition, and a line number, representing a single choice option. They can also be associative, commutative terms of shape `C1,C2,C3...`, (constructed similarly to how words were constructed in the DFA/NFA examples in Section 2.2.2) where `C1`, `C2`, `C3` are `Choice`s themselves, representing an unordered list of choice options.

```
    sort Choice .
    op if_cangoto_ : Condition Nat -> Choice .
    op _,_ : Choice Choice -> Choice [assoc comm] .
```

Listing 23: MiniRen `Choice` sort declarations and associated operators.

Consider now the rule for the menu statement presented in Listing 24.

```
crl [menu]   : Fs            ; Is ; p(X)
            => Fs            ; Is ; p(Y)
               if (menu ((if P cangoto Y) , Cs)) := line (X)
               /\ booleval(P,Fs,Is) .
```

Listing 24: MiniRen interpretation rule representing interactive game menus.

We can see that for the menu rule to be applied, there must be a choice given in the argument of the menu_ operator, which contains a condition P which evaluates to true. (Ren'Py choices that do not come with a specified condition are represented on the Maude side with the empty condition, which always evaluates to true.) Of course, there can be multiple choices with conditions that evaluate to true at the same time, meaning that there can be multiple versions of the menu rewrite rule applicable at the same time.

MODEL CHECKING MINIREN PROGRAMS To use the Maude model checker module [Bae, 2014], it is advised to create a dedicated Maude module that defines properties and whether or not they apply to a given system state (corresponding to the labelling function L in our Kripke structure). We show the full property module in 25. Therein, we declare our sort SystemState as a subsort of the predifined sort State that comes with the model checker module, and use the also predefined sort |= to define the three operators visiting, after, and ending, the first and second being familiar from Section 3.2.

To now query the model checker, one needs to load all relevant Maude modules in an order consistent with the various inclusion relationships. Then one must use the reduce command on a term created with the operator modelCheck specified in the model checking module. For instance, to query whether it is impossible to reach a state that is labeled with the ending property, one would write:

```
red modelCheck(initial, [] ~ ending) .
```

The result of such a model checking reduction query will then either be of sort Bool (and have the value true) or it will be of sort counterexample, and give a path that does not satisfy the queried formula.

```
mod MINIREN-PROP is

   protecting MINIREN .
   protecting NAT .
   including MODEL-CHECKER .

   subsort SystemState < State .

   ops visiting after : Nat -> Prop .
   op flagged : Flag -> Prop .
   op ending : -> Prop .

   var N : Nat .
   var X : Nat .
   var F : Flag .
   var PC : ProgramCounter .
   var Fs : FlagState .
   var Is : IntState .

   eq ((Fs ; Is ; p(N)) |= visiting(N)) = true .

   ceq ((Fs ; Is ; p(X)) |= after(N)) = true
       if X > N .

   ceq ((Fs ; Is ; p(N)) |= ending) = true
       if line(N) == return .
endm
```

Listing 25: Maude module containing operators declaring properties and equations that specify what states they apply to.

4 EVALUATION

In the following chapter, the experiment we performed to evaluate the metrics that were developed in 3.1 will be described. The results of this experiment will be shown in detail. The result will be interpreted, and the limitations of these interpretations will be laid out. Finally, further applications of the metrics and tools developed will be briefly described with a focus on visual novel developers employing these tools and metrics in their development process.

4.1 Experimental Setup

We analysed open source visual novels and compared their performance according to our metrics developed in Section 3.1.
In the following section, the sample and experimental process will be described in Section 4.1. The results of the formally analysed metrics will be presented in Section 4.2. The results and their limitations will be discussed in Sections 4.3 and 4.3.1. Finally, further applications of the tools developed will be briefly described in Section **??**.

SAMPLE SELECTION For our study, we selected 6 open source visual novels from the platform `itch.io`. The novels were selected based on different criteria. All novels needed to be open source, and we required that they were released within the last 3 years. We attempted to have the sample represent a variety of genres.
We also needed to restrict our selection somewhat to not use any Ren'Py language features that are not included in the Ren'Py fragment we restricted our tool's analysis scope to. Table 1 shows an ovewview of this sample.

title	year	language	# of ratings	avg rating
A Valentine's Tryst [1]	2024	english	1	5
Love Bytes [2]	2023	english	0	N/A
My Own Worst Nightmare [3]	2024	german	0	N/A
We Are Passengers [4]	2021	english	20	5
Shrimply Yours [5]	2024	english	0	N/A
Kill The Rabbit [6]	2024	english	2	5

Table 1: Overview of the sample. Number of ratings and average rating were taken from the respective `itch.io` pages on the access date specified in the bibliography.

