On the Feasibility of using Use Case Maps for the Prevention of Sequence Breaking in Video Games

by

Matthew Shelley

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

Master of Computer Science

in

Computer Science

Carleton University
Ottawa, Ontario

Abstract

"Sequence Breaking" is a type of feature interaction conflict that exists in video games where the player gains access to a portion of a game that should be inaccessible. In such instances, a game's subsuming feature—its storyline—is disrupted, as the predefined set of valid event sequences—events being uninterruptable units of functionality that further the game's story—is not honoured, as per the game designers' intentions. We postulate that sequence breaking often arises through bypassing geographic barriers, cheating, and misunderstanding on the player's behalf.

Throughout this dissertation, we present an approach to preventing sequence breaking at run-time with the help of Use Case Maps. We create a "narrative manager" and traversal algorithm to monitor the player's narrative progress and check the legality of attempted event calls. We verify our solution through test cases and show its feasibility through a game, concluding that our solution is sufficient and feasible.

Acknowledgements

I would like to thank my supervisors, Wei Shi and Jean-Pierre Corriveau, for granting me the opportunity to complete my Master's degree at Carleton University and for all of their encouragement and support throughout my degree. I am thankful for the financial support provided by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the many resources that were made available to me at Carleton University.

I would like to thank my parents, Kathleen and Joseph Shelley, my brothers, Jimmy and Steven Shelley, and my extended family for their patience, support, and every bit of help along the way. I cannot imagine how I would have completed my degree without you.

To my brother, Steven Shelley, and Emma Victoria Peterson, congratulations on your beautiful baby girl, Evelyn-Leigh Kara Shelley, who was born during the early stages of this thesis. Happy (belated) first birthday, Evelyn!

To the Ottawa Bronies, thank you for your friendship. In your honour, I have named the major contribution of this thesis the *Royal Pegasi Algorithm*.

To all of my friends and everyone else who helped me along the way,

Thank you.

Table of Contents

List of Tables

List of Figures

List of Appendices

List of Terms

Term	Definition
Action / Command	a ‘function’ used within an event, such as for displaying text to the screen, prompting the player to make a choice, delaying the next action temporarily, moving entities, etc.
Adverse Feature Interaction / Feature Interaction Conflict / Feature Conflict	an “unwanted interference [among] two [or more] features” [1].
Entity	an in-game object or non-playable character, which may or may not move autonomously and, which may or may not call a script or an event.
Event / Narrative Event	an uninterruptable script, which serves the purpose of furthering the story when called and requires additional legality checking to ensure it is performed in the correct sequence (as specified by the designer).
Event-Id / Event Name	an integer or string that uniquely identifies an event.
Feature	a unit of specific, verifiable functionality that provides value to the end user of an application [2, p. 47].
Geography	a visual (or non-visual) medium (such as the game world, containing towns, paths, and dungeons) in which entities reside, and where the player’s character(s) can move to further the game’s narrative.
Illegal Event	an event that should not be called next based on the player’s progress within the narrative representation.

Legal Event	an event that is allowed to be called next according to the player's progress within the narrative representation.
Narrative	a set of pre-determined events and of event sequences, where an event sequence fulfills the purpose of moving the player through a game's story arc or intended means of progression, providing an experience for the player.
Narrative Representation	a knowledge representation, such as a diagram or 'graph' containing nodes and edges, which describes the set of valid event sequences within a narrative.
Progress	a set of 'positions' (i.e., indicators assigned to 'nodes' within a narrative representation), often associated with specific events, that serve the purpose of recording the events that have recently occurred, as well as determining the events that can be called next.
Script	a string, or sequence of individual lines of code, that can be interpreted to provide functionality such as: • calling commands; and/or, • reading and modifying variables —in addition to providing structural aspects such as labels with go-to (for jumping from one line to another), conditional branching, and comments.
Sequence Breaking	a type of feature interaction conflict, which arises when an intended sequence of events (that is specified by a designer) is not followed.
Sequence of Events	a sequence $(e_1, ..., e_n)$ where each e_i is an event called at time step t_i such that $t_1 < ... < t_n$.

Trigger	a geographical region, which, when entered by the player, calls an event or arbitrary function.

1 Chapter: Introduction

Feature Interaction is a well-known problem in the field of Computer Science [1][3][4] that can affect most applications including video games [5]. In short, an adverse feature interaction or a feature interaction conflict (hereafter feature conflict) occurs when there is "unwanted interference [among] two [or more] features," [1] where a feature is defined as a unit of specific, verifiable functionality that provides value to the end-user of an application [2, p. 47]. In the realm of video games, such conflicts arise when two or more features lead to 'undesirable behaviour' such as visual anomalies, unwinnable situations, or 'inaccurate' storylines (e.g., receiving a reward for defeating a boss before fighting it).

Video games, like any piece of software, have been subject to feature interaction conflicts since their inception. (For clarity, each feature is underlined.) In 1986, Enix's *Dragon Quest* expected the player to rescue a princess to acquire an item, but the item could be found from the beginning of the game effectively skipping the rescue entirely [6]. In 1996, Game Freak's *Pokémon Red Version* and *Blue Version* introduced perhaps the most famous video game glitch of all time: MissingNo. (shown in figure 1) [5][7]. This glitch let the player encounter a very strange Pokémon, when the player spoke to a character from an early town, flew to an island accessible near the end of the game, and swam up a small strip of water tiles. In 2011, Nintendo's *The Legend of Zelda: Skyward Sword* became impossible to finish if players completed tasks in a certain order [8]. Feature conflicts have existed in video games for decades and continue to exist to this day.

Figure 1 MissingNo. from Pokémon Red and Blue

More specifically, in the domain of video game narrative, feature conflicts occur when the designer's intended storyline is not followed by the player. For instance, parts of the story may be seen out of order or skipped entirely, which in turn creates an unintended experience for the player, from the designer's perspective. When a 'narrative' feature, referred to as a narrative event or simply an event, is called outside of the game's predefined narrative sequence, there exists unwanted interference with the storyline, as the integrity of this storyline—the game's subsuming 'narrative' feature—has not been honoured. We refer to this type of feature interaction as 'sequence breaking,' which occurs as a result of an unintended sequence of (narrative) events.

This thesis provides an approach to prevent sequence breaking, and then explores the feasibility of our solution. In doing so, we contribute to the feature interaction problem.

1.1 Motivation

To motivate the necessity for our sequence breaking solution we will argue several claims. One, feature interaction is a significant and well-known problem. Two, sequence breaking is a subset of feature interaction. Three, there are many examples of sequence breaking in video games. Four, complete testing of sequence breaking is intractable so a solution to this problem should exist at run-time. With these claims supported, we will have shown that a solution to sequence breaking is a significant contribution to the field of Computer Science.

1.1.1 Claim 1: Feature Interaction Is a Significant and Well-Known Problem

Our first claim consists of two parts: that feature interaction is *significant* and *well-known*. We can show that this problem is subjectively significant if other researchers believe it is important and that this problem is objectively well-known if considerable research exists on it.

Subjectively, feature interaction is significant as is evidenced by other researchers. For one, Professor Joanne Atlee of the University of Waterloo has written five papers on the subject [3][9][10][11][12]. Two, the International Conference on Feature Interactions, which is in its 11th year as of 2012 [4], states that

> [detecting], resolving, preventing, and managing the feature interactions at different phases of the system life-cycle are more than ever important problems that need to be addressed with cost-effective techniques and tools [13].

Given that an international conference is held annually on feature interaction and that an internationally-known researcher has considerably explored the topic, it is reasonable to suggest that feature interaction is subjectively a significant problem.

Objectively, feature interaction is a well-known problem as is demonstrated by searching IEEE Xplore for "feature interaction," "feature interaction analysis," "feature interaction problem," "feature interaction detection," and "feature conflict". These queries, made in the spirit of a systematic literature search, yielded *7,966*, *4,602*, *1,307*, *1,082*, and *525* results, respectively, at the time of writing. These numbers objectively show that feature interaction is a well-known problem.

Feature Interaction is a subjectively significant and an objectively well-known problem, and is therefore worthy of study.

1.1.2 Claim 2: Sequence Breaking Is a Subset of Feature Interaction

This claim is highly dependent on the definitions of four terms: 'feature,' 'event', 'feature conflict,' and 'sequence breaking'. It is necessary to show that an event is a feature and that sequence breaking is a specific type of feature conflict.

First, we will compare the common definition of a 'feature' against our definition of an 'event'. Recall that a feature is considered a unit of specific, verifiable functionality that provides value to the end user of an application. Our definition of an event is "an uninterruptable script, which serves the purpose of furthering the story when called…" where a script is, in short, a string that can be interpreted to provide functionality. We postulate that, in this context, events do not interleave; only one event can ever run at any time[1]. Though our use of 'event' is more focused than a 'feature,' an event can still be considered specific, as it serves a precise purpose, and verifiable, as it can be tested, while providing functionality to the end user, as it furthers the game's story. Thus, it is reasonable to consider events to be features.

[1] An event can be seen as a forced 'injection' of a story into a game, such that all other interactions—except where player input is required to proceed—must halt.

Second, we will compare the common definition of 'feature conflict' to 'sequence breaking'. Recall that a 'feature conflict' is an unwanted interference among two or more features. Our definition of sequence breaking is a "… conflict, which arises when an intended sequence of events … is not followed." This definition is modified from Simon Carless, who referred to sequence breaking as "tackling the levels of a game in an unintended order or skipping entire sections the designers intend you to play" [14, p. 262]. The primary difference between sequence breaking and a feature conflict is the 'grouping mechanism' as conflicts arise due to 'invalid sequences' of events; whereas, there is no restriction on the 'grouping mechanism' for feature conflicts. The second difference is that instead of "unwanted interference," sequence breaking specifies "unintended gameplay," which means opposition (i.e., unwanted interference) towards the intention of a game's feature(s) as specified by designers. The argument then is whether the overall storyline of a game can be considered a feature with which interference can occur. Given that the entire storyline, in our context, is predefined, thus specific; can be tested, thus verifiable; and, serves the purpose of telling a story to the user, thus functional; we believe the storyline of a video game can be considered a feature. Then, if an event can be called out of order, causing the storyline of a game to lose its integrity or for unintended gameplay to arise, it can be said that a feature conflict has occurred. Thus, it is reasonable to suggest that sequence breaking is a feature conflict.

As per these definition-based arguments, we can conclude that sequence breaking is indeed a subset of feature interaction, and similarly worthwhile of study.

1.1.3 Claim 3: There Are Many Examples of Sequence Breaking

To further suggest that 'sequence breaking' is a type of feature conflict worth exploring, we will provide an illustrative example and real-world examples

1.1.3.1 An Illustrative Example

Suppose a player is required to explore a dungeon and defeat a boss[2] in order to acquire a key item, which is necessary to unlock the next dungeon. Suppose further that the first dungeon becomes destroyed once the player leaves it.

[2] A boss is a special, often unique, enemy who may act as a barrier to progression.

This setup expects the sequence:

1. Enter Dungeon
2. Fight Boss to Receive Key Item
3. Exit Dungeon, Unable to Return
4. Enter Next Dungeon with Key Item

Clearly, if the player somehow managed to bypass the boss fight then they would not receive the key item necessary to enter the next dungeon, effectively resulting in an unwinnable situation as the player cannot return to acquire the missed key item.

This outcome gives the actual sequence:

1. Enter Dungeon
2. Exit Dungeon, Unable to Return
3. Unable to Enter Next Dungeon

The player cannot proceed in any way, thus preventing the intended gameplay. Should a player find his or herself in such a situation, they would not be pleased with the game.

1.1.3.2 Real-World Examples

The website tvtropes.org contains several hundred examples of sequence breaking, of which a few are specified below [6]:

- In an old Apple II game called *The Alpine Encounter*, you had to find an urn with spy information in it within two days in a ski resort. The game has an incredibly elaborate sequence of how the urn is passed from enemy agent to enemy agent that remains the same each game; the trick being ... you can intercept a drop-off. If you don't go *anywhere* until 9:45am (you can simply type "wait 2 hours") on the first day, and stay at the front desk entrance, you can steal the first spy's luggage containing the urn and call the Inspector (someone you shouldn't have met yet) and give the urn to him. This should take about *30 seconds* [to complete the game in its entirety].

- And in *The Elder Scrolls IV: Oblivion*, if you get into Cloud Ruler Temple early, you can retrieve the Amulet of Kings before it is even stolen, cutting out a significant chunk of the main quest and making the rest [of the game] make no sense whatsoever.

- *Tales Of The Abyss* is possibly the all-time champion [of sequence breaking] for traditional console [Role-Playing Games (see section 2.1.1.1)], thanks to a glitch that lets you go anywhere on the world map whenever you want. It's possible to skip ahead in the plot, get critical items and vehicles early, and so on.

- In *Sonic 3 & Knuckles*, the midboss of Ice Cap can be skipped if you're using Tails and fly above the midboss area. This causes a few glitches in Act 2, though. The Mecha Sonic "Nostalgia Boss" fights in Sky Sanctuary can also be skipped this way.

- In *Suikoden II[,]* walking into a specific gate will push it right off its hinges and allow you to skip far ahead in the story, encountering monsters many times your current level and recruiting characters with no good reason to join your army just yet. Alas, move too far forward and you risk glitching the game beyond repair, but if you can survive just one fight, you'll gain enough experience to breeze through the [rest of the] game ...

Though many instances of sequence breaking can be of benefit to the player, there are numerous cases where the experience of the game can also be disrupted.

1.1.4 Claim 4: A Solution Should Exist at Run-Time

It is our belief that a run-time solution to sequence breaking is necessary due to the number of possible combinations of events. As the nature of sequence breaking involves the unintended sequential ordering of events, it cannot be assumed that any arbitrary event is guaranteed to never be called simply because it was not intended to be called at an arbitrary point in time respective to the game's storyline. Instead, we have to assume that any arbitrary event can attempt to be called regardless of the player's progress within a storyline. Therefore, it would be necessary to minimally consider every event combined with every other event (i.e., pair-wise checking), or at least $O(n^2)$ possibilities, prior to the game's execution (e.g., design-time or play-testing). Thus, it is unreasonable to expect designers or play-testers to validate an exponential amount of combinations as such approaches simply do not scale.

For reference, Grand Theft Auto 4 contains approximately 100 missions [15], where each missions consists of several events. It is not practical for designers to consider 10,000 (100^2) combinations of missions (not even counting events) at design-time or to expect play-testers to find how invalid combinations could be called by players.

Instead of resolving and testing all possible event combinations to ensure only valid sequences are allowed, it is preferable to create a system to check if a requested event is legal based on the player's progress within the storyline at a given point of the game. When an attempted event call is deemed legal, it may proceed as expected; otherwise, the call is rejected. In this approach, designers need only consider valid sequences, while testers can verify this preventative procedure does indeed work. Thus, we believe a solution to sequence breaking should exist at run-time due to scalability.

Given that a run-time sequence breaking solution must work within the limitations of a game's environment without hindering playability, this problem further motivates the research for live checking of feature interaction, and is thus a significant contribution.

1.2 Problem Statement

Sequence breaking is a type of feature interaction conflict within the domain of video games that arises when a game designer's intended sequence of events is not followed by the player. The problem applies primarily to designers, who fail to convey their intended plot and may subsequently have to release fixes, and secondarily to players, who may also be affected when they accidentally 'corrupt' the storyline, crash the game, or experience other problems. Since players often perform sequence breaking for their own enjoyment or benefit [16], sequence breaking can also be detrimental to *other* players who become victims of those who take advantage of these problems for the sake of cheating. Even though sequence breaking is a recently-coined term [6], such conflicts have existed in video games since their inception and continue to exist to this day.

As discussed in our "Motivation" section, we believe a run-time solution is necessary. Furthermore, due to the complex nature of games, a solution must work within a game environment using minimal CPU time, to not degrade the rest of game's processes.

1.3 Overview of Solution and Methodology

We consider sequence breaking to be preventable at run-time when we can perform the following routine in an 'unnoticeable' amount of time (i.e. undetectable by the player) within a game environment given an event-id (such as a string or index that uniquely identifies an event), a narrative representation, and the player's progress:

1. Determine if the event associated with the given event-id is legal to call.
2. If ‘yes’, return a ‘success’ value, call the associated event, update the player’s progress, and then:
    a. Update the game’s world to allow the player to call events that are now considered legal; and,
    b. Update the game’s world to prevent now-illegal events from being called, effectively preventing the player from accessing illegal events.
3. If ‘no’:
    a. Ignore the attempted event call; and,
    b. Return a failure value such that an optional developer-defined function can be triggered to correct the game’s state using the set of legal events or notify the player of sequence breaking.

The purpose of our solution is to prevent sequence breaking by only permitting the player to call events in a game that are considered legal based on their current progress. To reach this goal, we associate three primary tasks with events: preloading, calling, and unloading. Preloading is the act of setting up the game world in order for the player to call an event. Calling is the act initiating an event through an entity, a trigger, or an automatic launch through code and, if the event is legal to call, results in an optional script being executed to display a scene for the player to watch or to update game variables. Unloading is the act of removing elements from the game world that were necessary to call the event. If we can preload an event when it becomes legal to call and unload an event when it becomes illegal to call, we believe we can reduce sequence breaking by eliminating illegal calls.

Our primary focus then turns to using the player’s progress within the game’s storyline and knowledge of all valid sequences to determine which events have become legal and which events have become illegal. This process is handled by our Narrative Manager, which runs our *Royal Pegasi Algorithm* on Use Case Maps [17], which specify all valid sequences of events within a game’s narrative. The Narrative Manager keeps track of the player’s progress, which is in turn used and modified by our *Royal Pegasi Algorithm* whenever the player attempts to call an event. With this approach, we can handle

preloading, calling, and unloading of events, but more importantly prevent sequence breaking by rejecting calls to illegal events entirely.

Additionally, we require a representation scheme that is easy for developers to use, clear to readers (with minimal explanation), and capable of describing the features of a game's narrative. This requirement is essential as we expect designers to convert the storylines of their games into a representation scheme that can be used by our Narrative Manager.

To show that our solution is sufficient, verifiable, and feasible, we will consider the narrative structure of five popular games, perform test cases of all covered features, and create a game where sequence breaking can be prevented in real-time. First, we will isolate the 'narrative elements' (e.g., variables, split paths, concurrency[3], loops, and stubs) within five popular games to be included in our representation scheme and algorithm. Second, we will compile a list of features from our chosen representation scheme and algorithm, so we may determine test cases. Third, we will run these test cases against our Narrative Manager to verify actual behaviour against intended behaviour. Finally, we will create our own game with its own legal sequences of events, allowing the player to attempt calls to illegal events through in-game cheats, and show that such sequence breaking can be prevented at run-time. In covering these steps, we believe we will have demonstrated that our solution is sufficient, verifiable, and feasible.

1.4 Contributions

We propose a Narrative Manager and traversal algorithm to prevent sequence breaking within a game environment in real-time, with a major focus on feasibility.

1. First, we analyzed five video games and reviewed literature to compile a list of features necessary to define the narrative structure of a game. We determined that Use Case Maps were a sufficient and easy-to-understand representation scheme for the purpose of describing a game's narrative, due to the key semantics of UCMs including variables, loops, split paths, and concurrency [17].

[3] By concurrency, we mean two or more narrative paths can be taken in parallel. The events of these paths do not execute at the same time, but they can interleave in that the player may call events from one path, then another, and so forth until those concurrent paths end.

2. Second, we created a Narrative Manager to perform legality checks given event identifiers; update the player's narrative progress; handle the preloading, calling, and unloading of events; fetch the current set of legal events; and, optionally run a developer-defined function to handle sequence breaking upon its detection, by querying the current legal set of events.
3. Third, we compiled a list of requirements for covered features of Use Case Maps along with the specifics of our Narrative Manager; created test cases for these requirements; wrote a testing tool to verify our test cases; and then, verified the actual outcome of our test cases against the expectations from our requirements.
4. Fourth, we created a game featuring intended sequences of events, created a Use Case Map to represent these sequences, and then showed that we could prevent sequence breaking in real-time with no detectable delay using our solution.

1.5 Organization

The rest of this thesis is laid out as follows. Chapter 2 explores background knowledge, observations from five video games, and related work. Chapter 3 describes our solution to sequence breaking and contributions. Chapter 4 details the results of various tests on our solution with additional feedback from designers. Chapter 5 concludes the thesis with a summary of the work accomplished and considerations for future work.

2 Chapter: Background, Observations, and Related Work

In this chapter, we will explore the state of the art for preventing sequence breaking in video games. First, we will examine the background of game narrative and feature interaction, followed by errors in games, game development, game testing, and types of errors, ending with the definition of sequence breaking as found in literature. In doing so, we will gather general techniques for managing feature interactions, an understanding of current approaches to developing games and their narrative, and an idea of how games differ from traditional software, particularly from a testing standpoint. Second, we will observe five different games to explore their means to handling narrative and preventing sequence breaking. In doing so, we can determine strengths and weaknesses to these methods, while compiling a list of necessary features to reproduce in our solution (to be shown in chapter 3). Third, we will detail the state of the art for the prevention of sequence breaking, including the exploration of two different representation schemes—Petri Nets and Use Case Maps—for the purpose of defining valid sequences within a game's narrative. In doing so, we will see that sequence breaking is a topic worthy of research as some solutions do exist; Petri Nets have been used define game narrative to reduce sequence breaking; but, Use Case Maps have not been directly associated with the problem. With these steps completed, the reader should have a thorough understanding of the state of the art for the prevention of sequence breaking in video games.

2.1 Game Narrative and Feature Interaction

In this section, we will provide an overview of both game narrative and the feature interaction problem. For game narrative, we will consider genres, narrative elements, and common techniques. For the feature interaction problem, we will consider several types of adverse interactions, some basic approaches, and a few additional methods. Our goal is to establish reasonable knowledge on these topics.

2.1.1 Overview of Game Narrative

Game narrative is the process in which "aspects of a game … [contribute] to it telling a story" [18]. More generally, any narrative is a sequence of events (i.e., actions and happenings), which, when strung together, intends to create a story of interest for the audience. In this sense, we see that time passes differently through the story, which is the

chronological order of events, and the discourse, which is the actual order in which events are presented to the audience [19]. As an example, one may learn of a character's motivation towards the end of a game (i.e., discourse time) instead of at the beginning (i.e., story time) for dramatic effect. For our purposes, we are interested in honouring the intended discourse time of a game's narrative in order to maintain a compelling story.

2.1.1.1 Genres

Table 1 below summarizes major video game genres, notably those with significant use of narrative [20]. Regardless of genre, 'narrative' can be added to any game, even if the player is only following a simple linear sequence. This list is by no means exhaustive.

Genre	Description	Examples
Action	A very broad genre, which "requires good reflexes and quick reaction to overcome challenges," [20] typically with a focus on combat.	*Batman: Arkham Asylum*
Adventure	Heavy emphasis rests on interacting with characters or the environment, such as through the solving of puzzles.	*Myst*
Action-Adventure	A combination of the above genres.	*The Legend of Zelda*
Horror	Any game where horror elements give the primary experience.	*Resident Evil*
Interactive Fiction	User input and game output almost entirely consist of text; similar to a choose-your-own-adventure story.	*Zork*
Interactive Movie	Despite minimal player input, players can alter the story significantly.	*Heavy Rain*

Role-Playing Game	Through "player-dependent character growth," [20] the player explores a world and participates in a story.	*Final Fantasy* *Pokémon*
Shooter	Action games where the player mainly shoots enemies. Variants include first-person, fixed-screen, and rail.	*Perfect Dark* *Goldeneye 007*

Table 1 Common or relevant game genres

2.1.1.2 Narrative Elements

A narrative element of a game "communicates aspects of [the] story to the player" [18]. We can think of narrative elements as the 'who,' 'what,' and 'where' of a game, while the 'when,' 'why,' and 'how' of the game's story are either told or shown to the player. For instance, within the game's fictive world, the player (who) may battle enemies (who) or interact with other characters (who); the player may collect objects (what); and, the player may explore geography (where). The time of the story (when), motivations of characters (why), and actions that occur (how) can be presented in events. In creating a 'narrative-centric' game, it may be desirable to have as many of these types of elements as possible, though it is not strictly necessary.

Every actor (i.e., character) or object (collectively entities) can either interact with or be 'acted upon' by the player. It is important that the "individual actions of players and enemies … be consistent with, and support [the] overall narrative goals" [18].

Geography can be likened to a stage, where all of the action takes place. It may be a two-dimensional, three-dimensional, or even non-visually-existent world—though the latter is an over-simplification (e.g., text-based games). For example, a game's geography may include towns, houses, dungeons, and paths in between. The player's character(s), other characters, and objects all reside within the game's geography, passing through and interacting within this 'medium'.

Events are "individual moments of narrative as the game progresses" [18]. Jesper Juul of *Game Studies* refers to events, using the term 'cut-scenes' instead, as such:

> We should also note that most modern games feature cut-scenes, i.e., passages where the player cannot do anything but most simply watch events unfolding. Cut-scenes typically come in the form of introductions and scenes when the player has completed part of the game [19].

We liken events to features in the sense that they have pre-conditions (i.e., requirements in order to call), triggers (i.e., how to call), and post-conditions (i.e., what to do). These terms will be described in further detail in *section 2.1.2 Overview of Feature Interaction*.

A narrative-based game may include actors, objects, geography, and events. As the player progresses through the game by interacting with other actors or objects, and by passing through the geography, they are awarded narrative events, which further the story.

2.1.1.3 Common Techniques

There are two forms of narrative that can be presented in any game. Embedded narrative is "pre-generated narrative content that exists prior to a player's interaction with the game," [18] such as cut-scenes and back story, which "are often used to provide the fictional background for the game, motivation for actions in the game, and development of [the] story arc" [18]. Emergent narrative, alternatively, is the player's experience with the game that can "[vary] from session to session, depending on [the] user's actions" [18]. While a good "game design involves employing and balancing the use of these two elements," [18] the focus of this thesis is purely on embedded narrative.

In Jakub Majewski's Master Thesis *Theorizing Video Game Narrative*, three basic means of creating embedded game narrative are given:

- The **String of Pearls** approach consists of a "player [moving] from one pre-designed event to the next, with a greater amount of freedom of actions between the events" [21, p. 29]. He claims this "approach is perhaps most common," and that "it is often described in video game design books" [21, p. 29].
- The **Branching Narrative** approach allows the player "to affect the narrative by choosing from pre-designed narrative paths" [21, p. 29]. This approach differs from a "String of Pearls"-based narrative having alternative paths, as that method ultimately follows one linear story with only minor deviations. Unfortunately,

branching narrative requires additional “cutscene material and thus [ends] up costing more” [21, p. 29].

- The **Amusement Park** approach places the player in a world with many possible plots to tackle [21, p. 29]. Examples may include Role-Playing Games, which are very narrative-driven and often contain numerous, optional side quests for players to follow to gain additional rewards such as better equipment, special items, or further back-story.

In addition, Majewski describes a fourth model: **Building Blocks**. In this approach, “designers only create a framework for the narrative, and do not create the narrative itself,” [21, p. 30] thus relying on emergent narrative from the player’s experience.

Furthermore, a personal e-mail exchange with Tyler Moore of FrostFire Games reported:

> In terms of tracking a sequence of events, I think in the perspective of "Gates", which basically mean "can't do X unless condition Y is met". The trick is to order the gates in such a way that a narrative flow emerges. In games where you can sequence break, it's pretty clear this has been the traditional method, once you bypass a gate, you can usually clear our [sic] the ones intended to be behind it.
>
> … this also means there's no way for a script to call a function that returns the "current progress" of the main story arc. However, I tend for the world to be very non-linear, by which I mean, there are very few events with a prerequisite that is story-based (user must witness event A to see event B). The gates are 99% based on skill (soft gates, must know how to use a button) and hard gates (must have the CUT skill to go to the next world).
>
> …
>
> In terms of calling narrative events, depends on how much player interaction is required. Pure cinematic events with no potential for player interrupt would be handled with some type of Co-Routine or linear script that fires off certain events one by one. If some player interaction is required (press a button, select a dialog choice, perform some action, etc.), then it would be handled on a case by case basis, but I would keep the management of the event inside a single script or game object.
>
> Geographical barriers should always be intimately tied in with the narrative design and not be there as magic walls. For example, if a door is supposed to open after event D, then it should be a lot more interesting a story then [sic] "Find Red Key to open Red Door". It should tie into the world design. For my game, I would have a door that's powered down and can't be open until a breaker is thrown. This tells a small story that the doors run on electricity, have no failsafes for manual entry, and are generally poorly designed for impeding you in the first place. Other examples could be damage zones (pools of lava/electricity)

> that can't be crossed until you get the powerup that protects you. Another could be a portal to another world. You don't know the world exists until the portal is opened.
>
> I don't do checks for legality.

Moore makes use of "gates" and geographical barriers to enforce a narrative structure, but he acknowledges that this approach does not validate whether or not any given event is legal based on the player's current progress within the story.

With all of the above approaches to embedded narrative, we see that valid sequences of events are given by the designer in order to build a story. The String of Pearls and Branching Narrative methods clearly specify narrative paths (i.e., sequences of events). The Amusement Park method may consist of many smaller plots, which could be performed concurrently or upon meeting specific requirements from other plots. These plots, of course, are also sequences of events. Moore's method consists of "gates" (i.e., geographical barriers) presented in a linear fashion with a focus on justifying design decisions for the player. These "gates" serve to block the player from progressing. Given that many common means of creating narrative involve sequences of events, it would be beneficial to create a means of enforcing such sequences of events.

2.1.2 Overview of Feature Interaction

Recall that feature interaction is the unwanted interference among features, which are units of functionality that benefit the end-user [2, p. 47]. Features can be thought of as having preconditions, pre-states, trigger events, and post-conditions. A precondition is a set of pre-states, which describe the state of the system prior to execution, and trigger events, which are used to execute the feature when all pre-states evaluate to true. Post-conditions describe the state of the system after a feature's execution or, in other terms, the effect of executing the feature. The state of the system, or the environment, can be considered as a set of variables. In this section, we will briefly look at the types of feature interaction, basic approaches, and additional methods [22][23][3].

2.1.2.1 Types of Feature Interaction

A survey on feature interaction specified five types of adverse interactions, based on their causes [22]. These interactions are only problematic if the behaviour is unintended.

1. **Non-Determinism** occurs when multiple features can be triggered at the same time, yet the system can only permit a single feature to run at that time.
2. **Negative Impact** is similar to non-determinism as multiple features can be triggered at the same time, but instead features are allowed to run in parallel. Problems arise because their post-conditions may not produce the correct result.
3. **Invocation Order** occurs when the sequential order in which two or more features are performed produces unexpected behaviour.
4. **Bypass** occurs when one feature changes the environment in such a way that another feature cannot be called, as its pre-conditions cannot be satisfied. A real-world example briefly existed on *Google Documents* where the user's name covered the "Share" button, effectively preventing sharing, as shown in figure 2.

Figure 2 Real-world example of bypass feature interaction

5. **Looping** occurs when feature F_1, upon completion, immediately calls feature F_2, which, upon completion, in turn immediately calls F_1 again, and so forth.

Of these types, it is apparent that invocation order is most similar to sequence breaking.

2.1.2.2 Basic Approaches

The paper "Composing Features and Resolving Interactions" describes three approaches to the feature interaction problem [9].

2.1.2.2.1 Eliminating Interactions at Design-Time

By formally defining features as separate entities, composing their specifications, and searching those compositions for interactions, conflicts can be resolved by modifying feature specifications [9].

> … most design-time resolutions specify how groups of features behave together, either by adding exception clauses to features, adding supervisors that explicitly interleave the actions of feature combinations, adding rules or feature that explicitly define the behaviour of feature combinations, etc. [9]

However, there are several problems to this approach. First, "features cannot be redesigned independently to eliminate interactions" [9]. Second, "by specifying the behaviour of feature combinations, they counteract the benefits of separation of concerns;" for instance, it may be difficult to resolve future interactions [9]. Third, any features, which have already been implemented, may not be open to redesign. Last, and perhaps most important of all, the number of pair-wise feature combinations grows exponentially as every feature needs to consider interaction with every other feature, resulting in at least n^2 combinations for n features. Realistically, this approach can only work with small applications containing a limited number of features [9].

2.1.2.2.2 Preventing Interactions via Architectural Constraints

An architecture, which constrains and coordinates features' access to shared variables and resources, can help to avoid conflicts. Using a pipe-and-filter system (where the output of one element is used as the input of the next element), messages can be passed from one feature to the next, enabling "features to react to the current system state" [9]. As a feature's reaction is serialized, the next feature may miss a key event that it might have reacted to; thus, features may be over-constrained. Another concern lies in determining which input events should terminate at which features. Unfortunately, while this approach allows for "more flexible access to the basic service and shared resources," it cannot prevent interactions where features' constraints are violated [9].

2.1.2.2.3 Resolving Interactions at Run-Time

Any conflicts not eliminated at design-time or prevented by the system's architecture must be detected and resolved at run-time. Once a feature conflict is detected, the system can make a resolution through "[feature priorities], negotiating compromises, involving the user, rolling back [offending] actions, disabling feature activation, terminating features, or ending the application" [9]. This approach serves as a catch-all case, which relies on some application-specific means of detecting conflicts [9].

2.1.2.3 Additional Methods

In [22], the authors detail four additional methods for feature conflicts: logic, state / model-based, algebraic, and structural. Though not all of these methods directly relate to sequence breaking, it is beneficial to review alternatives before excluding them.

The first approach, which uses logic, "involves the use of logics to describe system desired properties" [22]. An associated axiom system is used to check the validity of this logic, such as through Event Calculus or Temporal Logic [22].

The second approach, which is the state / model-based one, makes use of "state-based languages … to model the behaviour of a feature-based system in terms of abstract machines with sets of states and transitions between the states" [22]. Such a method "is intended for specification of complex, event-driven, real-time, and interactive applications" [22]. Message Sequence Charts may be used to model the behaviour of scenarios, focusing on messages sent between features, to determine if states are reachable [22][10].

The third approach, which is algebraic, is "similar to state-based approaches," [22] but focuses on actions that cause transitions. In particular, the survey mentions sequential 'events' (whose definition differs from ours), making this method of relevance:

> A feature-based system is modelled as a set of communication processes with each process modelling a single feature. Each feature process describes the order in which events can occur (sequentially or concurrently) [22].

Furthermore, the survey provides two references that have proposed Use Case Maps for this approach. Yet upon further investigation, while both references describe features with Use Case Maps, their interest is in finding potential conflicts at the requirements stage [24][25]. Though Use Case Maps have been proposed in a similar context as sequential 'events,' we have not found them in use for real-time detection of these events.

The fourth approach, which is the structural one, defines the organization of a system "in terms of its components – the features" [22]. Again, both sequences and Use Case Maps are mentioned together; "Structural approaches are useful as visual notation for representing sequences of actions and the causality among them, e.g., Use Case Maps" [22]. The survey then adds:

> Structural approaches are not formal in themselves and consequently they are often accompanied by a formal underpinning which describes rules of valid connection between components. …
>
> Architecture-centric methods to handling feature interactions... demonstrate the use of structural approaches [22].

An architectural / structural approach with Use Case Maps may be a reasonable solution.

In summary, we see that algebraic and structural methods relate to our goal of preventing sequence breaking at run-time. Furthermore, Use Case Maps have been proposed in both approaches, so it is reasonable to focus our research on this representation scheme, while pushing other approaches to handling feature interaction aside.

2.2 Errors in Games

In this section, we will explore errors in games by considering the development process and types of errors that often arise. The development process will describe the general approach to creating games, and how testing is regularly handled. Types of errors will be briefly highlighted to show common problems in games. Finally, we end this section with the definition of 'sequence breaking' as found in literature.

2.2.1 Game Development

While development process and time vary greatly from one game to another, there are some common phases including pre-production, production, and post-production. Since the production phase is where most of the work is completed, pre-production is the setup for this work and post-production the maintenance.

Pre-production consists of writing documents to explain the game's concept, and optionally making a prototype. A high concept of a few sentences is first created, then a short pitch detailing the selling points, then a concept document highlighting resource requirements, and then a design document describing major gameplay elements. This phase serves to propose the game to interested or involved parties [26, p. 6].

Production is the phase where the game is actually developed, and thus when most of the work and testing occurs. Games are most often created with the aid of game engines, by abstracting away common details such as rendering of graphics to the screen, detecting collisions among objects, performing physics calculations, and handling user input:

> Generally though, the concept of a game engine is fairly simple: it exists to abstract the (sometime platform-dependent) details of doing common game-related tasks, like rendering, physics, and input, so that developers (artists, designers, scripters and, yes, even other programmers) can focus on the details that make their games unique.
>
> Engines offer reusable components that can be manipulated to bring a game to life. Loading, displaying, and animating models, collision detection between objects, physics, input, graphical user interfaces, and even portions of a game's artificial intelligence can all be components that make up the engine. In contrast, the content of the game, specific models and textures, the meaning behind object collisions and input, and the way objects interact with the world, are the components that make the actual game. To use the car analogy …, think of how the body, CD player, in-dash navigation system, and leather seats make the actual car. That's the content [27].

The first major milestone is creating a playable, "Alpha" version of the game, containing most of the major elements. This "Alpha" version of the game is considered feature-complete in that it contains all major features. The second major milestone is the "Beta" version, which is considered both feature- and asset-complete; only bugs are fixed beyond this point. The "Beta" stage is completed when the game has reached a point where no known problems exist that would prevent it from being shippable. The third major milestone is "Code Release," where the game is made available for review and the majority of known bugs are fixed. The final major milestone is the "Gold master", which will be copied and sold. The "Production" phase contains numerous milestones that are heavy on testing [26, pp. 9-14].

Post-production occurs after the game has been completed and released. This phase focuses on learning from experience and maintenance; it may even be non-existent if the means or resources for additional work are unavailable [26, p. 14].

Despite the efforts of testing during the production phase, bugs often survive that are only discovered when the mass public has access. As every player's experience is unique, we believe a run-time solution for sequence breaking is the best approach.

2.2.2 Game Testing

As discussed previously, throughout the production phase of a game, each major milestone is completed in conjunction with testing. However, it is important to note that video games are not considered productivity software, and are thus tested less intensely

on accuracy and more intensely on the overall experience [28][29]. Below, we will describe testing methods that a game endures during its creation.

2.2.2.1 Internal Testing

Throughout most of development time, game testing is often handled internally by developers, programmers, designers, and Quality Assurance teams. The individuals directly involved in a game's creation are naturally the first to test it. For one, developers may create prototypes before getting into the heavy workload of a game to verify that their ideas will work. Two, a programmer may find a bug while working on a portion of code, and then fix it shortly after. Three, artists may similarly spot visual anomalies and fix them immediately. Four, designers may find new ways to balance their games to keep players interested. With developers, programmers, and designers directly involved in their games on a daily basis, it is natural for them to test and improve upon their work regularly. Alternatively, Quality Assurance teams typically consist of highly-skilled game players who continuously evaluate games for difficulty, play time, and balance while finding and reporting bugs to the development team. Though many bugs and problems found with the gameplay experience can be fixed by developers, programmers, artists, or designers, it is also common for the data collected by Quality Assurance to influence significant game changes [30].

2.2.2.2 External Testing

As the development of a game nears completion, it is common for individuals outside of the organization to become involved in testing. Unlike internal testing, however, the purpose of these instances is more often to ensure that the gameplay experience is enjoyable; bug testing is less of a focus during external tests [30].

Usability tests are performed by selected members of the target audience "to better understand interactions with and reactions to the game" [30]. Such tests occur late enough that the game is near completion, making the development team too familiar to "realize what is obvious or not to players," [30] but early enough that changes can be made without incurring high costs. With controlled psychological research protocols, developers can observe subjects (i.e., players) as they play "without instructions or

interference," [30] in order to gauge how consumers will react to the finish product. Both quantitative and qualitative data can be collected at this time. Typical outcomes of usability tests include better tutorials and clearer interfaces [30].