From Table 2 that has a list of the creator given tags for each of the novels in the sample, we can assume that our sample spans a variety of genres, with two novels being tagged as "Horror". It is somewhat interesting to see the longest novel in our sample still being tagged as "short", giving us an indication that our sample might include particularly short novels.

ref.	tags
[1]	Cute, Feel Good, Furry, Gay, LGBT, Male protagonist, Multiple Endings, Narrative, Romance, Short
[2]	2D, Dating Sim, Funny, Multiple Endings, Open Source, pansexual, Short, Singleplayer, Surreal
[3]	Creepy, Horror, Psychological Horror
[4]	Lo-fi, Surreal
[5]	2D, Global Game Jam, Ren'Py, shrimp, Singleplayer
[6]	2D, Dark, Horror, Mystery, Sprites

Table 2: Tags given to the sample visual novels from their creators.

EXPERIMENTAL PROCESS This sample was analysed with the tools described in Sections 3.3 and 3.4 to measure the variables described in 3.1.
For this work, the bad endings and frightening, shocking or surprising moments were tagged by hand, with an emphasis on trying to match developer intent as close as possible. For example, if 2 out of 5 endings were preceded by the text *"GAME OVER"*, whereas the other three were preceded by the text *"THE END"* being shown, then exactly those first two were tagged as bad endings.
To analyse each sample novel, we created a Maude file with all the queries pertaining to that novel. From the responses to these Maude queries, the final results for each metric were computed with a Scala script.

4.2 Experiment Results

CODE STATISTICS For the first part of the analysis, we analysed the input code and the parsed AST to learn about the prevalence of certain types of statements in the code, as well as the number of different flags and variables that are specified. In Table 3, the findings of this analysis can be seen.

ref.	# r-lines	# r-statements	# flags	# int vars	# say	# menu	# return
[1]	1103	967	2	3	552	11	6
[2]	247	172	0	1	81	6	0
[3]	279	224	0	0	102	6	6
[4]	152	85	0	0	48	2	1
[5]	236	224	0	1	124	6	3
[6]	759	426	0	1	179	7	4

Table 3: Code statistics of the visual novels.

The columns '# lines' and '# statements' describe the number of lines and statements in the Ren'Py source code respectively. Note that these two are not typically equal,

as the source code will often include comments and empty lines. Additionally, some Ren'Py statements take up multiple lines.
The columns '# flags' and '# int vars' describe the amount of different flags and integer variables used in the source code.
Finally, the columns '# `say`', '# `menu`' and '# `return`' show how many `say`-, `menu`-, and `return`-statements are found in the source code respectively. Note that the number of `return`-statements is used to calculate ending positions in the following section, this is because of how the Ren'Py engine embeds the interactive story into its application structure, as further described in Section 3.3.

It should be noted that two sample novels, [3] and [4] do not use any flags or variables. This could hint at those two novels having a more linear story structure (see Section 2.1.1). However, that is not necessarily the case, as in-game menus can still be used to create a non-linear story structure without using any variable state.

REACHABILITY ANALYSIS For every visual novel, we checked every statement for reachability and "obligatoriness", i.e. whether or not a line must be visited on every run. The result of this analysis is displayed in Table 4. It is pleasant to note that most statements are reachable, as unreachable code is usually a sign of programming oversights. None of the sample novels were found to have more than 5% unreachable statements. The amount of statements that is obligatory to reach an ending could be another hint as to certain novels being more linear.

ref.	# statements	#reachable	% reachable	#obligatory	% obligatory
[1]	967	966	99.9	119	12.3
[2]	172	164	95.3	35	20.35
[3]	224	222	99.1	23	10.3
[4]	85	85	100	67	78.8
[5]	224	224	100	29	12.9
[6]	426	426	100	30	7.04

Table 4: Reachability analysis of the of the sample. We give absolute and relative numbers for reachability and obligatoriness.