As an extension of usability testing, beta tests are occasionally given to players over the internet, so they may test the game on their own computers. In this approach, data can automatically be sent back to the development team, while catering to many different computer setups [30].

In general, the purpose of external testing is to ensure that the gameplay experience will actually lead to sales. Most of the time, external testers are not involved in fixing bugs.

2.2.2.3 Expert Review

Expert review is when knowledgeable "usability experts evaluate a product by using usability heuristics" [28]. Despite the popularity in evaluating productivity software, expert review "has not received much attention among game designers" [28].

> The intention of game evaluation is to reduce the obstacles of fun, rather than the obstacles of accomplishments. This is a remarkable difference compared to productivity software evaluation, which focuses on efficiency and ease of use of the product [28].

However, recent trends have shown an interest beyond traditional game testing.

> … with the increasing complexity of contemporary computer games – in terms of the amount of possible user interactions and – behaviours that they afford, as well as the breath [sic] of interaction options between the user and the software/hardware – the informal testing methods traditionally utilized in the game industry, which were adopted directly from productivity software testing, have come under pressure [31].

2.2.2.4 Game Metrics

With user-oriented testing as a major aspect in the development of video games, a central problem is evaluating interaction using traditional approaches while games continue to increase in complexity [31][32]. Borrowing from Human-Computer Interaction, gameplay metrics is the collection of objective data representing user-game interaction and "potentially any action the player takes while playing … including which buttons are pressed, the movement of player-characters within the game world, or which weapons are

used to eliminate opponents" [32]. As gameplay metrics are a new concept, it is not known what data should be tracked or how this data varies among games [32].

2.2.3 Types of Errors

As any error in a game can be reduced to 'undesired behaviour,' many types of errors can exist. While individual errors can be associated with feature interaction, such claims for specific types—outside of sequence breaking—are beyond our scope. For instance, games may be subject to miscommunication between hardware, undetected invalid input, hacking or cheating, or more generally any incorrectly written piece of software. Table 2 summarizes a few common errors that are more specific to games:

Type of Error	Description / Example
Visual Anomalies	Graphics are distorted or generally appear in a manner never intended by designers or artists [33][34].
Audio Anomalies	Similar to visual anomalies, but instead audio problems occur. For example, a sound may 'get stuck,' repeat unexpectedly, or simply play when it should not.
Tunneling	As collision detection is often performed on a 'frame-by-frame' basis, i.e., at discrete intervals, it is possible for objects to quickly pass through one another if a collision is not detected on a specific frame [35]. This type of error can directly relate to sequence breaking, by letting a player sneak past barriers within the game world.
Freezing / Crashes	The game either temporarily freezes or completely crashes, with the latter requiring a reset.
Unwinnable Situations	When the player is placed in a situation, such that it is no longer possible for them to complete or 'win' the game [36].

Table 2 Common errors in games

2.2.4 Definition of Sequence Breaking

Sequence breaking is a type of error that affects a game's narrative. James Newman of Bath Spa University defines sequence breaking as "sidestepping large tracts of the game to access advanced weaponry before the logic of the narrative/structure ordinarily allows" [37]. In another publication, Newman states that:

> Sequence breaking... techniques can be split into two (related) categories. Some are focused on skipping actions or sequences. These breaks are most obviously useful to the speedrunner as they allow large chunks of the game to be sidestepped entirely thereby reducing the amount of space to be traversed. Other techniques allow gamers to reach … objects out of sequence… [38, p. 139].

Simon Carless defines sequences breaking as:

> …tackling the levels of a game in an unintended order or skipping entire sections the designers intended you to play. This usually requires using tricks to achieve the breaks, but occasionally exploiting an engine or map design feature may yield fruit [39, p. 262].

Dr. Mirjam Eladhari's Master thesis further illustrates the problem of sequence breaking:

> A common strategy for imposing a specific story sequence within a highly interactive game is to make progress in the game conditional upon completing a specific sequence of actions or plot points. This is where design problems may arise. Consider, for example, the following clichéd scenario. The player plays the part of a fantasy protagonist (the player character) moving through a medieval world inhabited by various helpful or enemy non-player characters (NPCs). The designers have created a quest: an ailing wizard will give the player character a key to an underground cave system in return for killing an old enemy dragon that the wizard has failed to destroy in time before his own death, and which therefore now threatens the local town. This is programmed into the game. However, as a function of the virtual geography of the game, the player character's interactive possibilities for traversing this geography, and the way the quest is imposed upon the player character, several story outcomes are possible [40, p. 36].

In her scenario, the player progresses through geography in such a way that intended sequences of events may not be followed or may make no sense to the player.

Within literature, the term 'sequence breaking' is generalized to gameplay outside of the intended sequence of events, which agrees with our definition from chapter 1. In this dissertation, we postulate that sequence breaking primarily occurs through the medium of geography, as the player can pass geographic barriers, either intentionally or on accident, to enter areas, which should not yet be accessible. The section ahead provides support for this postulate based on real-world examples.

2.3 Game Observations

In reviewing the state of the art for the prevention of sequence breaking, it is necessary to explore how commercial games handle their narrative by looking at real-world examples. With each game, we will briefly allude to its relevance within this thesis and describe its general gameplay. Afterward, we will examine how we believe the narrative structure was handled based on five hours of gameplay. The goal of these observations is to further understand narrative approaches and their weaknesses within commercial games.

2.3.1 *The Legend of Zelda: A Link to the Past*

The Legend of Zelda: A Link to the Past was first released in English on the Super Nintendo Entertainment System on April 13, 1992 [41] and went on to sell 4.61 million copies [42]. Developed by Nintendo, *A Link to the Past* formed the third game in *The Legend of Zelda* series. By some sources, it is considered among "the greatest video games of all time" [43]. In this 2D action-adventure game, the player controls Link on his journey to save Hyrule, rescue the seven descendants of the Sages, and defeat Ganon. The story progresses as Link explores dungeons, fights enemies, defeats bosses, and collects key items to aid him on his quest.

From our experience with *The Legend of Zelda* series, we postulate that a consistent narrative structure (or means of progression) exists. In general, the player requires a special item to move forward in the world, but that item cannot be obtained until the player has found it in a dungeon and, in most cases, defeated the dungeon boss using that item. *The Legend of Zelda* series has been successful with this formula. During our playthrough, we saw geographic and narrative barriers to progression, aligning with our postulate on the series' recurring structure.

2.3.1.1 Geographic Barriers

There are times where it can be seen that the narrative structure could be broken if the player could simply bypass some 'geographic barrier'. For example, rocks requiring a special glove to lift may block the player from progressing to another part of the map. If, by some means, the player could get past these barriers, they could access parts of the game that should not yet be available.

In fact, this type of barrier can be bypassed through glitches or unintended play [44]:

> In The Legend Of Zelda A Link To The Past, it's possible to get items out of the temples without actually defeating the bosses. This allows the player to visit locations that he shouldn't be able to and get better items or more heart pieces before finishing the first temple. It's also often recommended to do certain dungeons out of order to make the game easier.
>
> It is quite possible to glitch oneself into the majority of the Dark World before the confrontation with Agahnim. By combining this with the more traditional "Get item, skip boss" format of Zelda sequence breaking, it can lead to such shenanigans as gaining the Tempered Sword before Zelda's kidnapping, obtaining the Golden Sword, using the Golden Sword to defeat third boss of the Light World, and then retrieving the original Master Sword (which you never received in the first place) and tempering it. Again.

With another glitch, it is also possible to get hit into a wall early on in the game, and then walk through all in-game barriers directly to the final room of the game where the last boss resides [45]. Doing so allows the player to 'beat' the game in about ten minutes.

Geographic barriers create curiosity for the player, but they can often be overcome.

2.3.1.2 Narrative Barriers

From the name itself, narrative barriers are more forceful when it comes to keeping the player within the narrative structure of the game. These barriers rely on in-game criteria to ensure the player is doing what they should be allowed to do at the given moment.

Perhaps the most obvious example of a narrative barrier from this playthrough is the Master Sword. Even though we were able to locate the Master Sword within the foggy forest, we could not do anything with it. The game insisted that all three pendants be collected before obtaining the Master Sword.

Continuing with that example, another narrative barrier occurs on the outside, upper level of the castle where a door is blocked by a magical field. Only with the Master Sword can this magical field be destroyed, allowing for the door to be entered.

Narrative barriers rely on variables and constraints to ensure the intended narrative structure is followed. It is not obvious how the player could skip these types of barriers.

2.3.2 *Super Mario 64*

Super Mario 64 was released on September 26, 1996 [46] in the United States as the launch title for the Nintendo 64. Developed by Nintendo, the game went on to sell over 11 million copies [47], making it one of the best-selling video games of all time and the best-selling video game on fifth-generation platforms. Additionally, Mario and Super Mario are the best-selling video game franchises of all time.

In *Super Mario 64*, the player controls Mario through large, three-dimensional, open-ended levels to collect Power Stars and ultimately rescue Princess Toadstool from Bowser. Levels are entered through the Princess's Castle, which acts as a central hub, typically through paintings. In general, the flow of the game consists of entering a level, collecting a Power Star, and then returning to the castle to repeat. Once collected, a Power Star can be reclaimed but will not add to the total Power Star count. As the player collects Power Stars additional levels and areas of the castle become accessible.

Super Mario 64 has no strict path of progression outside of levels being unlocked when so many Power Stars have been collected or keys have been received by defeating Bowser. In fact, it is very common for a player to obtain a Power Star in a late level, return to the castle, and then collect a Power Star in an early level. Unlike *A Link to the Past,* the player is not guided along a specific narrative path [48].

2.3.2.1.1 Constraints

This game relies heavily on constraints to limit the player's progress throughout the castle and access to levels. By requiring a minimum number of Power Stars to unlock star doors, the player is forced to play earlier levels until they have received enough Power Stars to continue deeper within the castle. Additionally, the player is required to defeat Bowser to acquire keys – so even the star count is not the one true means of progression. Using a single variable, two keys, and a looping mechanism most of the narrative structure of this game can be modelled with constraints.

However, these constraints are not as strictly enforced as one might expect. In theory, levels should not be accessible without the minimum Power Star count or correct key.

But, in reality, the player can bypass geographic barriers through the use of glitches. It can be shown that these constraints are only verified at specific areas, such as doors.

For instance, it is possible to get past the 50-star door on the second floor with fewer than 50 Power Stars. Doing so grants access to all levels requiring 50+ Power Stars:

> [Start] at the bottom of the staircase that leads to the star door. Position the camera so that it's facing the stairs. Do a fowards [sic] long jump, and turn it into a backwards long jump so that you are going up the stairs. As soon as you start jumping on the stairs, start tapping A as fast as you possibly can. With much luck, you will start to "Rapid Jump" up the stairs, and you will go straight through the star door. Don't expect this rapid jump thing to happen the first time. It will take loads of attempts before you do this ... [49].

Even worse, the never-ending staircase can be skipped with less than 70 Power Stars, allowing the player to beat the game early [50]:

> Yes, believe it or not, but it is actually possible to get to the top of the endless stairs. I guess they really aren't endless, huh? Anyway, this glitch is not very easy to pull off. To do this, you must be able to perform a backwards long jump. Get ontop [sic] of some of the stairs. Face the bottom of the stairs, then do a fowards [sic] long jump toward the bottom of the stairs. As soon as you do the long jump, move the control stick from forwards [sic] to backwards to make Mario lose momentum to the front and gain it for backwards long jumps. Don't let go of Z. When you hit the ground, immediately hit A again and you will do another long jump. Continue this process and you will start doing backwards long jumps up the stairs. Now, this is the tricky part. Start tapping A as fast as you possibly can. If you stop doing backwards long jumps, then start over. Anyway, if you get lucky, you will begin to "Rapid Long Jump" ... If you begin to do this, you will quickly go up the stairs and go directly to the top [49].

Based on the glitches described above, we can see that *Super Mario 64* uses constraints combined with geographic barriers. For a typical player this approach is sufficient, but a devious player could sequence-break the already-limited means of progression.

2.3.3 *Grand Theft Auto 4*

Grand Theft Auto 4 was released for PlayStation 3 and Xbox 360 on April 29, 2008 where it quickly broke industry records, grossing more than $500 million in its first week of release [51], and ultimately shipped over 22 million copies worldwide [52]. The Grand Theft Auto series ranks among the best-selling franchises in video game history.

Grand Theft Auto 4 is a very narrative-driven game, which offers an expansive world to explore. When the player is not stealing cars, driving recklessly, killing others for the fun

of it, or generally 'enjoying' the open world, they can progress the game's story through missions – which often contain these aspects, but in a more-structured way. Missions unlock more missions, areas of the world, weapons, places to live, and other rewards. As the game's story unfolds the player experiences the life of Niko Bellic as he rises through the world of crime in Liberty City.

Through a sophisticated use of narrative barriers, we see that little sequence breaking is possible. Both inside and outside of missions, the game assumes the player will go anywhere and do anything, so it prepares itself accordingly. Within missions, the game prevents the player from interfering with scripted events – going so far as to move objects, which interfere with the requirements of the mission. This feature makes it impossible for a player to cheat the game by using knowledge from failed attempts. Missions themselves depend on previous missions to be completed in order to proceed. Outside of missions, the player can access areas that should not yet be available, but is penalized with a six-star warning level, which brings out the army to pursue them, and a lack of interesting activities. Due to a complex use of narrative barriers and knowledge of scripted events, sequence breaking is impossible to the best of our knowledge.

2.3.4 *Pokémon: SoulSilver Version* and *HeartGold Version*

Pokémon: SoulSilver Version and *HeartGold Version* were released in English for the Nintendo DS on March 14, 2010. While these games are enhanced remakes of the earlier *Pokémon: Silver Version* and *Gold Version*, released in 2000, they had been updated to make use of the newest Pokémon game engine – effectively mimicking the style of the newer Pokémon games. Developed by Game Freak and published by Nintendo, the games went on to sell 11.9 million copies combined [53]. Outside of this instalment, the Pokémon series itself has sold over 215 million games [54], making it the best-selling video game franchise after Mario and Super Mario.

The basic concept of the Pokémon games, including *SoulSilver Version* and *HeartGold Version*, consists of capturing creatures known as Pokémon for battle and trade; earning badges by taking on gym leaders; becoming the Regional Champion by defeating the Elite Four; and, if the player is so inclined, collecting one of every species of Pokémon for their Pokédex, of which there are currently 649.

2.3.4.1 Geographic and Narrative Barriers

Pokémon: SoulSilver Version and *HeartGold Version* follow a near-identical approach to their narrative structure as *The Legend of Zelda: A Link to the Past*. In particular, there are obstacles, which should only be passable when a certain "Hidden Machine" can be used. The player needs to locate these Hidden Machines, and then gain the ability to use them by collecting badges, which are narrative barriers. This approach means that only by progressing the game's narrative should it be possible to get past geographic barriers. Notably, *A Link to the Past* similarly requires specific key items, which are usually obtained by exploring dungeons, in order to pass its geographic barriers. Again, we see the use of both geographic and narrative barriers, telling us that after 20 years of game development this approach still remains.

The existence of geographic and narrative barriers can be proven with two videos where the player uses a walk-through-barriers cheat. The first video shows the player walking from New Bark Town to Kanto without using HM Surf, and then ignoring a guard who tries to block their path for not meeting some requirement [55]. The second video shows that a narrative event (i.e., Team Rocket's takeover of the Radio Tower) does not load as the player has not reached that point in the story; we also see that the player still cannot use the train as they lack the privilege [56]. When the player cheats to skip geographic barriers, some narrative barriers still enforce parts of the intended progression.

2.3.5 *Bastion*

Bastion was released via Xbox Live Arcade on July 20, 2011 and Steam on August 16, 2011. Developed by Supergiant Games as their first title, the game has received wide praise and numerous awards, including "Best Narrative in a Game" [57].

In this action role-playing video game, the player takes on the role of "the kid" as he explores isometric, floating worlds to collect cores to rebuild the Bastion after the "events of Calamity." As an added feature, the entire game is narrated.

The typical progression of gameplay as we have observed consists of completing a level, returning to the Bastion, upgrading the Bastion when possible, unlocking new levels, and

then repeating the process. From our play-through, we found Bastion to have a linear narrative with the occasional side quest, which had little to no impact on the story.

2.3.5.1 Knowledge of Progression

In playing Bastion, there was a sense of 'intelligence' within the game, as though it knew where we should be or what was happening. We attribute this belief to both the narrator and the use of arrows. The narrator tells the story as it unfolds, having regular, brief messages to say, even for instances like dying to an enemy. The narrator helps to move the story along. The arrows directly point me where to go next, and are relevant upon sight. By some means, the game maintains knowledge of our current 'narrative state'.

Unfortunately, there was one occasion where the game did make a mistake. When we had returned to the Bastion before completing "The Workmen Ward," the game told us that we had found "some of the materials we need," and then placed the narrator's in-game character next to the monument to unlock levels. Naturally, we believed we must have found something useful, which we could use at the monument; the narrator further suggested we knew what to do. However, it seemed the game expected me to complete "The Workmen Ward" first, and then use the item found there at the monument instead.

2.3.5.2 Central Hub

The Bastion itself acts as a central hub, which the player can return to after each level in order to move to the next. Were a directed graph to be created, the Bastion might represent the edges between each level (the vertices), as it is the means of transition.

2.4 Sequence Breaking

We acknowledge that a few solutions to sequence breaking have been attempted as will be explored in this section, and thus we make no claim that our solution is the first to the problem in general. Rather, to the best of our knowledge, we believe that a solution has not been attempted at run-time with Use Case Maps, making our solution, and thus this thesis, an original contribution to the state-of-the art. Throughout this section we will examine various solutions, with a focus on those depending on sequence representations.

2.4.1 Object-Oriented Story Construction

Eladhari provides two major techniques for better handling a game's narrative structure. The first technique is "causal modelling for game logics," which is the creation of causal relationships between events [40]:

> This analysis suggests that it may be possible to define a general set of normal forms for the causal relationships in story logics. Assuming a representation of causal relationships that links a set of causes to a specific effect, such normal forms for story logics should at least:
>
> - extract recurrent subsets of causes representing independent events as separate cause-event relations (an analog of Codd's first normal form for relational databases)
>
> - eliminate irrelevant causes from cause sets (an analog of Codd's second normal form)
>
> - separate multiple effects of a common set of causes into multiple relations, one for each effect (an analog of Codd's fourth and fifth normal forms)
>
> - eliminate interdependencies between causes within any single relation (an analog of BCNF) [40, p. 40]

The second technique, and primary contribution of her Master thesis, is "Object Oriented Story Construction":

> In the following definition I am not making a complete transferral of everything that the term object-oriented includes when it comes to the development of software onto the concept of Object Oriented Story Construction. Instead I am borrowing certain features that I believe to be useful when creating and analysing story driven computer games.
>
> Object Oriented Story Construction is to let all objects in the world have integrity and contain their own stories, functions, conditions, possible developments and counter reactions.
>
> If there is a story teller in the world it too will be a story carrying object, and not with necessity all-knowing.
>
> The player character, the player entity which the player controls, is the subject.
>
> Everything which is possible to name in the world is an object, may it be doors to open, characters to meet, boats to ride, monster to kill or things to find.
>
> …

> That an object has its own integrity means that the information available in the object is only available via itself, and that the fetching of information can only be done on the object's conditions [40, p. 42].

With this strategy, all objects and characters (i.e., actors)—entities in our context—are given knowledge of the game's story in order to make them more intelligent with the overall narrative goals. This method can prevent sequence breaking by avoiding the dependence on specific sequences of events, and instead making sense of the player's current involvement within the narrative. Most importantly, we remark that this work confirms that sequence breaking is indeed a valid research topic.

2.4.2 Sequence Representation

As the narrative structures of video games commonly rely on 'sequences of events,' it is only logical to consider representations for these structures. While such representations may be used by designers to create diagrams to describe the various means in which players progress through a game's story, we believe that these diagrams could further be used beyond design-time to provide knowledge to the game itself. In turn, we believe that this knowledge could assist in the prevention of sequence breaking. Thus, representation schemes of sequences are very important to our work.