METRICS FOR FUN In Tables 5 , 6 and 7 the main results of the analysis of the metrics for video game fun we constructed in Section 3.1 are displayed. Each column corresponds to one of these formalised metrics. In certain instances, the prenormalised absolute values as well as the previously computed reference numbers the values are normalised against are given, as well. Prenormalised values are denoted with an $*$.

ref.	# lines	A_1^+	A_2^-*	A_2^-
[1]	1103	true	253	0.22
[2]	247	true	71	0.29
[3]	279	true	35	0.13
[4]	152	true	119	0.78
[5]	236	true	56	0.24
[6]	759	true	106	0.14

Table 5: Analysis of the sample with regards to metrics that relate to the category *Achievement and Completion*.

Beginning with table 5, we can see that every visual novel can reach an ending state. We also note that higher values in A_2^-*, the absolute length of the shortest path, seem to be associated with higher obligatory line counts (see Table 4.) The scatterplot in Figure 26 shows this relationship. We can see that the number of obligatory lines seems to be fairly close to half the length of the shortest path to an ending for our sample.

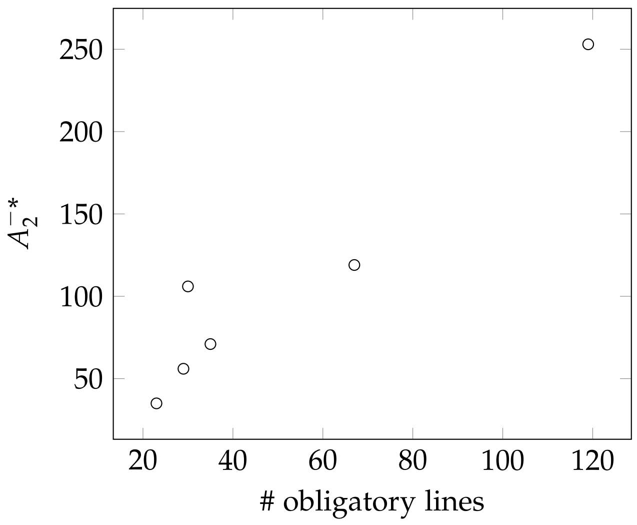


Figure 26: Scatterplot showing the relationship between A_2^-* and the number of obligatory lines.

Looking at table 6, we observe that C_1^+ is equal to the number of menu statements for all rows, meaning all menu statements are reachable. C_2^+ is equal to the number of `return`-statements in all but one case; in the case of [1] there is one `return` statement that can't be reached.

The result for C_3^+ for [3] is consistent with the static analysis; because there are no flags or variables, the only way that ending states can possibly differ from each other is with regards to the program counter, i.e. the `return` statement in which the program ends, meaning that for this novel, C_3^+ – the metric that determines the different states the system can endin – must be equal to the number of `return` statements.

ref.	# `menu`	# `return`	C_1^+	C_2^+	C_3^+	D_1^{+*}	D_1^+	D_2^+	D_3^+
[1]	11	6	11	5	61	4	0.07	0	false
[2]	6	0	6	0	10	0	0.00	0	false
[3]	6	6	6	6	6	5	0.83	5	false
[4]	2	1	2	1	1	0	0.00	0	false
[5]	6	3	6	3	7	4	0.57	1	true
[6]	7	4	7	4	8	4	0.50	6	false

Table 6: Analysis of the sample with regards to metrics that relate to the category *Choice and Autonomy* and *Danger, Uncertainty*.

For D_1^+, we see that two novels ([2] and [4]) do not have any bad endings. Interestingly, the novel [1] has 5 reachable return statements, out of which 2 (or 40%) are tagged as "bad" endings. Compared to this, only 0.07 % of distinct reachable ending states land in one of those bad ending positions, leading us to suppose that most of the lines in the visual novel are on paths to one of the three "good" endings.