Throughout this subsection, we will examine both Petri Nets and Use Case Maps, with greater depth on the latter. Though we have chosen Use Case Maps for our research—due to their ease-of-use, clarity, and ability to represent features of game narrative—it is important to consider an alternative representation, notably one that has been applied to game narrative, to further justify our decision. For further motivation of Use Case Maps, see chapter 3. By the end of this subsection, we will show, to the best of our knowledge, that Use Case Maps have not been used to prevent sequence breaking in video games.

2.4.2.1 Petri Nets

Petri Nets are a graphical notation that has been used to express game plots [58][59][60][61][62]. This representation scheme includes places, tokens, transitions, and transition functions. Places, shown as circles, are either input or output from a transition. Tokens are simple markings, shown as pellets, contained within places, which may be moved about the Petri Net. Transitions, shown as rectangles, are actions that can occur.

Transition functions, shown as arrows, "fire [transitions] according to tokens' [locations] in [places], or … add or remove tokens from [places]" [58]. In this section, we will examine how such a representation scheme has been applied to game narrative.

In the paper "Petri Nets for Game Plots," the authors "developed a technique for authoring a nonlinear plot and for managing a story according to the plot in an interactive story-based virtual reality application" through the use of Petri Nets applied to artificially intelligent, autonomous actors, within a large virtual world. Though their "approach presents a compromise between pure scripting and pure emergent narrative", the authors admit that their "portrayal of even a small story … is fairly complicated" [58].

In Felix Martineau's Master Thesis, "PNFG: A Framework for Computer Game Narrative Analysis," he "[describes] a high level computer narrative language, the Programmable Narrative Flow Graph (PNFG)," [61] which builds atop Narrative Flow Graphs (NFG). These Narrative Flow Graphs are Petri Nets that define a game's narrative structure, as directed acyclic graphs (DAG) are said to be too limited:

> Traditional, text narratives can be simply represented through a straightforward DAG structure. Narratives in computer games, however, require a more complex presentation system, including the ability to represent … narrative choice, … [which is] not possible with a DAG format [62].

PNFG is then a language that "maps onto a low level representation [i.e., NFG] suitable for expressing and analysing game properties" [60]. It is necessary because:

> Flaws in game narratives can be easily identified during the gameplay, but we currently lack the tools to analyze these narratives … to detect these flaws and have good narrative properties. From that problem arises the need for a high level programming language that is built on formal principles. Such a language will provide the strong base that is needed to formally analyze these narratives [61].

Unfortunately, this approach is applied only to games within the Interactive Fiction genre, which Martineau refers to as purely textual. Given that most games involve some sort of traversal through geography, especially as a means to perform sequence breaking, this solution is not adequate for our objectives. Notably, however, we see that the representation of a game's narrative is indeed a topic worthy of research.

We hypothesize that the Petri-net based representation scheme is not user-friendly, primarily due to the need to understand and use tokens, and secondarily due to the

‘diversion’ away from the overall narrative, as a designer needs to incorporate places, tokens, transitions, and transition functions. We believe a representation scheme, given a simple description, should be readily understandable to anyone who might need to use it.

2.4.2.2 Use Case Maps

The article “Combining UCMs and Formal Methods for Representing and Checking the Validity of Scenarios as User Requirements” describes Use Case Maps:

> Responsibilities are generic processing activities indicated by crosses representing actions, operations, tasks, etc. Components represent generic software entities (objects, processes, databases, servers, etc.) as well as non-software entities (e.g., actors or hardware). The relationships are said to be causal because they link causes … by arranging responsibilities in sequence, as alternatives, or concurrently. Essentially, UCMs show related use cases in a map-like diagram, whereas UCM paths show the progression of a scenario along a use case. A UCM is intended to be intuitive and high-level. … Thus the UCM view represents a useful piece of the puzzle that assists in bridging the gap between requirements and design [63].

2.4.2.2.1 Use Case Maps and Narrative

It is not uncommon to locate literature, which references the terms ‘narrative’ or ‘story’ in connection with Use Case Maps [63][64]. Typically, these words are meant as a way of describing use case scenarios. As the purpose of Use Case Maps is to represent causal sequences, it is inaccurate to believe narratives or stories (even in our context) have never been described with such diagrams.

2.4.2.2.2 Use Case Maps and Video Game Narrative

To the best of our knowledge, Use Case Maps have not been used to represent video game narrative in a non-trivial manner (e.g., beyond student assignments to learn UCMs). From our searches for *use case maps* + *video games* (and many variants) on Google, Google Scholar, IEEE Xplore, and ACM Digital Library we found 10 articles with ‘notable’ relevance and 9 others worth skimming. Of these articles, few directly connect video games with UCMs while the rest cater to either of these subjects. Below, we summarize current literature to show that UCMs and video game narrative have not been connected, especially for the purpose of preventing sequence breaking.

"Modeling in Software Architecture" [65]

This paper reviews multiple textual and graphical modeling notations, including Use Case Maps. Textual notations depict architecture in a text file, usually in a linear fashion, making it difficult to express graph-like structures but easy to represent hierarchical information. Graphical notations require additional overhead in developing tools for designers, but offer more control and better describe non-hierarchical information such as graphs. In particular, Use Case Maps aim to separate "how it works" from "how it is constructed" in order to specify high-level system behaviour by "[exploiting] human-pattern recognition abilities." Use Case Maps "[enable] stakeholders to visualize, think about, discuss and explain the overall behaviour" of a system. While the survey within this paper discusses the benefits of Use Case Maps, its only 'game' reference falls outside the section of Use case Maps.

"High Level, Multi-Agent Prototypes from a Scenario-Path Notation: A Feature-Interaction Example" [66]

This paper presents "a novel approach, centering around Use Case Maps (UCMs), for linking together behaviour requirements, software design models and BDI-style reference models" for the purpose of resolving feature interaction conflicts within the telephony domain. While Use Case Maps are shown to be a beneficial tool for dealing with the feature interaction problem, they are not applied to video games.

"Use Case Maps: A Visual Notation for Scenario-Based User Requirements" [67]

This paper extends the Use Case Map notation for use in four dimensions—Task, Dialog, Structural, and Usability—by creating two types of UCMs: Conceptual (CUCM) and Physical (PUCM). The *Task Dimension* "represents tasks that are relevant for interactions." The *Dialog Dimension* "explains the style of human-computer interaction and also describes the sequence of dialogs that can take place between the user and the system." The *Structural Dimension* "identifies the objects comprising the user interface, their grouping and specifies their layout." The *Usability Dimension* "[measures] the usability of the use case maps by using usability metrics." "The CUCM and PUCM together integrate the four dimensions stated above and capture a complete picture of user interface and usability requirements." This extended UCM notation could be applied to

the domain of video games, where human-computer interaction and user interfaces are prominent, but there is no mention of such application—and we later argue (in chapter 3) that Use Case Maps are a sufficient representation for video game narrative.

"Deriving Interaction-Prone Scenarios in Feature Interaction Filtering with Use Case Maps" [68]

This paper proposes heuristics on Use Case Map paths for the purpose of finding scenarios that are prone to feature interaction conflicts. Their solution produces a table of filtering results given a UCM to indicate scenarios "(1) [where] FI occurs, (2) [where] FI does not occur or (3) [are] FI-prone." The authors admit that they need to "apply the proposed method to more practical services, which may reveal more effective heuristics for FI filtering." Again, it is shown that UCMs are a viable approach for detecting feature interaction, but there is no reference to video games.

"Patterns of Agent Interaction Scenarios as Use Case Maps" [69]

This paper describes Use Case Maps as "a good candidate for representing scenarios of autonomous agents interacting with other autonomous agents" as they "make a convenient visual aid to analyze, at an abstract level, these agent interactions." This representation relates to video games, largely from an Artificial Intelligence standpoint, which may be interesting to consider within a narrative. However, our approach focuses purely on the narrative structure of a game; thus, while Use Case Maps can be shown to apply to games, they have not been applied to game narrative in our sense.

"Emotional Requirements in Video Games" [70]

This paper "[introduces] emotional terrain maps, emotional intensity maps, and emotion timelines as in-context visual mechanisms for capturing and expressing emotional requirements." Though a new type of non-functional requirement is explored that can arguably relate to video game narrative, there is no acknowledgment of Use Case Maps.

"Towards Conscious-like Behavior in Computer Game Characters" [71]

This paper expresses the desire to create agents (i.e., non-playable characters), which pass the Turing test. Though in-game characters are involved in the story of a game, they do not necessarily belong to the narrative structure; instead these agents better fall into the

field of Artificial Intelligence. In addition, there is no mention of Use Case Maps as a potential solution to the problem.

"Explicit Domain Modelling in Video Games" [72]

This paper discusses the complexity of game design and suggests the use of a component-based architecture over traditional class hierarchies for game entities. Components provide "basic pieces of functionality" allowing for flexible software development, which adapts with game design changes. Entities, "self-contained pieces of logic that can perform different tasks," can be assigned numerous components to specify their behaviour, instead of inheriting classes in traditional Object-Oriented Programming. Components can require additional overhead, but often make the behaviour of entities easier to understand. Use Case Maps and video game narrative are not considered.

"<e-Adventure3D>: An Open Source Authoring Environment for 3D Adventure Games in Education" [73]

This paper introduces an adventure game engine intended for educational purposes and describes their means of handling narrative. Educational and game engine details aside, game narrative is handled with flags:

> Flags can be used to block or unblock certain actions or elements in a moment through the definition of conditions on them. The course of the games can be changed by triggering effects to activate or deactivate flags.

This simplistic narrative solution could be difficult for "people without a programming background" to implement loops, concurrency, or branches. Furthermore, as all elements or actions require knowledge of the current narrative state to make sense, human error leads way to sequence breaking. Use Case Maps are not referred to in this document.

"Feature Interaction Analysis: A Maintenance Perspective" [74]

This paper combines Use Case Maps and Formal Concept Analysis (FCA) to "assist maintainers in identifying feature modification impacts at the requirements level, without the need to examine source code." UCMs describe features while FCA groups and filters feature interactions. Again, Use Case Maps can aid in the feature interaction problem, but there is no reference to video games or narrative.

Additional Papers

The remaining papers of any relevance either describe Use Case Maps in greater detail [75][76][77], Artificial Intelligence [78][79], or other gaming topics [80][81][82][83]. Furthermore, Amyot and Mussbacher's recent survey of User Requirements Notation and Use Case Maps contains no mention of the terms "game," "story," or "narrative," despite their relation to representing sequences [17].

2.5 Summary

With the research presented in this chapter, we believe we have captured the state of the art for sequence breaking in video games. First, we reviewed game narrative, feature interaction, game development, and game testing. Second, we observed five notable, commercial games from between 1992 and 2011 to see how they handled their narrative progression. Third, we researched representation schemes as they relate to narrative in games, and followed with a literature review of Use Case Maps as they relate to video games and sequence breaking. We concluded that while Use Case Maps are appropriate for our solution, they have not been used in such a case.

In reviewing game narrative and feature interaction, we found further proof for our claims from chapter 1. For one, we saw that game narrative does indeed consist of sequences of events. Two, sequence breaking relates to the 'invocation order' type of feature interaction. Three, an architectural / structural approach with Use Case Maps is a potential method for solving such conflicts. Thus, sequence breaking is indeed a feature interaction problem where the intended narrative sequence of a game is not followed.

In reviewing game development and testing, we found that games are more error-prone than traditional 'productivity' software as their interests rest in the overall experience for the player rather than the accuracy of specific features. Despite the efforts of testers, problems will often remain in games, so a run-time solution is most appropriate for the prevention of sequence breaking.

In observing the narrative progression for five notable, commercial video games, we noticed geographic and narrative barriers, constraints based on variables, knowledge of progression, and central hubs. We also saw that narrative barriers, which make use of

constraints, are significantly more difficult to break than geographical barriers, while central hubs are ‘transitions’ between events.

In researching representation schemes, we saw that Petri Nets had been applied to sequence breaking in video games, but never had Use Case Maps been considered, despite their mention as a means to reducing feature interaction.

From our literature review, it can be seen that Use Case Maps have been applied to the Feature Interaction Problem, but not to video games or video game narrative. In the next chapter, we will further motivate our choice of Use Case Maps as a means of representing game narrative to assist in preventing sequence breaking. While Use Case Maps represent causal sequences, which are highly relevant to narrative structure, to the best of our knowledge no one has considered or incorporated these diagrams with the intent of ‘guaranteeing’ a game’s narrative structure.

3 Chapter: An Approach to Preventing Sequence Breaking

In chapter 1, we provided an overview of our approach to preventing sequence breaking in video games. This overview suggested a means of determining if a unique event, given its unique identifier (provided by a game programmer), is considered legal based on the player's current progress within a game's narrative representation. It further described the concept of preloading, which adds elements to a game's world that are necessary to make an event callable, and unloading, which removes such elements. While the latter portion of this approach serves to reduce calls to illegal events, the process of determining whether an event is legal or illegal is the major requirement towards preventing sequence breaking. Throughout this chapter we will detail our solution for determining the current set of legal events while handling the preloading, calling, and unloading of events; in doing so, we will have described our approach to preventing sequence breaking.

This chapter is outlined as follows. Section 1 briefly highlights the features of game narrative that we must represent. Section 2 details Use Case Maps, their relation to features of game narrative, and ultimately our justification in choosing them. Section 3 describes our modification to the traditional form of Use Case Map traversal, in order to determine the current set of legal events. Section 4 details our Narrative Manager, which manages a game's narrative to maintain a set of legal events. Section 5 details our *Royal Pegasi Algorithm*, which explores a Use Case Map to determine the next set of legal events upon calling an event (or at initialization), based on the player's progress. Section 6 details the technical environment, in which our solution was developed. By the end of this chapter, we will have justified our choice of Use Case Maps as a representation for game narrative, explained the intent of our approach, and detailed our solution for the prevention of sequence breaking at run-time.

3.1 Features of Game Narrative

In chapter 2, we reviewed the concept of events, approaches to handling game narrative, and played five games to observe both their narrative structures. From that information, we will compile a list of necessary features to represent a game's narrative.

To begin, recall that literature on game narrative defined events (or cut-scenes) as "individual moments of narrative as the game progresses," [18] that are often "passages

where the player cannot do anything but most simply watch [the story] unfolding" [19]. These events form narrative pieces that ultimately complete a game's story. With events out of their intended order, due to sequence breaking, the story cannot be told as defined by designers; thus, representing sequences of events is a primary requirement.

Then we presented three narrative approaches from literature: String of Pearls, Branching Narrative, and Amusement Park. The String of Pearls approach consisted of a linear path of events with little to no deviation. The Branching Narrative approach consisted of branching paths, which could significantly change the outcome of a game's story. The Amusement Park approach consisted of many sub-plots or optional side quests for the sake of acquiring special rewards. These three narrative approaches can again be represented through a sequence of events, but require additional features such as split paths (for Branching Narrative); and, concurrency (wherein an event from one path can be called, then an event from another or the same path, and so forth), multiple paths, or condition-based 'waiting places' (for Amusement Park). As games do not necessarily use just one of these approaches to their narrative, but rather a combination, it is necessary to include all such features found within each approach [21, p. 29].

Then, we presented the notion of gates, which are geographic barriers requiring a skill or item to proceed; though in a personal e-mail exchange Tyler Moore of Frostfire Games admits that he does not perform checks for legality to prevent sequence breaking. Such barriers are common in games, as was seen in our game observations. To represent this feature we require simple variable types and constraints, which use variables to permit or deny access to a narrative path.

Finally, during our game observations we saw a continuation of the above features, along with narrative barriers, central hubs, loops, and knowledge of progression. *The Legend of Zelda: A Link to the Past* introduced geographic barriers, which created curiosity for players but could be overcome through glitches or designers not handling all cases of progression, and narrative barriers, which forced the player to complete a specific sequence of events. From our playthrough, the narrative of this game was linear with side quests opening up as we progressed; thus, *A Link to the Past* uses a combination of the String of Pearls and Amusement Park methods. *Super Mario 64* made significant use of

variables and constraints to permit or deny access to later levels, including a counter of stars collected and two keys to unlock more parts of the castle. The game also made the concept of a 'central hub' very clear, as the player returns to the castle between levels. Furthermore, we see the use of loops as the player can enter a level, complete it, move around the castle, and then either enter the level again, a new level, or a previous level. *Grand Theft Auto 4* relied almost entirely on narrative barriers to prevent access to future missions, while often allowing one of multiple missions to be started when the player was not currently in one. In this sense, events (or missions) can be called in 'parallel', so a means for describing such concurrency is necessary. *Pokémon SoulSilver Version* and *HeartGold Version* demonstrated further use of geographic and narrative barriers. *Bastion* extended our requirements by suggesting knowledge of the player's progression. While this feature is not strictly necessary as it served more to aid the player rather than dictate the story, we believe the ability to 'query' the player's progress could be of benefit, especially for the sake of resolving a game's state when sequence breaking has occurred. Our gameplay observations emphasized the need for specifying sequences of events, including split, concurrent and looping paths; incorporating variables as well as constraints; making central hubs clear; and, allowing 'queries' on the player's progress.

Referring to *Grand Theft Auto 4*, we postulate that denying access to geography is not necessary as part of our solution. In this game, the player can use cheats to skip geographic barriers, but it is assumed the player can go anywhere at any time. Instead of preventing access to geography, which would be considered unavailable based on the player's progress, the game refuses to load anything of relevance for the player to do in those areas and even punishes the player for entering them. As a result, the player quickly grows bored and opts to continue on with missions in order to enter those areas legally. We choose to include a similar approach to handling geographic barriers by preloading elements of the game world necessary to call legal events and unloading elements of the game world that would assist in calling illegal events, and thus inherently capture this feature outside of our narrative representation.

Based on literature and observations of five games, we can settle on a list of features that are required to capture a reasonable portion of video game narratives. First, we require a means of representing all valid sequences of events that includes splitting paths, running

paths in parallel, looping paths, and allowing multiple paths. Second, we require simple variables and types (e.g., integers for counters, Booleans for switches), and a means of evaluating conditions upon these variables to permit or deny access to narrative paths. Third, we require a means of waiting for a condition to succeed on a narrative path, effectively being blocked until then. Fourth, we can consider central hubs to be the transitions from one event to another, such as an edge in a graph. Additionally, we would like to 'query' the player's progress, but this feature has less to do with a representation and constitutes a more-technical aspect of our solution. Providing we maintain a set of legal events, we can allow programmers to fetch this set. In using a representation scheme, which handles all of these features, we can prevent sequence breaking for events that have been stored within, by verifying an attempted call to an event against the player's progress and a narrative representation.

3.2 Features of Use Case Maps

In both chapters 1 and 2, we suggested Use Case Maps as our representation scheme for game narrative, but we have not fully justified this decision. Throughout this section we will compare features of Use Case Maps to the requirements specified for game narrative, to argue that Use Case Maps are sufficient for our purposes; in addition, we postulate that their specification is easy-to-understand.

To begin, Use Case Maps are a "visual scenario notation, [which] focuses on the causal flow of behaviour optionally superimposed on a structure of components" [17]. One can think of a Use Case Map as a collection of diagrams (e.g. graphs) containing nodes and edges between those nodes. Unlike a graph, however, each node can affect traversal along the diagram based on its type (e.g., an And-Fork node may allow two or more paths to be followed in parallel). Amyot and Mussbacher define elements of the notation:

> A map contains any number of paths and components. Paths express causal sequences and may contain several types of path nodes. Paths start at start points … and end at end points …, which capture triggering and resulting conditions respectively. Responsibilities … describe required actions or steps to fulfill a scenario. OR-forks …, possibly including guarding conditions such as [NotOk], and OR-joins … are used to show alternatives, while AND-forks … and AND-joins … depict concurrency. Loops can be modeled implicitly with OR-joins and OR-forks. As the UCM notation does not impose any nesting constraints, joins and forks may be freely combined and a fork does not need to be followed by a

> join. Waiting places … denote locations on the path where the scenario stops until a condition is satisfied [17].