Three of the sampled novels have positions that are tagged as frightening, shocking or surprising. In two of those ([6] and [3]), an ending cannot be reached without visiting one of those positions, meaning only [5] fulfills the metric D_3^+, where there is both at least one path to an ending that traverses a "scary" position, and one that is not.

ref.	# reachable	P_1^{+*}	P_1^+	P_2^-	V_1^{-*}	V_1^-	V_2^-	V_3^-
[1]	966	966	1.00	0	0	0.00	0.00	1
[2]	164	164	1.00	0	0	0.00	0.00	1
[3]	222	216	0.97	0	0	0.00	0.00	1
[4]	85	84	0.99	0	0	0.00	0.00	1
[5]	224	213	0.95	0	7	0.03	0.00	1
[6]	426	275	0.65	0	146	0.34	0.00	1

Table 7: Analysis of the sample with regards to metrics that relate to the categories *Making Progress* and *Optimal Variety & Novelty*.

In Table 7, it is most apparent that two of our metrics, P_2^- and V_2^-, yielded 0 as a result for all our sampled novels. This means that none of the novels we investigated contain any infinitely loopable lines (P_2^-), and none of the novels contain any lines that must be visited at least twice to reach an ending (V_2^-). V_2^- being 0 of course means that V_3^- is always 1 (for each novel contains lines that must be visited once to reach an ending, as we can gather from Table 4).

4.3 Discussion

In the following section, we will discuss the results from Section 4.2 with regard to whether or not each of the metrics that were developed in Section 3.1 can be said to accurately measure the degree to which the story structure of a visual novel game contributes to game enjoyment in their respective category.

4.3.1 *Interpretation of the Results*

LINEARITY IN NOVEL [4] Out of the two novels that stood out initially for not using any flags or variables (novels [4] and [3]), novel [4] also has the highest percentage of obligatory statements, substantiating our supposition that it might have a more linear structure than other novels in the sample. Novel [3] on the other hand achieves a similarly low percentage of obligatory lines compared to the other novels in the sample while not using any flags or variables.
Because a linear story leaves less room for player choice and autonomy, this supposition that novel [4] might have a more linear story structure would lead us to assume that novel [4] would also have low values in the metrics measuring choice and autonomy, C_1^+, C_2^+ and C_3^+. Looking at table 6, we can see that this is the case: Novel [4] has lower values than all other novels for $C_1^+$and C_3^+, and the second lowest value for C_2^+.
This means that, assuming that our initial intuition that the numbers of variables and percentage of lines that are obligatory speak to a more linear structure of the story, a correspondence between those values and the metrics we chose to measure player choice and autonomy could have favourable implications for the validity of these metrics. A second study with a larger sample size could investigate if these values are indeed correlated.

OBLIGATORY LINES AND SHORTEST PATHS From the apparent connection between the value A_2^- (Table 5) and the percentage of obligatory lines (Table 4) that is shown in Figure 26, we could make the supposition that the length of the shortest path to the ending might be correlated to the amount of obligatory lines. It would be interesting to specifically see how many lines on the shortest path to the ending are obligatory, and vice versa.

OBSERVATIONS ON REPETITION As we saw in Table 7, our sample showed only values of 0 for the metrics P_2^- and V_1^-, meaning that there were neither infinitely repeatable lines nor lines that needed to be visited more than once to reach an ending position. (The latter also leading to V_2^-, the maximum amount any one line needs to be repeated to reach an ending, 1 for the entire sample.) This means that unfortunately, we are not able to see any differences between the novels in our sample with regards to the variables P_2^- V_1^-, and V_2^-. This seems to be a peculiarity of the sample as infinitely loopable positions are not known to be uncommon in visual novels altogether.

GENRE OBSERVATIONS Novels [6] and [3] were both tagged as "horror" by their creators on their respective itch.io-pages. This would lead us to assume that they should have rather high values in metrics $D_1^+$and D_2^+, which speak to prevalence of bad endings and scary moments in the game. However, considering that bad moments were tagged by hand, it is entirely possible that genre indicators in title, art, and novel script[8] lead to horror novels having disproportionately many moments tagged as scary or bad simply because the tagging person expected them to be there.

CONCLUSION AND LIMITATIONS We were able to measure differences within our sample with regards to 11 out of 13 of our constructed metrics. We found a relationship between A_2^{-*} (the length of the shortest path to an ending) and the amount of obligatory lines. If investigated further, this relationship might give interesting insight in the structure of novels with regards to the categorisation of interactive stories by [Lebowitz and Klug, 2011a] that was described in Section 2.1.1. Specifically, it would be interesting to investigate how many of the lines on the shortest path to the ending are also obligatory (meaning: part of all other paths as well).

The metrics in the category *choice and autonomy* (the amount of reachable ending positions C_1^+, the amount of reachable in-game menus C_2^+, and the number of distinct ending states C_3^+) also seem to connect to the linearity of a story. Specifically, C_2^+ and C_3^+ can determine if a story has multiple endings, and give us a hint about how many different paths lead to these endings. It would be interesting to create similar metrics to C_3^+ measuring the amount of different states that can be reached corresponding to each individual ending statement. It would also be interesting to further analyse the different ending states and see if they can perhaps be meaningfully categorised, revealing relationships between variable states and ending position. We believe that the metrics in the categories of *making progress* and *variety and novelty* were by and large not able to differentiate the sample meaningfully. This might be because the novels in the sample did not contain any infinitely loopable lines whatsoever (P_2^- always being zero). We do not believe the sample to be representative of the totality of visual novels in this regard.

When evaluating the quality of empirical metrics, we refer to three different criteria: Objectivity, reliability, and validity. Our metrics are (with the exception of the ones that rely on tagging of specific visual novel lines as bad etc.) highly objective, because they are measured through mathematical models. For the same reason, they are also reliable in the sense that repeated measurements will always yield the same results. However, the validity of our metrics is largely unclear for the folliwing reasons.
Ideally, we would like to benchmark our metrics on other empirical data measuring visual novel story structure enjoyment. Unfortunately however, there is no such data available. Another possible avenue to compare with other metrics of visual novel enjoyment would be to benchmark against itch.io ratings, however, because

8 Person who tagged the endings and scary positions solemnly swears she did not look at the itch.io tags beforehand.

our sample size was so small, and specifically three out of our six sample novels do not have any ratings at all, this was also not really feasible.
The small sample size also leaves little room for statistical analysis of the result measuring scale reliability. This means we cannot make any claims that speak to the validity of our metrics besides that we have constructed them from a comprehensive, empirically based model.

We hope that by describing objective mathematical metrics that can distinguish visual novels based on different attributes of their story structure, with regards to specific empirically grounded categories of video game enjoyment, we have created a meaningful stepping stone for further research into the deeply relevant question of What Makes Video Games Fun.

4.3.2 Applications

Assuming that story structure meaningfully contributes to the enjoyment of visual novel games, visual novel writers and developers put a lot of work in designing this structure. Visual novels are commonly used as serious games and e-learning tools, which are developed by research teams or other interest groups that are not professional game designers, empirical measures that speak to the linearity of a story could be useful for them.

Furthermore, developers might have certain structural properties in mind when designing a story. Because visual novel programs are programs they can, like all programs, be difficult to reason about by the programmer without additional tools once they reach a certain length and complexity.
The tools described in Sections 3.3 and 3.4 can be used by visual novel developers to specify certain structural properties of their visual novels, and to verifying them. For example, in a detective story, one might want to make sure that a certain piece of evidence must be collected before the player character can accuse a suspect of the crime. This can be done by defining another property *flagged*:

$$\mathit{flagged}(f) \in L(F, I, p) \text{ iff } F(f) \text{ is true.}$$

Assuming that the accusation of the suspect happens in line n, and the flag for the piece of evidence being collected is $\mathit{evidence}_1$, one would then query the model checker with

$$s_0 \vDash \Box\left(\mathit{visiting}(n) \Rightarrow \mathit{flagged}(\mathit{evidence}_1)\right)$$

to find whether the visual novel behaves accordingly.
If the developer is not ready to trust that the flag is always set correctly, but instead knows a specific line m in the story that corresponds with this piece of evidence being found, they could query:

$$s_0 \vDash \left(\neg \mathit{visiting}(n)\right) \mathcal{W}\, \mathit{visiting}(m)$$

5 CONCLUSION AND FUTURE WORK

We created operationalised metrics to measure the impact of visual novel story structure on video game enjoyment with regards to specific categories described by [Schaffer and Fang, 2018] by analysing a Ren'Py visual novel's source code. We formalised these metrics into temporal logic and queries to the term rewriting system Maude. To investigate the Ren'Py source code with regards to these formalised metrics, we also created a tool to translate Ren'Py code into Maude, and a Maude program specifying a rewriting theory that simulates a Ren'Py interpreter within Maude. We were able to measure differences with regards to most of these metrics within our sample. The differences we saw in the categories *choice and autonomy* and *achievement and completion* could yield insight into a visual novel's story's linearity.
However, because we only investigated a very small sample size, we were not able to investigate the validity of the developed metrics. As a future study, it would be interesting to have players evaluate visual novel games with regards to their enjoyment in the relevant categories, and see how their judgements compare to the developed metrics. It would also be worthwhile to investigate internal consistency reliability of the developed metrics on a larger sample.