From this description, we see that Use Case Maps can fulfill several of our requirements for a narrative representation. First, maps contain paths, which are used to express "causal sequences" of responsibilities, satisfying our need to represent valid sequences of events; we equate responsibilities to events. Second, within these paths, Or-Forks can be used to show alternatives (i.e., splitting), while And-Forks increment and And-Joins decrement levels of concurrency. Third, loops can be specified through Or-Forks and Or-Joins. Fourth, variables are suggested as they can be used to handle conditions such as with Or-Forks. Fifth, Waiting Places block progression along a path until a condition has been met. It is evident from this description that Use Case Maps satisfy most of our requirements, but we lack proof of their support for multiple paths, central hubs, and simple variables types.

To verify these remaining requirements are indeed supported by Use Case Maps, we turn to jUCMNav [84], which is a tool that aids in creating such diagrams. (Note that these features are not exclusive to jUCMNav.) In this software, we can build specification diagrams, paths within those diagrams, and all of the node types specified above (along with a few others). Multiple paths can be created within any diagram as figure 3 shows:

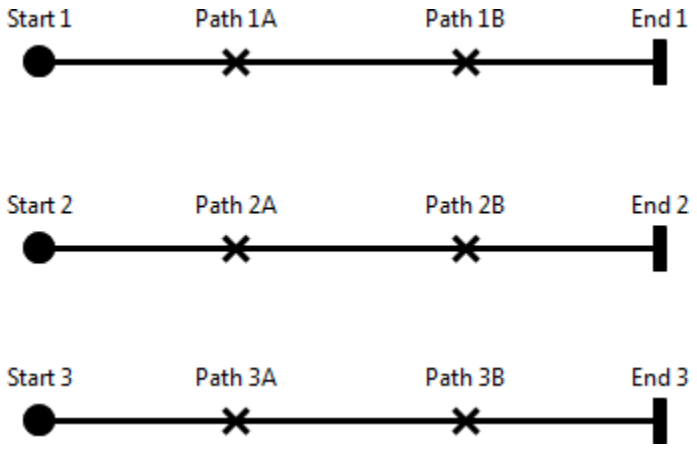


Figure 3 Use Case Maps support multiple disjoint paths

Central hubs simply represent the edges between nodes, as no actions are forced upon the player during those transitions. Simple variable types are shown in figure 4, where the

player collects stars to unlock levels—which are simplified to events—but must also complete “Dire, Dire Docks” before accessing “Bowser in the Fire Sea”:

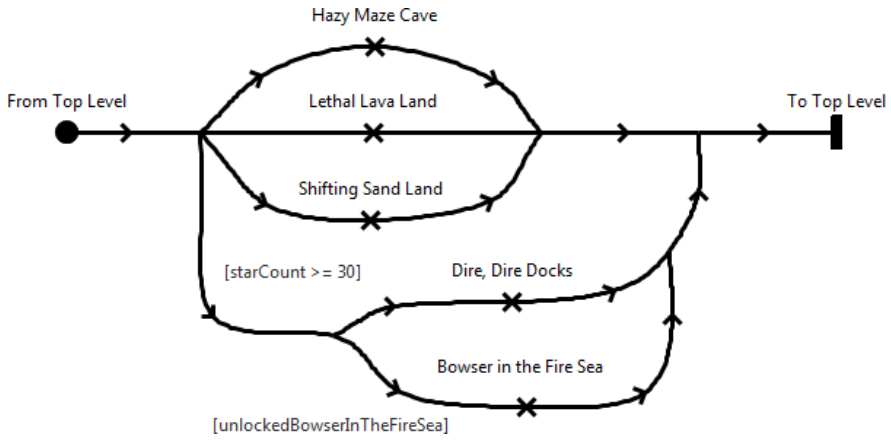


Figure 4 Unlocking "Bowser in the Fire Sea" from within a loop

Note that external code would be assigned to “Dire, Dire Docks” to enabled the switch that allows access to “Bowser in the Fire Sea.” Note further that an overlying loop repeats the process shown (and other processes not shown). Designs can also create their own enumerated sets within jUCMNav, allowing for more data types. Furthermore, jUCMNav lets designers make scenarios, which specify zero or more start points and variable initializations. By examining jUCMNav’s feature set on Use Case Maps, we conclude that this representation scheme is sufficient for describing game narrative.

Beyond the features of Use Case Maps described that relate to game narrative, the specification also contains components, static and dynamic stubs, and timers – of which only static stubs are included in our solution. Components contain responsibilities, and are used to refer to objects or people involved in a process. Amyot states that this feature is “optional”, so we have excluded it [17]. Stubs are nodes that refer to sub-maps in their place. They can assign incoming and outgoing edges to and from the stub node to different start and end points, respectively, within their associated sub-maps. Static stubs refer to one sub-map at most, while dynamic stubs may refer to multiple sub-maps, with one selected based on certain criteria (e.g., variable conditions). Only static stubs have been accounted for in our solution as they help with scalability, since sub-diagrams can be created to represent recurring narrative paths or to provide abstraction for another diagram altogether. This idea is similar to writing functions in a programming language.

Dynamic stubs, which reference diagrams (similar to static stubs) based on conditions, were unnecessary as they were not required to represent game narrative based on our feature list. Timers are similar to Waiting Places in that they wait for a condition to be true, but can also access a timeout path in the event that no other narrative path can be further explored. Such a feature was not considered necessary as per our requirements to represent game narrative; though it could be handled in a future iteration as game designer may see a purpose. Of the remaining features of Use Case Maps, we chose only to handle static stubs as they aid in scalability, while dynamic stubs and timers were not considered necessary for our purposes [17].

As a final motivation for Use Case Maps, we postulate that they have an ease of understanding. For one, it is visually clear how one moves along such a diagram, from one node to another. Two, each node type serves a specific, yet simple behaviour that affects movement along the diagram. Three, stubs allow for scalability and clarity by hiding away unnecessary details at different 'levels' of the entire Use Case Map. Four, responsibilities serve to fulfill obvious objectives, based on their labels. Five, variables are changed through responsibilities and compared on edges, which show labels. In comparison, Petri Nets make use of tokens, which designers need to handle and are arguably less understandable and potentially less usable. While the argument to ease of understanding is subjective, we believe, at least in comparison to Petri Nets, that Use Case Maps qualify for such a claim.

To conclude, we chose Use Case Maps as our representation scheme for game narrative as the specification handles all of our requirements, allows stubs for scalability, and is easy to understand. As diagrams of narrative are likely to be used amongst designers, writers, and programmers, it is necessary that the format be clear to all involved, while still managing to capture the necessary characteristics of a game's storyline.

3.3 Modifying Use Case Map Traversal

Our solution largely consists in modifying the traversal of Use Case Maps to better suit the purpose of preventing sequence breaking in video games. In particular, our focus is on exploring a Use Case Map to find a set of legal events (i.e., accessible responsibilities)

given a player's progress (i.e., a set of locations within diagrams) and an identifier for a legally called event. For further details on Use Case Maps, refer to Appendix A.

We refer to Amyot's path traversal requirements for Use Case Maps [85], interspersing our modifications throughout. For clarity, we comment on one or more requirements after listing them. These requirements have significantly influenced the design of our Narrative Manager and its *Royal Pegasi Algorithm*:

1. Path Traversal shall start at 1 to N parallel scenario start points as defined by the user (scenario-start).
2. Path Traversal shall start with initial values (true, false, or undetermined) for each path data variable as defined by the user (variable-init).
3. Path Traversal shall move from path element A to path element B if
    a. Path Traversal is currently visiting path element A, and
    b. there is a direct connection from A to B (hyperedge-connection[4]), and
    c. the *path continuation condition* of path element A to path element B is fulfilled.
4. The path continuation condition for a *start point* shall be fulfilled if the logical expression for its guard evaluates to true (logical-condition of start).

In the traversal algorithm we propose, we do not handle preconditions on Start Points as such behaviour can be mimicked by adding a Waiting Place after a Start Point.

5. The path continuation condition for *end points* not directly connected to waiting places or timers shall be always fulfilled.

We do not allow Timers and do not allow End Points to be connected to Waiting Places. These features were not deemed necessary based on our narrative requirements.

6. The path continuation condition for a *responsibility* shall be always fulfilled.

When a responsibility is found, we preload its associated event and stop traversal. Only when an event is called, does traversal continue on past it – after unloading the event.

7. The path continuation condition for an *OR-fork* shall be fulfilled if the path continuation condition of exactly one branch of the OR-fork is fulfilled.

We allow Or-Forks to have multiple outgoing connections whose conditions evaluate to true. When more than one crossable connection exists, traversal needs to consider all paths ahead as 'speculative', as the player may visit only one of these paths, though found events along these paths are still preloaded. When a path is determined, through an

[4] A hyperedge-connection simply refers to a connection between a source and target node where the target may not reside in the same specification diagram (i.e., map) as the source, e.g., target is within a stub.

event call, all other paths need to be rejected and their traversal 'undone' by unloading events along those paths.

8. The path continuation condition for a branch of an *OR-fork* shall be fulfilled if the logical expression for the branch evaluates to true (branch-condition of path-branching-characteristic).
9. The path continuation condition for an *OR-join* shall be always fulfilled.
10. The path continuation condition for each branch of an *AND-fork* shall be always fulfilled.
11. The path continuation condition for an *AND-join* shall be fulfilled if Path Traversal is currently visiting the AND-join for all of its incoming paths.

As we are handling 'speculative' paths due to Or-forks, it is necessary to consider if each incoming path of an And-Join is 'speculative' or 'guaranteed'. Once and only once all incoming paths are guaranteed, can we be certain of traversal past the node. If any incoming paths are speculative, then we can only speculate on the path leaving the And-Join. If not all incoming paths have been completed, then the And-Join blocks traversal.

12. The path continuation condition for a *loop* shall be fulfilled if the path continuation condition of exactly one branch is fulfilled (either the loop branch or the exit branch).
13. The path continuation condition for the loop branch shall be fulfilled if the logical expression for the loop exit evaluates to false (exit-condition of loop).
14. The path continuation condition for the exit branch shall be fulfilled if the logical expression for the loop exit evaluates to true (exit-condition of loop).

We treat loops as a combination of Or-Forks and Or-Joins, which can both be handled as described above. It is necessary to speculate on paths leaving an Or-Fork, however, if leaving a loop is optional, as we cannot guarantee if the loop will continue or end until the player chooses an event to terminate the loop, as shown in figure 5:

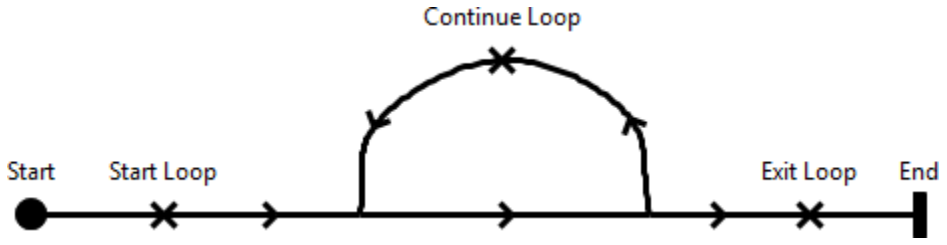


Figure 5 A loop may not be guaranteed to continue or stop

15. The path continuation condition for a *static stub* shall be always fulfilled.

We assume that each incoming and outgoing edge of a static stub is bound to a start and an end point, respectively. We chose not to handle unbound static stubs as we believe designers intend for such nodes to be bound, lest they not serve their purpose.

16. The path continuation condition for a *dynamic stub* shall be fulfilled if the path continuation condition of exactly one plug-in of the dynamic stub is fulfilled.
17. The path continuation condition for a *plug-in* of a dynamic stub shall be fulfilled if the logical expression for the selection policy of the plug-in evaluates to true (branch-condition of plug-in-binding).

Our solution does not support dynamic stubs, as they are not part of our requirements.

18. The path continuation condition for an *end point* and a waiting place connected directly with each other shall be fulfilled if
    a. Path Traversal is currently visiting the end point and the waiting place and
    b. the logical expression for the guard of the waiting place evaluates to true (logical-condition of waiting-place).

Again, an End Point connected to a Waiting Places is a feature not supported, as it is not considered part of our requirements to represent game narrative.

19. The path continuation condition for a *waiting place* shall be fulfilled if the logical expression for its guard evaluates to true (logical-condition of waiting-place).
20. The path continuation condition for an *end point and a timer connected* directly with each other shall be fulfilled if
    a. Path Traversal is currently visiting the end point and the timer and
    b. the path continuation condition for the non-timeout path of the timer is fulfilled.
21. The path continuation condition for a *timer* shall be fulfilled if exactly one of the following cases occurs:
    a. The path continuation condition for the non-timeout path is fulfilled.
    b. The path continuation condition for the timeout path is fulfilled.
22. The path continuation condition for a *non-timeout path* shall be fulfilled if
    a. the timer's timeout variable is set to false (timeout-variable of waiting-place) and
    b. the timer's guard evaluates to true (logical-condition of waiting-place).
23. The path continuation condition for a *timeout path* shall be fulfilled if
    a. the timer's timeout variable is set to true (timeout-variable of waiting-place) and
    b. a timeout path exists for the timer.

We chose to exclude timers as they were not deemed necessary as per our collected requirements of game narrative. While such a feature could be relevant to some game designers, we believe a large portion of games with narrative could do without them. Furthermore, we do not believe accounting for timers in a future iteration of our solution would negatively impact its performance or ability to prevent sequence breaking.

24. The path continuation condition for an *empty point* shall be always fulfilled.

The same rule also applies to direction arrows, which are considered 'empty' nodes that serve the purpose of joining two connections together. Direction arrows are used for visual clarity, and do not affect the representation.

25. Path Traversal shall execute the value assignment statements of a *responsibility* (variable-operation-list) if the path continuation condition for the responsibility is fulfilled.

26. Path Traversal shall execute the value assignment statements of a responsibility in the order defined by the user.
27. Path Traversal shall update the values of the path data variables immediately after executing one value assignment statement.
28. Path Traversal shall evaluate a logical expression to undetermined if any value within the logical expression evaluates to undetermined.

The above rules apply to expressions stored within responsibilities, but we choose not to support these expressions in favour of using our own scripting language instead. The reason for this change is to provide extended functionality to act out scenes, as shall be demonstrated in the next chapter.

29. Path Traversal shall stop if it cannot move to another path element from any of the currently visited path elements.
30. Path Traversal shall regard the values of the path variables at the time path traversal stopped as postconditions of the traversed scenario.
31. Path Traversal shall issue a warning if Path Traversal has stopped, and:
    a. Path Traversal is currently visiting one or more path elements other than end points or
    b. Path Traversal is currently visiting one or more end points connected directly to waiting places or timers or
    c. the postconditions of the traversed scenario do not match the postconditions defined by the user.

In our solution, it is not necessary to issue a warning if path traversal terminates, as we assume that path traversal only terminates at an unbound End Point and is thus not a problem. Instead, we simply mark a 'path- traversing' object for deletion so we may terminate traversal along that path.

It is worth mentioning other Use Case Map traversal algorithms that exist, and briefly specifying how they differ from our approach. The ITU-T Z.151 recommendation of User Requirements Notation provides the most recent (at the time of writing) set of traversal requirements for Use Case Maps in addition to depth- and breadth-first algorithms [86]. We chose not to use the newest set of traversal requirements as the set provided above was sufficient for our needs. Furthermore, this recommendation does not account for traversal along multiple Or-Fork branches. Miga et al. introduced a traversal algorithm, as part of UCM Navigator, that converts Use Case Maps into linear forms, which are "context-free text … descriptions of UCM's that may be used as input to other tools" [87, p. 52]. While this algorithm is depth-first and explores all branches of Or-Forks, similar to our approach, it does not consider such branches to be speculative and does not account for special cases in dealing with speculative paths. Kealey et al. offered an

algorithm for strategy exploration, in introducing jUCMNav, "to determine how well goals in a model are achieved in a given context" [88]. However, this algorithm does not necessarily traverse Use Case Maps directly and also propagates values within a tree-like structure to a 'root' as part of its evaluation; such behaviours are vastly different from our approach, which directly traverses Use Case Maps in a 'forward' manner. While there are similarities between my approach and other algorithms, none of these alternatives, to the best of my knowledge, specifically seek out responsibilities that can be accessed or handle the complexities that arise during traversal when speculative paths are considered, and are thus significantly different from my approach.

As described in the above modifications, a number of changes to the path traversal of Use Case Maps must be made for our particular solution. Primarily, our approach involves searching a Use Case Map for events, from a given set of locations, so they may be preloaded and added to the legal set (of allowable events). Traversal either occurs at initialization or upon an event call, for the sake of repeatedly updating the legal set of events. When an event is removed from the legal set, it is unloaded. Secondarily, throughout our exploration we may find paths without a guarantee, i.e., a speculative path. In the instance of an Or-Fork with multiple legal outgoing edges, we can only speculate on what path the player will take. Once a speculative path has been confirmed, all other speculative paths that could have been taken instead of that one must be dropped by unloading all of their events. Consider figure 6:

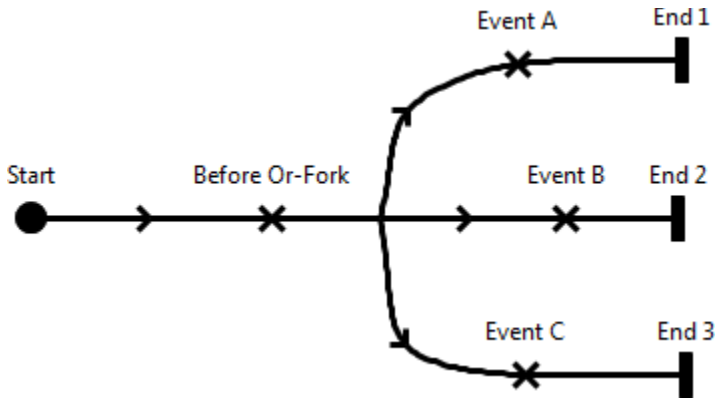


Figure 6 Or-Forks may lead to speculative paths

After completing the event "Before Or-Fork," the player will be presented with three speculative, preloaded events: "Event A," "Event B," and "Event C." When the player

selects one of these events, the other events must be unloaded so they cannot be called. For instance, if the player selects “Event B,” then “Event A” and “Event C” would no longer be callable and thus unloaded; traversal would continue down the path containing “Event B.” Third, And-Joins require special care to account for our approach to speculative paths. These changes to the typical rules of Use Case Maps significantly shape our solution, and are necessary to prevent sequence breaking.

3.4 Narrative Manager

The major contribution of this thesis is the creation of a Narrative Manager that stores a narrative representation, updates narrative progress, and checks the legality of attempted event calls, in order to prevent sequence breaking in video games. Within this Narrative Manager, we store a narrative representation consisting of one or more specification diagrams, which each contain nodes and connections, as per the UCM file format provided by jUCMNav; events, which can be called to execute an associated script and update narrative progress; and, Royal Guards, which serve as markers of narrative progress. Royal Guards may additionally contain children Pegasi, which serve as ‘speculative’ markers of potential nodes or events, which could be accessed but have no guarantee, and may in themselves contain children Pegasi to create a tree structure. We then allow attempts at calling arbitrary events from code external to the Narrative Manager. When an event is considered legal, based on its existence within a set of legal events, that event is called, the narrative progress updated accordingly, and a ‘true’ result returned. The update process involves the traversal of Royal Guards and their ‘sub-Pegasi’ through the narrative representation using a technique we refer to as the *Royal Pegasi Algorithm*, due to its use of Royal Guards and Pegasi as ‘markers’ to indicate progress and explore. When an event is considered illegal, by not existing within the set of legal events, it is not called, and a ‘false’ result is returned, so sequence breaking may be handled externally as per the designer’s desire. It is possible to fetch the current set of legal events, if this information is helpful in correcting the game’s state upon detection of sequence breaking. The Narrative Manager is ultimately a singleton object that stores a narrative representation, monitors narrative progress, and allows arbitrary events to be called provided their legality checks succeed.

3.4.1 Illustrating the Algorithm

To illustrate the complexity of the task involved in and of validating our Narrative Manager and its *Royal Pegasi Algorithm* we will walk through the intended outcomes of numerous event calls on a small, but difficult example as seen in figure 7. Within this scenario, we will see an And-Fork, an Or-Fork with multiple legal paths, a loop, and an And-Join. For each legal event call, our goal is to correctly determine the new set of legal events, given the player's current progress and the identifier of the called event. Any events that are added to the set are preloaded, while any events removed from the set are unloaded. With knowledge of the current set of legal events, sequence breaking can be prevented by only allowing calls to events within that set and rejecting all others.

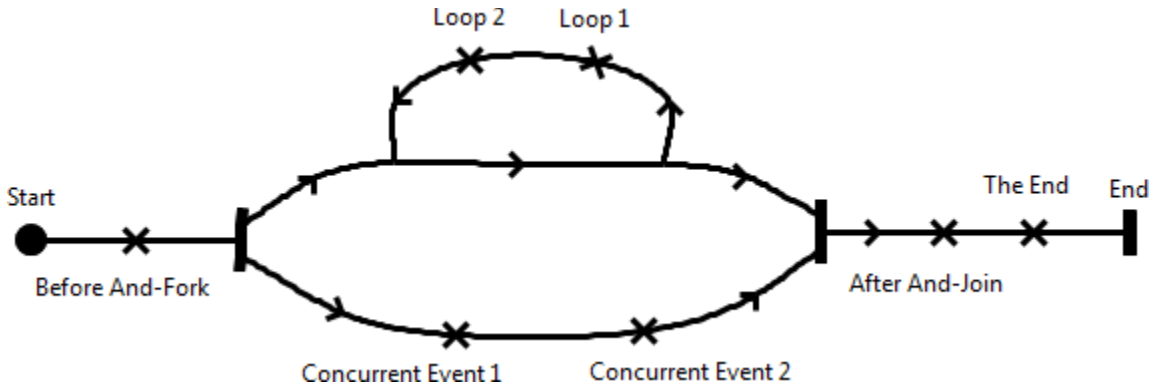


Figure 7 An illustrative example representing concurrency with a loop

Suppose we begin traversal at the single Start Point. At initialization, we preload the first and only legal event: "Before And-Fork". If the player attempts to call any event other than "Before And-Fork" such calls should be rejected. Only once the player calls the event "Before And-Fork," should it be unloaded, and should traversal proceed to find the next set of legal events, preloading them.

```
At Initialization, Legal Event Set = {Before And-Fork}

Call Event: Before And-Fork

Legal Event Set = {Loop 1, Concurrent Event 1}
```

Due to the And-Fork, we can now follow two narrative paths in parallel, granting access to "Loop 1" and "Concurrent Event 1". Though the Or-Fork connecting to "Loop 1"

allows for both of its connections to be crossed, as there are no restrictions, we cannot access the event after the And-Join yet as not all of its incoming paths can be completed.

If we call event “Loop 1,” then the event “Loop 2” will become legal instead. Calling “Loop 2” after “Loop 1” will return us to the above legal set of events, as follows:

```
Call Event: Loop 1

Legal Event Set = {Concurrent Event 1, Loop 2}

Call Event: Loop 2

Legal Event Set = {Concurrent Event 1, Loop 1}
```

Suppose at this point in time, the player chooses to call “Concurrent Event 1,” lest the loop repeat indefinitely:

```
Call Event: Concurrent Event 1

Legal Event Set = {Loop 1, Concurrent Event 2}
```

We now reach an interesting scenario with a few cases to consider. First, the player could repeat the entire loop indefinitely, again returning to the current legal event set. Second, the player could start the loop again, but call “Concurrent Event 2” right after “Loop 1,” resulting in the following outcomes:

```
Call Event: Loop 1

Legal Event Set = {Concurrent Event 2, Loop 2}

Call Event: Concurrent Event 2

Legal Event Set = {Loop 2}
```

Continuing with this case, the player can only call event “Loop 2”, giving:

```
Call Event: Loop 2

Legal Event Set = {Loop 1, After And-Join}
```

This end result is similar to our last case. Third, calling “Concurrent Event 2” instead of “Loop 1” gives the following outcome:

```
Call Event: Concurrent Event 2
```

```
Legal Event Set = {Loop 1, After And-Join}
```

At this point, the player has the choice of repeating the loop indefinitely, with the form:

```
Call Event: Loop 1

Legal Event Set = {Loop 2}

Call Event: Loop 2

Legal Event Set = {Loop 1, After And-Join}
```

Since the player chooses to continue the loop, they cannot break out of it, proceeding to event "After And-Join" until they return to the Or-Fork. While this behaviour is expected for an Or-Fork and a loop, it should be noted that the event "After And-Join" is, as the name suggests, located after an And-Join, which has a strict requirement that all paths before it be possible to complete. Understanding when an And-Join can and cannot be crossed along with our ability to "speculate" on paths the player might take presents a large challenge for our solution, for which there is no trivial answer.

Finally, once the player chooses to leave the loop and pass the And-Join by calling the event "After And-Join," they will gain access to the last event: "The End."

```
Call Event: After And-Join

Legal Event Set = {The End}

Call Event: The End

Legal Event Set = {}
```

At this point, traversal of the Use Case Map has ended and no more legal events remain. Presumably, the game has come to an end as there is no more narrative to manage.

In creating a Use Case Map to represent game narrative, it is important to understand the legal sets of events that will be possible for every legal situation. The Narrative Manager and its traversal algorithm can only generate its legal sets based on the representation scheme given to it, along with the player's current progress and a legal event identifier. We cannot be expected to produce results to match the designer's intentions, if these intentions are not properly presented. As a result, the designer of a Use Case Map must

verify their expected legal sets of events for sequences against the actual legal sets of events for sequences that our solution generates. Since the number of legal sequences may be intractable, we can only verify test cases that provide coverage of our solution along with random paths taken in specific narrative representations. In chapter 4, we provide a tool to assist in validating our solution.

While our solution may involve Use Case Maps to represent game narrative, determining the set of legal events with each legally called event is a complex task that involves validation through test cases for specific requirements presented in our solution and random paths within a designer's representation.

3.4.2 Using the Narrative Manager

The Narrative Manager can be used externally through its three public functions, which serve the purpose of initializing the manager and calling arbitrary events. The `init()` method readies the Narrative Manager by loading information from a Use Case Map file, preparing a scenario, and optionally assigning scripts to events from an XML file. The `tryToCallEventById()` and `tryToCallEventByName()` methods call events given their unique index within the UCM (i.e., the index assigned to a responsibility associated with the event) or the label applied to a UCM responsibility, which again refers to an event, respectively. These methods are further detailed below, with the intent of providing an explanation as to how the Narrative Manager is triggered.

The `init()` method takes in two required and two optional arguments, and then performs six major steps to set up the Narrative Manager. The required arguments—`jucmFile` and `scenarioName`—refer to a .jucm file, which stores the narrative representation, and the default scenario, which contains variable initializations and starting points, respectively. The optional arguments—`eventsFile` and `settings`—refer to an XML file containing scripts for some or all events, and provide additional settings, respectively. The first step in this method is to read in the necessary information from the given Use Case Map (.jucm) file. The second step is to store designer-defined enumerated sets and variables with their initial values, based on the given scenario. Third, all responsibilities are stored as Event objects. Fourth, each specification diagram of the Use Case Map is stored as a SpecDiagram object, which contains Node objects for its

nodes and Connection objects for its connections. Fifth, Royal Guards are placed at the start nodes specified in the given scenario, and begin traversal immediately to determine the first set of legal events. Finally, the XML file containing event scripts is read in, and those scripts are passed along to their associated events. With these six major steps complete, the Narrative Manager will have preloaded and determined the first set of legal events, in addition to storing the narrative representation for future use.

The `tryToCallEventById()` method takes one argument, `eventId`, which refers to the unique index of a responsibility within a Use Case Map file, and attempts to call the event to which that responsibility corresponds. If the given index is invalid (i.e., negative or out of range) or if the associated event is considered illegal, the function does not call the event and instead simply returns the value 'false'. When the associated event is legal, however, the event is called; the Narrative Manager updates its progress to determine the new set of legal events, by moving one or more Royal Guards, which have access to the event; and, the value 'true' is returned.

The `tryToCallEventByName()` method takes on argument, `eventName`, which refers to an event by its unique name instead of its unique index. The method is a simple wrapper for `tryToCallEventById()` in that it looks up the event index from the given name, and then calls that function, returning its results. If the event name is not associated with an event, then the value 'false' is returned instead. This method is primarily used for designers who may not know an event's id, but know its case-sensitive name.

From these methods, it can be seen that traversal only occurs when the Narrative Manager is first launched (i.e., when a UCM is loaded) and after a legal event is called. The initial traversal is an exception to load the first set of legal events, while all other occasions update the narrative progress upon the completion of the called event's script, to determine the new set of legal events.

3.4.3 Nodes and Connections

The narrative representation stored within a specification diagram of a Use Case Map can be considered a list of nodes and a list of connections between them. Unlike a graph, however, each node can be of a different type, which alters the traversal of the diagram from that point forward. For instance, an And-Fork splits the representation into multiple

‘concurrent’ narrative paths. In this section, we describe our approach to handling ‘generic’ nodes, specific types of nodes, and connections.

Using an object-oriented approach, we begin with a generic (or base) Node class, from which all other node classes are derived. This generic node class contains a constructor, a set of incoming connection references, a set of outgoing connection references, and a function to get the next set of legal of connection references leaving the node object. (For our purposes, a ‘connection reference’ is a means of referring to a connection stored within a specification diagram.) The function evaluates each outgoing connection’s condition expression, and then returns all of the outgoing connections that can be crossed. An empty array of connections may also be returned, if no edges can be crossed or there are no outgoing connections. This generic node class is sufficient for representing Empty Points, Direction Arrows, Start Points, Or-Joins, and Waiting Places; however, Or-Forks, which also follow very similar behaviour, require distinction, by performing a check through *instanceof*, within the traversal algorithm as they are treated as a special case.

Beyond the generic node class, several node types have their own derivations.

Responsibility Reference nodes actually refer to events through an index, but the name is taken from the UCM file format, which uses the term ‘responsibility’ instead. When traversal reaches such a node, the associated event is preloaded; when traversal leaves such a node, the associated event is unloaded. (For further details, see section *3.4.4 Events*.) Traversal can only pass a Responsibility Reference node when the associated event has marked itself as passable: a temporary occurrence that happens upon the completion of the event’s script.

Stub nodes additionally contain in- and out-bindings, a function to get a start point within a specification diagram given an incoming connection reference, and a function to get an out-binding given a valid index. Stub nodes serve the purpose of joining one specification diagram to another, by using in-bindings to connect an incoming connection reference to a start point node of another specification diagram and out-bindings to connect an end point node to an outgoing connection reference of another specification diagram.

End Point nodes additionally store references to out-bindings of zero or more stub nodes and contain a function to get a connection reference, leaving a stub node, given a stub

node reference. This extra information and extended behaviour allows the traversal algorithm to either terminate (i.e., marking the Royal Guard or Pegasus for deletion) or move past an end point, by crossing a connection leaving a stub node.

And-Fork nodes require a class simply to perform an *instanceof* check, as their behaviour requires that the traversal algorithm allow all outgoing paths to be followed in parallel.

And-Join nodes serve the purpose of reducing a layer of concurrency, by merging completed concurrent paths into a single path once all incoming connections have been crossed. To handle this behaviour, we extend the generic Node class to include an associative array of attendees (i.e., Royal Guards or Pegasi located at the node); an associative array of connections to cover, with a count of unique attendees who have crossed; a counter of satisfied connections; a counter of satisfied connections with Royal Guards; two functions to add and remove unique attendees, while updating necessary variables; two functions to destroy attendees; a function to check if all incoming connections are covered by Royal Guards; and, finally a check before returning the single outgoing connection. Recording attendees who arrive at or depart from the And-Join allows us to check if all incoming connections have been satisfied. The additional counters are added as an optimization to offer O(1) time complexity in adding or removing unique attendees and verifying if the passing condition has been met. As a primary feature of the And-Join is to reduce several paths into one it is necessary to destroy attendees so only one 'survivor' may proceed along the outgoing connection. The first such method, `destroyAllWaitingRoyalGuards()`, is used when Royal Guards cover all incoming connections and a Royal Guard wishes to pass the And-Join. In this case, we destroy all Royal Guards *after* one 'survivor' has already left. The second such method, `destroyAbsolutelyAllWaiting()`, is used when a Royal Guard follows a Pegasus past the And-Join. In this case it is necessary to destroy all Royal Guards either attending the And-Join directly or who have at least one sub-Pegasus attending the And-Join, effectively destroying their entire Pegasi trees as well; again, it is assumed that the 'survivor' has already left. As the ability to pass an And-Join depends on its current attendees and if a Royal Guard or Pegasus wishes to pass, such nodes require significant extension to handle these requirements.

With the behaviour of all node types created, it is then necessary to join nodes together through Connection objects. Connections have source and target node references, which refer to a node within a specification diagram, along with an optional condition, which must be true before the connection can be crossed. Combining nodes with connections, in a similar manner to a graph, completes the structure of our narrative representation.

3.4.4 Events

Events serve the purpose of furthering the story within a game through uninterruptable scripts when called; are preloaded to set up the game world so they may be called; and, are unloaded to remove elements of the game world to prevent them from being called. In addition, events store a list of attendees, similar to And-Join nodes, in order to determine if they are legal to call, and a flag to indicate if the event can be passed during traversal. Responsibilities from Use Case Maps directly refer to events and are used to create them, with the responsibility label becoming the event name; however, their expressions are completely ignored. Upon calling an event, exactly one or exactly all of its attendees are moved forward, based on a setting provided to the Narrative Manager at its initialization, with all Royal Guards traversing immediately after.

Using the `preloadMaybe()` method, which takes in an attendee (i.e., Royal Guard or Pegasus), on an event, the attendee is stored within the event's list of unique attendees, and the game world is modified so that the event may be called – if no other attendees were previously available. Note that the actual preloading is handled externally by game programmers, while our solution merely calls such a function by passing along the name of the event to preload. When an event has at least one attendee, it is considered legal.

Using the `unloadMaybe()` method, which also takes in an attendee, on an event, the attendee is removed from the event's list of unique attendees, and the game world is modified so the event may not be called – on the condition that no more attendees remain. Note again that the actual unloading is handled externally, in a similar means as preloading. When an event has no attendees, it is considered illegal.

For our purposes, we repeat, we ignore expressions assigned to responsibilities in favour of scripts written for our own scripting language. The reason for this change is that our scripting language performs actions, which would be better suited for progressing a

game's narrative, that are beyond the ability of UCM's responsibility expressions. For instance, the designer may display messages to the screen. Scripts are instead read from an XML file, and then assigned to events.

Ultimately, the major work of the Narrative Manager occurs when an event is called through its `call()` method, as the set of legal events needs to be updated. This method begins by running the associated script, which may be empty, through the Script Manager[5]. Upon completion, a callback function fires that first selects attendees to move, as specified within the 'onEventCall' setting assigned to the Narrative Manager (i.e., MOVE_FIRST, MOVE_LAST, MOVE_RANDOM, MOVE_ALL); makes the event briefly passable; moves the selected attendee(s) past the event by invoking their `onEventCall()` method; makes the event no longer passable; and then, moves every Royal Guard by invoking its `trot()` method. After all Royal Guards have moved, including those at the event or associated with a Pegasus at the event, any Royal Guards that have been flagged for deletion during their traversal are immediately destroyed. Thus, when an event is called, the Narrative Manager is able to update accordingly.

3.4.5 Royal Guards and Pegasi

Royal Guards and Pegasi can be thought of as markers to indicate positions along a Use Case Map representation of a game's narrative. A Royal Guard only traverses the Use Case Map as far as it can go with absolute certainty. For instance, if a Royal Guard reaches an Or-Fork with multiple legal outgoing connections, it must stop, as it does not know with certainty which path to take. Pegasi can be considered 'speculative' markers in that they explore ahead along all available paths. Both Royal Guards and Pegasi can have 'sub-Pegasi,' creating a tree structure; in addition, each has a unique id to identify it. The main goals of Royal Guards and Pegasi are to 1) locate possible events, so they may be preloaded, effectively becoming legal, and 2) unload events upon leaving their associated nodes, effectively making them illegal.

From a technical perspective, Royal Guards and Pegasi have been merged into the same class—`RoyalGuardOrPegasus`—which maintains a reference to the current node, a stack of stub node references (for entering and leaving stubs), a reference to the last

[5] Details on the Script Manager and its scripting language are beyond the scope of this thesis.

connection crossed, a parent instance, a list of children Pegasi, and a unique id. Royal Guards include an additional flag for deletion, while Pegasi include an additional list of connections crossed since their creation. The reason for merging these types of objects into one class is that both instances act very similarly, with some minor exceptions; we also can determine if an instance is a Pegasus, by checking if its 'parent' property is null.

It should be noted that when a Royal Guard or Pegasus is destroyed, its entire Pegasi sub-tree is also destroyed. Furthermore, if the instance was located at a Responsibility Reference node or an And-Join node, then its current node is notified of its removal.

Royal Guards and Pegasi move on two occasions. When the Narrative Manager is first launched, a Royal Guard is assigned to each Start Point specified in the scenario. These Royal Guards call their `trot()` methods to traverse along the Use Case Map in order to find the initial set of legal events. At certain nodes, such as Or-Forks, sub-Pegasi are made to explore further ahead, using their `flyAhead()` method; notably, additional Royal Guards are created at And-Forks for concurrency. In these cases, the new instances immediately begin traversal upon their creation. Both of the above methods serve as wrappers for the common `traverse()` function, and are provided for 'metaphorical clarity,' although `trot()` also destroys any sub-Pegasi first. When an event is called, the common `onEventCall()` method, for selected Royal Guards or Pegasi attending the event, attempts to unload the event associated with the instance's current node, and then either invokes `followPegasus()` if the instance is a Pegasus or simply moves the instance past the event node if it is as Royal Guard. The `followPegasus()` method concatenates connections crossed from a leaf Pegasus to a root Royal Guard, so the Royal Guard may move to the intended location, and then past the event node. Along the way, special cases for And-Forks, And-Joins, Stubs, and End Points are handled. Immediately after an event has moved one or all of its attendees, all Royal Guards call their `trot()` methods to explore the narrative representation even further. To understand the behaviour of the aforementioned methods, we move into the *Royal Pegasi Algorithm*.

3.5 Royal Pegasi Algorithm

The *Royal Pegasi Algorithm* refers to three major functions, which enable Royal Guards and Pegasi to traverse a Use Case Map to find sets of legal events. When the Narrative

Manager is first launched, Royal Guards are placed at Start Points specified in a given scenario of a given Use Case Map. These Royal Guards initially traverse the narrative representation, creating sub-Pegasi as necessary, to find and preload the first set of legal events. As a game's narrative progresses, specific events are called, which, if legal, update the narrative progress to find the next set of legal events, preload those events, and unload events that have become illegal. This process continues either indefinitely, until the end of the game is reached (as per the designer's specifications), or until the set of legal events becomes empty. Throughout this section we will provide pseudo code for three functions—`traverse()`, `onEventCall()`, and `followPegasus()`—within the `RoyalGuardOrPegasus` class that handle our traversal approach, interspersed with explanations of our logic, and beginning with a few additional, necessary details.