A APPENDIX

Scala code that was used to generate the Maude queries, as explained by section 3.2. Note that `rs` is a collection of all line numbers that are associated with a statement in the Ren'Py code.

```
val unreachability = rs
    .map(i => s"red modelCheck(initial, [] ~ visiting($i)) .")

val obligatoriness = rs
    .map(i => s"red modelCheck(initial, <> visiting($i)) .")

val checkpointiness = rs
    .map(i => s"red modelCheck(initial, (~ visiting($i)) W O [] after($i)) .")

val unloopability = rs
    .map(i => s"red modelCheck(initial, ~ [] <> (visiting($i))) . ")

val unique = rs
    .map(i => s"red modelCheck(initial, ~ visiting($i) W O [] ~ visiting($i))
        .")

val visitedatleasttwice = rs
    .map(i => s"red modelCheck(initial, (<> (visiting($i) /\\ O <>
        visiting($i)))

val endingUnreachable = List("red modelCheck(initial, [] ~ ending) .")
val endingObligatory = List("red modelCheck(initial, <> ending) .")
val endingWithoutScary = List(s"red modelCheck(initial, ~ scary($novel) U
    ending) .")
```

REFERENCES

Aviation6 (2023). Love bytes. https://aviation6.itch.io/love-bitshd. Accessed 29.03.2024.

Baader, F. and Nipkow, T. (1998). *Term rewriting and all that*. Cambridge University Press.

Bae, K. (2014). The Maude LTL Logical Model Checker. https://maude.cs.illinois.edu/tools/lmc/. Accessed 27.03.2024.

Bae, K., Escobar, S., and Meseguer, J. (2013). Abstract Logical Model Checking of Infinite-State Systems Using Narrowing. In van Raamsdonk, F., editor, *24th International Conference on Rewriting Techniques and Applications (RTA 2013)*, volume 21 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 81–96, Dagstuhl, Germany. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

Bae, K., Escobar, S., and Meseguer, J. (2014). The maude ltl lbmc tool tutorial. https://maude.cs.illinois.edu/tools/lmc/manual.pdf. Accessed 27.03.2024.

Bartle, R. (1996). Hearts, clubs, diamonds, spades: Players who suit muds. *Journal of MUD research*, 1(1):19.

Black Tabby Games (2023). Slay the princess. https://www.gog.com/en/game/slay_the_princess. Accessed 29.03.2024.

Bril, I. and Degens, N. (2016). Applying formal design methods to serious game design: a case study.

Bruni, R. and Meseguer, J. (2003). Generalized rewrite theories. In *International Colloquium on Automata, Languages, and Programming*, pages 252–266. Springer.

casket (2021). We are passengers. https://catsket.itch.io/we-are-passengers. Accessed 29.03.2024.

Cavallaro, D. (2009). *Anime and the visual novel: narrative structure, design and play at the crossroads of animation and computer games*. McFarland.

Celina R. and Kaya E. (2024). My own worst nightmare. https://your-worst-nightmares.itch.io/my-own-worst-nightmare. Accessed 29.03.2024.

Christoph, K. and Tilo, H. (2012). Effectance, self-efficacy, and the motivation to play video games. In *Playing video games*, pages 153–168. Routledge.

Clarke, E. M., Henzinger, T. A., Veith, H., Bloem, R., et al. (2018). *Handbook of model checking*, volume 10. Springer.

Clarkson, M. R. and Schneider, F. B. (2010). Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210.

Clavel, M., Durán, F., Eker, S., Escobar, S., Lincoln, P., Martı-Oliet, N., Meseguer, J., Rubio, R., and Talcott, C. (2020). Maude manual (version 3.1). *SRI International University of Illinois at Urbana-Champaign*.

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Talcott, C. (2007). *All About Maude - A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic*. Lecture Notes in Computer Science. Springer Berlin Heidelberg.

CODI/CLAUDIO (2024). Kill the rabbit. https://collisiondiscourse.itch.io/kill-the-rabbit. Accessed 29.03.2024.

DizzyKa (2017). Profiling your renpygame code for performance. https://lemmasoft.renai.us/forums/viewtopic.php?t=42107. Accessed 29.03.2024.

Documentation, R. (2023). Interactive director. https://www.renpy.org/doc/html/director.html. Accessed 27.03.2024.

Etchells, P. (2019). *Lost in a good game: Why we play video games and what they can do for us*. Icon Books.

Geoffrey Bunting (2023). Why are visual novels suddenly so popular? https://www.eurogamer.net/why-are-visual-novels-suddenly-so-popular. Accessed 29.03.2024.

Hamari, J. and Keronen, L. (2017). Why do people play games? a meta-analysis. *International Journal of Information Management*, 37(3):125–141.

Hamari, J. and Tuunanen, J. (2014). Player types: A meta-synthesis. *Transactions of the Digital Games Research Association*.

Lebowitz, J. and Klug, C. (2011a). *Interactive storytelling for video games: A player-centered approach to creating memorable characters and stories*. Taylor & Francis.

Lebowitz, J. and Klug, C. (2011b). Japanese visual novel games. *Interactive storytelling for video games: a player-centered approach to creating memorable characters and storie.*, pages 192–4.

Lil' Beastman (2024). A valentine's tryst. https://lil-beastman.itch.io/a-valentines-tryst. Accessed 29.03.2024.

Lochman, M. (2020). Program planning through a visual novel-style game. https://scholarspace.manoa.hawaii.edu/bitstreams/344da066-4aed-4bed-9832-dfd0a6576e7a/download. Accessed 27.03.2024.

Meseguer, J. (1992). Conditional rewriting logic as a unified model of concurrency. *Theoretical computer science*, 96(1):73–155.

Øygardslia, K., Weitze, C. L., and Shin, J. (2020). The educational potential of visual novel games: Principles for design. *Ritsumeikan Center for Game Studies (RCGS), Ritsumeikan University*.

Pachipower and rainbowvomit (2024). Shrimply yours. https://pachipower.itch.io/shrimply-yours. Accessed 29.03.2024.

Plagge, K. (2014). Pokémon Alpha Sapphire and Omega Ruby Review. https://www.ign.com/articles/2014/11/18/pokemon-alpha-sapphire-and-omega-ruby-review. Accessed 29.03.2024.

Pnueli, A. (1977). The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57.

Pratama, D., Wardani, W. G. W., and Akbar, T. (2018). The visual elements strength in visual novel game development as the main appeal. *Mudra Jurnal Seni Budaya*, 33(3):326–333.

Quimerc'h, E. (2022). Ren'py graph visualizer - branches flowchart generator. https://github.com/EwenQuim/renpy-graphviz. Accessed 29.03.2024.

Ren'Py (2023). The Ren'Py visual novel engine. https://www.renpy.org/. Accessed 27.03.2024.

Rouse III, R. (2004). *Game Design: Theory and Practice: Theory and Practice*. Jones & Bartlett Learning.

Rozier, K. Y. (2011). Linear temporal logic symbolic model checking. *Computer Science Review*, 5(2):163–203.

Schaffer, O. and Fang, X. (2018). What makes games fun? card sort reveals 34 sources of computer game enjoyment. *Americas Conference on Information Systems (AMCIS) 201*.

Sharp, N. (2018). Top 10 most controversial video game reviews. https://www.watchmojo.com/articles/top-10-most-controversial-video-game-reviews. Accessed 29.03.2024.

Slawik, I. (2017). Generalisierbarkeit von Gamification-Ansätzen in E-Learning – eine explorative Studie. *Gesellschaft für Informatik, Bonn*, pages 273–284.

Team Salvato (2017). Doki doki literature club. https://ddlc.moe/. Accessed 29.03.2024.

The Maude Team (2023). The Maude System. https://maude.cs.illinois.edu/w/index.php/The_Maude_System. Accessed 29.03.2024.