To start, consider the constructor of the `RoyalGuardOrPegasus` class and a few of its helper methods. Note that '#' symbols are used to indicate comments, while 'this' refers to the Royal Guard or Pegasus object being acted upon. The term 'instance' in the descriptive text ahead refers to 'this' Royal Guard or Pegasus, and is used for brevity.

```
class RoyalGuardOrPegasus:
   function init(initialNodeRef, stubNodeRefStack, parent):
      this.currentNodeRef = initialNodeRef; # indicates location
      this.parent = parent; # defaults to null for Royal Guards
      this.uniqueId = getNextUniqueIdForRoyalGuardsAndPegasi();
         # the uniqueId is used by external objects for reference
      this.pegasi = []; # sub-Pegasi, i.e., children within a tree
      this.connectionRefsTakenByPegasus = []; # only used by Pegasi
      this.lastConnectionRef = null; # no connections crossed
      this.stubNodeRefStack = stubNodeRefStack; # deep-copy stack
      this.deleteMe = false; # flag for deletion

   function isPegasus():
      return (this.parent != null);

   function destroy():
      var currentNode = getNodeByRef(this.currentNodeRef);

      # Inform some nodes of removal
      if (currentNode instanceof RespRefNode)
         currentNode.unloadEvent(this);
      else if (currentNode instanceof AndJoinNode)
         currentNode.removeUniqueAttendee(this);

      this.destroyAllSubPegasi();
```

Due to its repeated usage, we define a function to follow a connection reference. This function looks up a connection given its reference, and then, if that connection does indeed exist, moves the Royal Guard or Pegasus across that connection by updating the current node reference to that of the connection's target; we store the last connection reference as well, as it is needed in some cases such as stubs.

```
function followConnectionRef(connectionRef):
   var legalConnection = getConnectionByRef(connectionRef);

   if (legalConnection != null):
      this.currentNodeRef = legalConnection.getTargetNodeRef();
      this.lastConnectionRef = connectionRef;
```

3.5.1 Traverse

Moving forward, the recursive `traverse()` method, which is called through both `trot()` for Royal Guards and `flyAhead()` for Pegasi, transitions an instance through a narrative representation, creating sub-Pegasi as necessary, for the purpose of finding and preloading legal events. The `trot()` method additionally destroys all sub-Pegasi before calling `traverse()`, while `flyAhead()` simply calls `traverse()`.

This function begins with an extra loop check for Pegasi, by comparing the most-recently crossed connection reference against all previous connection references. Provided the Pegasi does not get stuck in a loop, it records the newest connection. We acknowledge that this feature has not been tested thoroughly and that it is added as a safety measure.

```
function traverse():
   if (this.isPegasus() && this.lastConnectionRef != null):
      for each (connRef in this.connectionRefsTakenByPegasus):
         if (connRef == this.lastConnectionRef):
            return;
      this.connectionRefsTakenByPegasus.push(this.lastConnectionRef);
```

Afterward, the current node is fetched by looking up the current node reference. Should the node be null, the operation terminates. Otherwise, the algorithm moves onto handling specific types of nodes.

```
   var currentNode = getNodeByRef(this.currentNodeRef);

   if (currentNode == null):
      this.deleteMe = true;
      return;
```

In the case of Stub nodes, the correct start point of the associated sub-diagram can be determined based on the incoming connection reference, which is the connection crossed to reach the stub. For this reason, it is necessary to record the reference of the last connection crossed. It is important to note that when entering a Stub node, a reference to that node is stored for future use. One can think of this approach as a stack of function calls, where a parser needs to know where in the source code to return once a function terminates. It is assumed that all incoming edges of a Stub node are bound to Start Points. This approach only handles the arrival at a Stub node as End Points handle the departure.

```
if (currentNode instanceof StubNode):
   this.stubNodeRefStack.push(this.currentNodeRef);
   this.currentNodeRef = currentNode.
      getStartPointNodeRefByConnectionRef(this.lastConnectionRef);
   this.traverse();
```

In the case of End Point nodes, an instance can either exit a stub, returning to a path leaving a stub node, or end traversal entirely. With the former, the correct connection to take depends on the stub node that was entered. For example, two stubs referencing the same sub-map may appear in different paths, so it is necessary to know which stub node was taken to enter the sub-map. When a stub node is left, it is popped from the stack of stub node references – similar to a call stack. Alternatively, if no connection can be found then the path is a dead end and the Royal Guard or Pegasus can simply be deleted.

```
else if (currentNode instanceof EndPointNode):
   # Is there a stub node to return to? Otherwise, time to be deleted
   if (this.stubNodeRefStack.length > 0)
      var topStubNodeRef = this.stubNodeRefStack.pop();
      var stubExitConnectionRef = currentNode.
         getStubExitConnectionRefFromStubNodeRef(topStubNodeRef);

      if (stubExitConnectionRef != null):
         this.followConnectionRef(stubExitConnectionRef);
         this.traverse();
      else:
         this.deleteMe = true;
   else:
      this.deleteMe = true;
```

In the case of Responsibility Reference nodes, which actually refer to events, traversal ends, as an event has been found and can be preloaded. By passing the instance as an argument, to be stored as an attendee, its `onEventCall()` method may later be invoked.

```
else if (currentNode instanceof RespRefNode):
   currentNode.preloadEvent(this);
```

In the case of Or-Fork nodes, it is necessary to create sub-Pegasi to explore each available path, unless exactly one outgoing connection is crossable. As there is no guarantee on which path the player will commit to, we can only speculate. When the player later commits to a path, we can remove events from the alternatives, which were not taken. In creating a tree of sub-Pegasi, paths can be concatenated for future use.

```
else if (currentNode instanceof OrForkNode):
   var legalConnectionRefs = currentNode.getNextConnectionRefs();

   if (legalConnectionRefs.length == 1):
      this.followConnectionRef(legalConnectionRefs[0]);
      this.traverse();
   else: # With no connections, the for-loop below will do nothing
      for each (connRef in legalConnectionRefs):
         var pegasus = new RoyalGuardOrPegasus(this.currentNodeRef,
            this.stubNodeRefStack, this);
         this.pegasi.push();
         pegasus.followConnectionRef(connRef);
         pegasus.flyAhead();
```

In the case of And-Fork nodes, there are two situations to consider. With Pegasi, it is simpler to create sub-Pegasi as the paths ahead are entirely speculative and having a common parent helps with following paths later on. With Royal Guards, the first path can be taken by the instance, while every other path can be explored by a newly created Royal Guard. The reason for this change is that all paths leaving the And-Fork when a Royal Guard has reached it can be explored with a guarantee – but Pegasi are speculative.

```
else if (currentNode instanceof AndForkNode):
   var legalConnectionRefs = currentNode.getNextConnectionRefs();
   if (this.isPegasus()):
      for each (connRef in legalConnectionRefs):
         var pegasus = new RoyalGuardOrPegasus(this.currentNodeRef,
            this.stubNodeRefStack, this);
         this.pegasi.push();
         pegasus.followConnectionRef(connRef);
         pegasus.flyAhead();
   else:
      for each (connRef in legalConnectionRefs):
         var royalGuard = createRoyalGuard(this.currentNodeRef,
            this.stubNodeRefStack);
         royalGuard.followConnectionRef(connRef);
         royalGuard.flyAhead();
```

In the case of And-Join nodes, when all incoming connections have been covered there are three situations to handle. The first situation is when a Pegasus reaches the And-Join. Here, the Pegasus can pass without restriction, though a sub-Pegasus is created due to technical reasons, as it is necessary to maintain a Pegasus at an And-Join in the event another Royal Guard or Pegasus from another incoming path crosses the And-Join as well. Failure to do so may allow a Royal Guard to move past the And-Join later on, effectively letting the outgoing path be taken multiple times, which is clearly not the intent. The second situation occurs when a Royal Guard is attempting to cross the And-Join, but some incoming connections are covered only by Pegasi. Since not all paths can be guaranteed to have come to an end, we can only explore a speculative path ahead. The first and second cases perform the same code and have been merged for this reason. The third situation occurs when a Royal Guard reaches the And-Join and all incoming edges have been covered by Royal Guards. In this occurrence, the Royal Guard can simply walk past the And-Join, provided it destroys all other Royal Guards after doing so. To maintain 'attendees' of the And-Join for the evaluation of the passing condition, including that of the third case, we use the functions `addUniqueAttendee()` and `removeUniqueAttendee()`, which both take the instance as an argument. When all incoming connections have not been covered, the And-Join node cannot be passed.

```
else if (currentNode instanceof AndJoinNode):
   currentNode.addUniqueAttendee(this);
   var legalConnectionRefs = currentNode.getNextConnectionRefs();
   if (legalConnectionRefs.length == 1):
      # Case 1: Pegasi can always pass
      # Case 2: 'this' = Royal Guard, but some Pegasi are at the And-Join
      if (this.isPegasus() || !currentNode.isPassableByRoyalGuard()):
         var pegasus = new RoyalGuardOrPegasus(this.currentNodeRef,
            this.stubNodeRefStack, this);
         this.pegasi.push();
         pegasus.followConnectionRef(legalConnectionRefs[0]);
         pegasus.flyAhead();
      # Case 3: Royal Guards are on all incoming connections
      else:
         currentNode.removeUniqueAttendee(this);
         this.followConnectionRef(legalConnectionRefs[0]);
         currentNode.destroyAllWaitingRoyalGuards();
         this.traverse();
```

In the case of all other nodes, we can simply query the next set of legal connection references, and, if exactly once connection is found, cross it to move forward. Due to the

design of the generic Node class, this final case can support Start Points, Empty Points, Direction Arrows, Or-Joins, and Waiting Places.

```
else:
   var legalConnectionRefs = currentNode.getNextConnectionRefs();
   if (legalConnectionRefs.length == 1):
      pegasus.followConnectionRef(legalConnectionRefs[0]);
      this.traverse();
```

While the `traverse()` function plays a significant role in preloading events, it does not handle their unloading or the need for Royal Guards to catch up with leaf Pegasi.

3.5.2 On Event Call and Follow Pegasus

When an event has been called legally and a Royal Guard or Pegasus attending that event has been selected to move, the attendee's `onEventCall()` method is invoked. The first task of this function is to remove the attendee, potentially unloading the event. Afterward, if the attendee is a Pegasus, then the root Royal Guard of its 'Pegasus true' needs to follow the path created to reach the node associated with the event. This additional traversal occurs through the `followPegasus()` method, which is called upon the attending Pegasus and up the tree until the root Royal Guard can call it to follow the path. Royal Guards can simply walk past the event, by looking up the valid connection, of which one is assumed. When the `onEventCall()` method terminates, all Royal Guards traverse the narrative representation yet again to find the new set of legal events.

```
function onEventCall():
   # It is assumed 'this' is at an event, lest the function not be called
   var currentNode = getNodeByRef(this.currentNodeRef);
   currentNode.unloadEvent(this);
   if (this.isPegasus()):
      this.parent.followPegasus(this.connectionRefsTakenByPegasus);
   else:
      # It is safe to assume that an event has an outgoing connection
      var legalConnectionRefs = currentNode.getNextConnectionRefs();
      this.followConnectionRef(legalConnectionRefs[0]);
```

The `followPegasus()` method recursively moves up a Pegasi tree starting with a leaf Pegasus, while concatenating paths, and then moves the root Royal Guard along the final path to reach an event. Pegasi found at And-Joins must remove themselves as attendees, to avoid deleting their Royal Guard later on. As the Royal Guard moves along the path, it needs to handle a few special cases for And-Forks, And-Joins, Stubs, and End Points.

```
function followPegasus(pathToTake):
   # It is assumed that the last node is an event, i.e., RespRefNode
   # Move up the tree until we reach a root Royal Guard
   if (this.isPegasus()):
      # Any Pegasi at And-Joins need to remove themselves
      var currentNode = getNodeByRef(this.currentNodeRef);
      if (currentNode instanceof AndJoinNode):
         currentNode.removeUniqueAttendee(this);

      # Move up the tree, concatenating paths
      this.parent.followPegasus(
         this.connectionRefsTakenByPegasus + pathToTake);
      return;

   # The Royal Guard needs to walk the path given
   for each (connRef in pathToTake):
      var currentNode = getNodeByRef(this.currentNodeRef);
```

When an And-Fork node is reached, more Royal Guards must be created for the additional paths not taken by this Royal Guard.

```
      # Handle special cases
      if (currentNode instanceof AndForkNode):
         var legalConnectionRefs = currentNode.getNextConnectionRefs();
         for each (legalConnRef in legalConnectionRefs):
            # Do not create a Royal Guard for the path taken by 'this' one
            if (legalConnRef != connRef):
               var royalGuard = createRoyalGuard(this.currentNodeRef,
                  this.stubNodeRefStack);
               royalGuard.followConnectionRef(legalConnRef);
```

When an And-Join node is reached, it is not sufficient to just delete the attendees at the node, but rather the entire trees of each attendee must be destroyed, including the root Royal Guard. This feature is very important and also dangerous, as an arbitrary Pegasus can be used to delete a Royal Guard and many other Pegasi. It is necessary to perform this feature however, as other Royal Guards could be allowed to pass the And-Join in the future, effectively voiding the intent of merging multiple paths into one to reduce a layer of concurrency. The current Royal Guard instance is also removed as a precaution, regardless of its appearance within the list of attendees.

```
      else if (currentNode instanceof AndJoinNode):
         currentNode.removeUniqueAttendee(this);
         currentNode.destroyAbsolutelyAllWaiting();
```

When a Stub node is reached, it is necessary to push the node to the stub node 'ref' stack.

```
      else if (currentNode instanceof StubNode):
         this.stubNodeRefStack.push(this.currentNodeRef);
```

When an End Point node is reached, it is necessary to pop the stub node reference stack.

```
else if (currentNode instanceof EndPointNode):
  this.stubNodeRefStack.pop();
```

After handling special cases, the connection in the path can be crossed.

```
this.followConnectionRef(connRef);
```

Finally, when the Royal Guard has reached the end of the given path, landing at an event, it is necessary to move forward one connection. Afterward, any sub-Pegasi are destroyed.

```
var legalConnectionRefs = currentNode.getNextConnectionRefs();
followConnectionRef(legalConnectionRefs[0]);
this.destroyAllSubPegasi();
```

With `onEventCall()` and `followPegasus()` implemented, in addition to the `traverse()` method, the Royal Pegasi Algorithm is complete. Not only can Royal Guards and Pegasi traverse a narrative representation to preload events, thus finding sets of legal events, but they can also react to legal event calls to unload events and account for paths originally explored by Pegasi. The *Royal Pegasi Algorithm* prevents sequence breaking by updating a player's progress, in order to maintain a set of legal events.

3.6 Technical Environment and Details

We have chosen to implement our solution and its related contributions as a set of web pages using the technologies of HTML5, CSS, JavaScript, XML, jQuery, and jCanvas. HTML5, at its minimum, creates the structure of web pages, such as specifying areas where content will be placed; however, the most recent specification includes features such as canvas for drawing 2D graphics. CSS provides style to make for pleasing visuals. JavaScript defines the logic of the web page, and thus contains the majority of our code including the Narrative Manager and its *Royal Pegasi Algorithm*. XML stores data, which can be used by JavaScript, such as the player's starting position, collision detection of in-game geography, entities, and so forth. Notably, all Use Case Maps, at least those created with jUCMNav, are actually XML files with the extension ".jucm" instead. jQuery is a "JavaScript library … [that] makes … HTML document traversal and manipulation, event handling, animation, and Ajax much simpler with an easy-to-use API that works across a multitude of browsers" [89]. jCanvas, written by Caleb Evans, extends jQuery to support canvas operations, to simplify the drawing of graphics [90]. Due to cross-browser

compatibility issues, our implementation was designed for use in Mozilla Firefox and Google Chrome on Windows 7.

Beyond our solution, we have also created a testing tool, game, and scripting manager. The testing tool allows us to run our Narrative Manager on many test cases (i.e., Use Case Maps) to verify our results. The game renders simple shapes for entities and the player's character, handles user input, provides basic collision detection, and generally allows the player to attempt to break a given sequence of events. The scripting manager runs scripts provided by events when they are called, to perform basic actions (such as displaying messages to the screen) and modify variables. We chose to write our own systems from scratch for convenience, so we could maintain full control over our implementation without the need to modify code written by someone else. As we are primarily interested in a feasibility test, which only requires minimal gameplay elements, we believe the decision to make our own game and scripting language is justified, while the testing tool was a necessity to verify our solution.

Ultimately, our technical environment was chosen due to its low overhead (compared to C++ for example), 'speed' of writing code, and ease of distribution. However, it should be noted that while HTML5 is a recent advancement in web technology that is still in development [91], numerous games have been created using current specifications [92][93], including *Browser Quest* [94]. In addition, GameMaker: Studio™ Master Collection features an HTML5 exporter [95]. For the sake of feasibility, this environment seemed most practical in developing a solution to prevent sequence breaking.

Our solution, the Narrative Manager and its *Royal Pegasi Algorithm*, consists of 2,500 lines of JavaScript code (including comments), while our other contributions consist of 400 lines for our testing tool, 1,300 lines for our game, 1,200 lines for our script manager, and at least 600 lines of additional code including a class to handle input, constants, and non-JavaScript files such as HTML and CSS. In total, we wrote more than 6,000 lines of code (including comments) for this thesis.

Chapter 4 details our testing tool and game, both of which are used to collect results that verify our solution and its feasibility for preventing sequence breaking at run-time.

4 Chapter: Experimental Procedures and Results

In this chapter, we address the verification of our solution and its feasibility of preventing sequence breaking at run-time within a game environment. With respect to verification, we challenged our solution against a set of Use Case Map features we support as well as specific functionality of our Narrative Manager and its *Royal Pegasi Algorithm*. As part of these tests, we also created three Use Case Maps that represent narrative structures from popular games (as discussed in section 2.3), to further reinforce the validation of our solution as it relates to game narrative. In order to demonstrate the feasibility of our solution, we created a simple game where the player proceeds through a sequence of events that must be performed in an intended order. We then allow the player to cheat by attempting to access illegal events, but prevent such calls at run-time and thus prevent sequence breaking. Throughout this chapter, we intend to verify our solution and show that our approach can prevent sequence breaking at run-time.

4.1 Description of Experimental Procedures

Our experimental procedures apply to both the verification of our solution and the feasibility of preventing sequence breaking at run-time. For the former, we created a web page where numerous Use Case Maps can be traversed to verify intended behaviour. For the latter, we created a game called *Dungeon Explorer* to show that our solution can work within the constraints of a game environment without detracting from the experience. This section details these procedures to provide an understanding of our experiments.

Verification of our solution is shown through a web page (see figure 8), which allows for traversal among many Use Case Maps intended to show the set of UCM features and the facets of our Narrative Manager that we cover. Within this webpage—which has only been tested in Firefox and Google Chrome—the tester can select a Use Case Map example to test, view individual diagrams within that example, attempt to call arbitrary events, view the current legal set of events, and call an event from the current legal set. As the viewer proceeds through an example, by calling legal events (either by clicking an event name or entering one in the specified textbox), the legal set of events updates, allowing other events to be called until no more remain. Calling an illegal event through the textbox will result in a notification and no update to the legal set of events. Since the

tester can see the legal set of events update while having immediate access to the diagrams, one can compare the actual results against the expected behaviour. We believe this web page contains enough examples to verify the set of covered Use Case Map features and the intricacies of our Narrative Manager and its *Royal Pegasi Algorithm*.

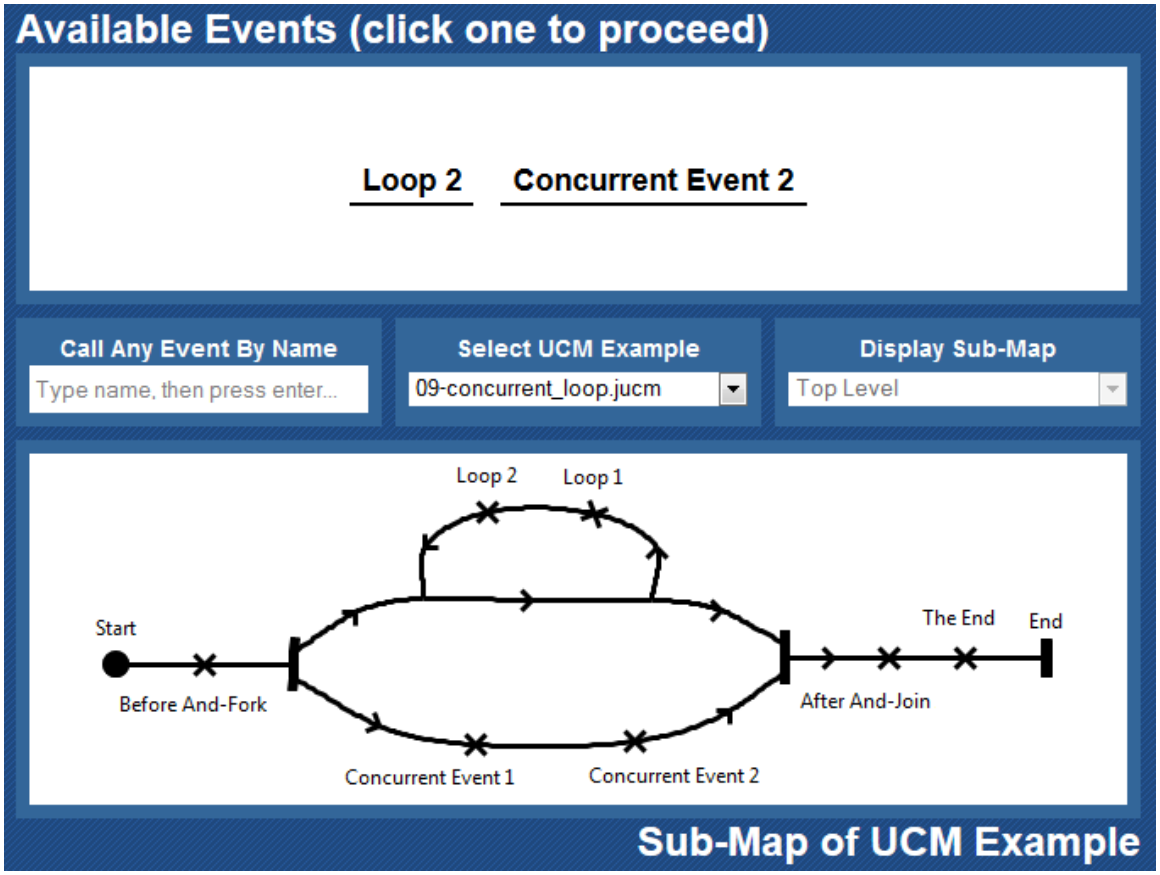


Figure 8 Verifying our solution through Use Case Maps as test cases

Feasibility is shown at run-time through *Dungeon Explorer*. In this game, the player navigates a dungeon to collect coins, which will unlock the exit when all have been found. As the player progresses they will activate switches to unlock more areas of the dungeon, granting access to more coins and switches. A Use Case Map was created to represent the intended sequence of switches (corresponding to events), along with the collection of coins and the exit condition, so that sequence breaking could be detected in case the player tried to access a switch or event out of its intended order. Further details are provided in section *4.3.1 Dungeon Explorer*.

4.2 Verification of Solution

To verify the functionality of our solution, we created and used a testing tool[6] to compare actual outcomes against expected outcomes. First, we compiled a list of requirements for the supported features from Use Case Maps that we cover. Second, we compiled an additional list of requirements based on the features of our Narrative Manager and *Royal Pegasi Algorithm*, as described earlier in chapter 3. Third, we created Use Case Maps (with jUCMNav) that would support these requirements – along with the narrative structures of three commercial games for added credibility. Finally, we tested each of these Use Case Maps to conclude that all of our requirements had been satisfied. For clarity, this section follows the aforementioned organizational approach.

4.2.1 Requirements for the Verification of Covered Use Case Map Features

Our first step to verifying our Narrative Manager and its *Royal Pegasi Algorithm* is to compile a list of requirements over the covered features of Use Case Maps. To do so, we consider the expected behaviour of each feature (e.g. node type) in reference to Amyot and Mussbacher's requirements [85], recalling our modifications from chapter 3, and add a requirement identifier (e.g., UCM-START for Start Points).

- **Start Points** (UCM-START) specify the starts of paths, such as within stubs or scenarios. Traversal proceeds by moving to the next node.
- **End Points** (UCM-END) specify the ends of paths, and may return to a stub node in another map. Traversal ends if there is no found binding to a stub node.
- **Multiple Paths** (UCM-MULT-PATHS) can be started through a scenario, and are traversed in 'parallel' such that an event from one path can be taken, then an event from another path or the same path can be taken, and so forth.
- **Responsibilities** (UCM-RESP) refer to events. When traversal reaches such a node, the associated event is preloaded; when traversal leaves such a node, the associated event is unloaded. Traversal does not move past a responsibility until the associated event has been called and then, only if the specific traversal instance (i.e., Royal Guard or Pegasus) has been given permission to proceed.

[6] The testing tool is located at: http://www.scriptedpixels.com/content/mcs-thesis/ucm-testing-tool.htm

- **Or-Forks** (UCM-OR-FORK) split a path into two or more paths, each of which may optionally contain a condition on its connection leaving the Or-Fork that determines if the path can be accessed. Only when the condition on a connection leaving the Or-Fork evaluates to false can the path not be taken, otherwise the path can always be taken. When two or more connections are available to cross, we can only speculate on the path taken next – requiring a later event call to unload events from paths not taken. Traversal proceeds in a 'speculative' manner along each crossable path; but if exactly one path is crossable, that path is taken.
- **Or-Joins** (UCM-OR-JOIN) merge two or more incoming paths together without constraint. Traversal simply proceeds to the next node.
- **Loops** (UCM-LOOP) are created with Or-Forks and Or-Joins, such that the Or-Fork determines when to (optionally) leave the loop or (optionally) stay in the loop. Traversal is formed through the behaviour of Or-Forks and Or-Joins.
- **And-Forks** (UCM-AND-FORK) split a path into two or more paths, such that all paths must be traversed in parallel – similar to the requirement of multiple paths.
- **And-Joins** (UCM-AND-JOIN) merge two or more incoming paths only when all of those paths have been or can be completed. Traversal only proceeds to the next node when the aforementioned constraint is met.
- **Waiting Places** (UCM-WAIT) only allow traversal to proceed if the condition on the single, outgoing connection evaluates to true; otherwise, traversal halts until the condition can be reevaluated.
- **Static Stubs** (UCM-STUB) refer to sub-maps, which are entered by traversing through an incoming connection bound to a start point and exited through an end point bound to an outgoing connection. We assume that incoming and outgoing bindings are always provided with stubs.
- **All other nodes** (UCM-GENERIC), such as direction arrows and empty points, should allow traversal to proceed to the next node.
- **Variables** (UCM-VAR) are an association between a key and a value of types Boolean, Integer, or any user-defined enumerated set. They can be modified or read within event scripts and read within conditions.

- **Conditions** (UCM-COND) are Boolean expressions assigned to the connection leaving a Waiting Place or optionally to a connection leaving an Or-Fork.

Referring to Amyot and Mussbacher's requirements of Use Case Map traversal provided in section 3.3, we have excluded requirements 16, 17, 18b, 20, 21, and 22 as dynamic stubs, end points connected to waiting places, and timers were not considered necessary to represent game narrative based on our observations; requirements 25, 26, 27, and 28 as we wrote our own scripting language to handle events; and, requirements 29, 30, and 31 as we assume path traversal only terminates at an end point that is not bound to a stub node. Furthermore, we have left out components as they were deemed optional [17].

In total, there are 14 requirements to consider for the set of features we cover of UCMs.

4.2.2 Requirements of Comprehensive Testing

Our second step to verifying our solution is to compile a list of extended requirements from our Narrative Manager and its *Royal Pegasi Algorithm*. To do so, we examined the code of our solution and the details described in chapter 3, and then assigned a unique identifier to each requirement for future reference.

Events were assigned four requirements:

- **Attending** (EVENT-ATTEND) an event occurs when a Royal Guard or Pegasus reaches a responsibility reference node. The Royal Guard or Pegasus is assigned to the associated event definition's list of attendees. Conversely, when a Royal Guard or Pegasus leaves an event, it is removed from the event's list of attendees.
- **Calling** (EVENT-CALL) an event runs an (optional) associated script, and then selects one or more attendees to move forward through their `onEventCall()` method, based on a setting. Afterward, all Royal Guards continue to traverse the UCM; and, any Royal Guards that become marked for deletion are destroyed.
- **Preloading** (EVENT-PRELOAD) an event modifies the environment (e.g. game world) to allow the player to call that event, and occurs when an attendee arrives at the event but no other attendees for the event exist (see EVENT-ATTEND). While an event will typically only have at most one attendee, it is possible for traversal along two or more paths containing the same event (such as through

stubs referring to a common sub-map) to reach the event at the same time; in this case, the path to continue traversal upon becomes ambiguous so we rely on an external setting provided by a game programmer.

- **Unloading** (EVENT-UNLOAD) an event modifies the environment to prevent the player from calling that event, and occurs when an attendee leaves providing no other attendees remain.

Royal Guards and Pegasi were assigned one generic requirement:

- **Creating a Pegasus** (RGP-CREATE-PEG) must result in the new Pegasus being given a reference to its parent Royal Guard or Pegasus and a copy of the parent's stub node reference stack (see sections 3.4.3 and 3.4.5).

Referring to section 3.5.1, Royal Guards and Pegasi were assigned eleven requirements for `traverse()`:

- **Crossed connections** (RGP-CROSS) must be appended to a 'traveled' list.
- **Upon arriving at a Stub** (RGP-STUB) the node reference must be pushed to the stub node reference stack, so traversal knows where to go from an End Point.
- **Upon arriving at an End Point** (RGP-END-POINT), which is bound to a stub node, the stub node reference stack must be popped and the Royal Guard or Pegasus returned to the stub node referred to in the popped element. Otherwise, the Royal Guard or Pegasus should mark itself for deletion.
- **Upon arriving at an Or-Fork** (RGP-OR-FORK) with more than one crossable outgoing connection, children Pegasi must be created to follow each legal path.
- **Upon arriving at an And-Fork** there are two requirements to consider:
    - **With a Pegasus** (RGP-AND-FORK-PEG), children Pegasi are created for each path, keeping the current Pegasus as a parent for later path following.
    - **With a Royal Guard** (RGP-AND-FORK-RG), the current Royal Guard selects any path to travel down, while creating Royal Guards for every other path. The current Royal Guard's stub node reference stack should be deep-copied [96] to each new Royal Guard.
- **Upon arriving at and And-Join** (RGP-AND-JOIN), Royal Guards and Pegasi must inform the And-Join of their attendance, including the connection that was

crossed to reach that node. When all incoming connections have been covered, there are three additional requirements to consider:

  - **With a Pegasus** (RGP-AND-JOIN-PEG), a child Pegasus must be created to explore further ahead, as the path is still considered speculative and an attendee must remain at the And-Join.
  - **With a Royal Guard, without all incoming connections containing Royal Guards** (RGP-AND-JOIN-RG-NOT), a child Pegasus must be created to explore further ahead, as not all incoming paths are guaranteed to finish. Following the newly created Pegasus will serve as finishing all incoming paths to the And-Join.
  - **With a Royal Guard, with all incoming connections containing Royal Guards** (RGP-AND-JOIN-RG-ALL), the current Royal Guard must move past the And-Join, remove itself as an attendee, and destroy all other Royal Guards, effectively preventing later access past the And-Join.

- **Destroying** (RGP-DESTROY) a Royal Guard or Pegasus located at an And-Join or Responsibility node should remove its reference from the And-Join node's or Responsibility's associated event definition's list of attendees, respectively.

Royal Guards were assigned one requirement for their `onEventCall()` method:

- **On Event Call, selected Royal Guards** (RGP-EVENT-CALL-RG) simply move past the associated responsibility reference node, crossing the next connection.

Pegasi were assigned one requirement for their `onEventCall()` method:

- **On Event Call, selected Pegasi** (RGP-EVENT-CALL-PEG) should recursively call `followPegasus()`, moving up the Pegasi tree and concatenating paths, in order to move their root Royal Guards with their `followPegasus()` methods.

Royal Guards were assigned five requirements for their `followPegasus()` method:

- **Royal Guards follow Pegasi** (RGP-FOLLOW-PEGASUS) across every stored connection within their 'traveled' list with four requirements to consider:

- **On reaching an And-Fork** (RGP-FOLLOW-PEGASUS-AND-FORK), additional Royal Guards are created for the paths not taken and given a deep copy of this Royal Guard's stub node reference stack.
- **On reaching an And-Join** (RGP-FOLLOW-PEGASUS-AND-JOIN), this Royal Guard must remove itself as an attendee and destroy absolutely all other attendees waiting at the And-Join. If Pegasi are at the And-Join, their entire Pegasi tree and root Royal Guard must be destroyed as well.
- **On reaching a Stub** (RGP-FOLLOW-PEGASUS-STUB), a reference to the node must be stored in this Royal Guard's stub node reference stack.
- **On reaching an End Point** (RGP-FOLLOW-PEGASUS-END), the stub node reference stack for this Royal Guard must be popped. We do not account for Pegasi finding End Points that are not bound to stubs, as they imply an Or-Fork that either terminates an entire path, in which case a Royal Guard would terminate and there would be no Pegasus, or that one or more other events could be called, in which case the path has not ended.

Finally, the Narrative Manager itself was assigned five requirements:

- **Trying to call a legal event** (NM-LEGAL-EVENT) should call that event, update the player's narrative progress, and then return a value of true.
- **Trying to call an illegal event** (NM-ILLEGAL-EVENT) should prevent the call to the event, and then return a value of false that can be used to notify the player of the attempted call to an illegal event (i.e., sequence breaking).
- **Trying to call an invalid event** (NM-INVALID-EVENT), such as through an invalid index or a non-existent event name, should return a value of false that can be used to notify the player of the failed event call.
- **Loading a Use Case Map** (NM-LOAD-UCM) correctly stores a valid Use Case Map, such that all specification diagrams (e.g. maps / sub-maps), responsibilities, nodes, and connections can be accessed by the Narrative Manager.
- **Loading a Scenario** (NM-LOAD-SCENARIO) correctly initializes a scenario stored within a Use Case Map given its unique name. A scenario must set variable values and place Royal Guards at every specified start point.

In total, our solution consists of an additional 28 requirements beyond those of UCMs.

4.2.3 Collecting Results and Satisfying Requirements

Our third and fourth steps consisted of creating Use Case Maps to cover each of the requirements specified in the previous two sub-sections and then testing those examples, respectively. With these examples, we performed an iterative testing process, such that we ran our Narrative Manager through each example one by one until a problem was found. When a problem was later resolved, we began our testing process from the first example and continued onward until another problem was found. When we reached the end of our examples without detecting problems, we considered our solution verified. In all, we created and tested Use Case Maps for 19 specific examples, 3 structures from commercial games for added credibility, and our *Dungeon Explorer* game. In this sub-section, we either explain how specific requirements were satisfied or, for the sake of brevity, present specific requirements satisfied in specific examples.

Minimally, each test case demonstrates NM-LOAD-UCM, NM-LOAD-SCENARIO, NM-LEGAL-EVENT, NM-ILLEGAL-EVENT, NM-INVALID-EVENT, UCM-START, UCM-RESP, EVENT-ATTEND, EVENT-PRELOAD, EVENT-UNLOAD, and RGP-CROSS. The first two requirements—NM-LOAD-UCM and NM-LOAD-SCENARIO—are immediately apparent as an example could not be interacted with, had a Use Case Map not been loaded and a scenario initialized. The second set of requirements—NM-LEGAL-EVENT, NM-ILLEGAL-EVENT, and NM-INVALID-EVENT—can be seen through the testing tool itself as a legal event can be clicked using the assigned button, while illegal or invalid event names can be entered into the textbox to be manually attempted. The third set of requirements—UCM-START and UCM-RESP—are always satisfied as at least one Start Point and at least one Responsibility (i.e., event) are in each test case. The EVENT-ATTEND requirement is shown whenever an event is added to or removed from the legal set. Similarly, The EVENT-PRELOAD and EVENT-UNLOAD requirements are shown whenever an event is added to or removed from the legal set, as a button is created or destroyed enabling or disabling calls to that event, respectively. Last, the RGP-CROSS requirement is presented as a Royal Guard must cross a connection between at least a Start Point and one other node. Each test case minimally satisfies 11

requirements for Use Case Maps, our Narrative Manager, and our traversal algorithm as can be demonstrated through the tool itself and the simplest example (figure 9).

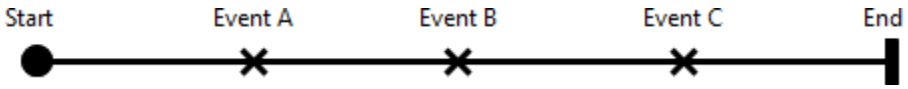


Figure 9 The simplest test case

Our next concern is handling the requirements from covered features of Use Case Maps. The UCM-END requirement is partially handled by each test case as an End Point is always given; however, Stub nodes are needed to fully satisfy its criteria (figure 10):

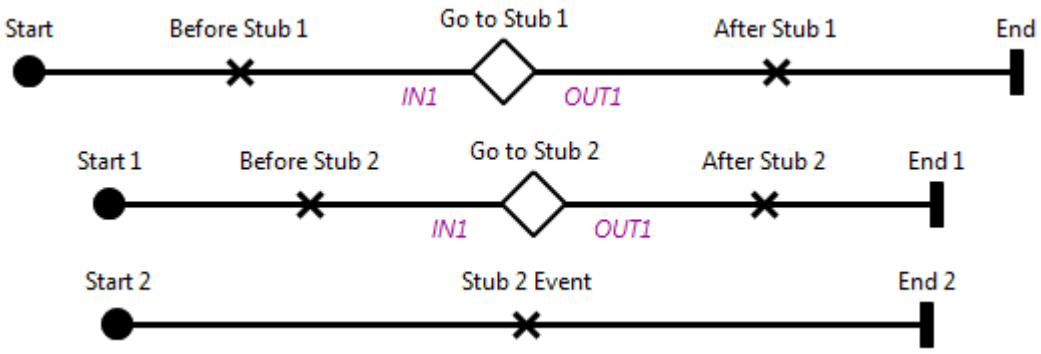


Figure 10 A test case containing stubs and end points

The UCM-STUB requirement is also satisfied in the test case presented in figure 10. The UCM-MULT-PATHS requirement needs a test case (figure 11) with multiple start points:

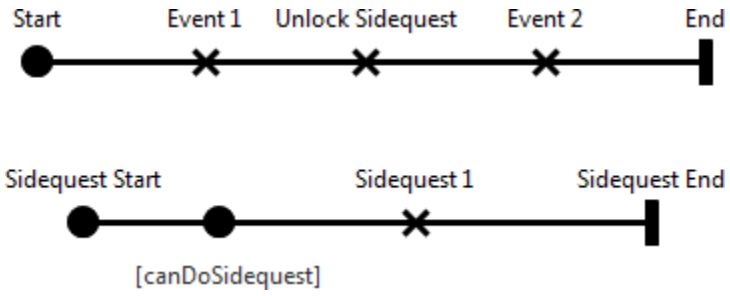


Figure 11 A test case with multiple start points and a waiting place

In addition, the example shown in figure 11 also satisfies the UCM-WAIT, UCM-VAR and UCM-COND requirements, as the `canDoSidequest` variable would be set to true in the "Unlock Sidequest" event and then read at the Waiting Place. The UCM-OR-FORK requirement needs two test cases, for exactly one legal outgoing connection and for more than one legal outgoing connection, as shown in figures 12 and 13:

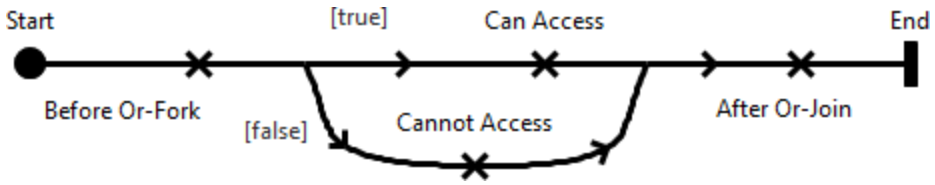


Figure 12 Or-Forks may have exactly one legal outgoing connection

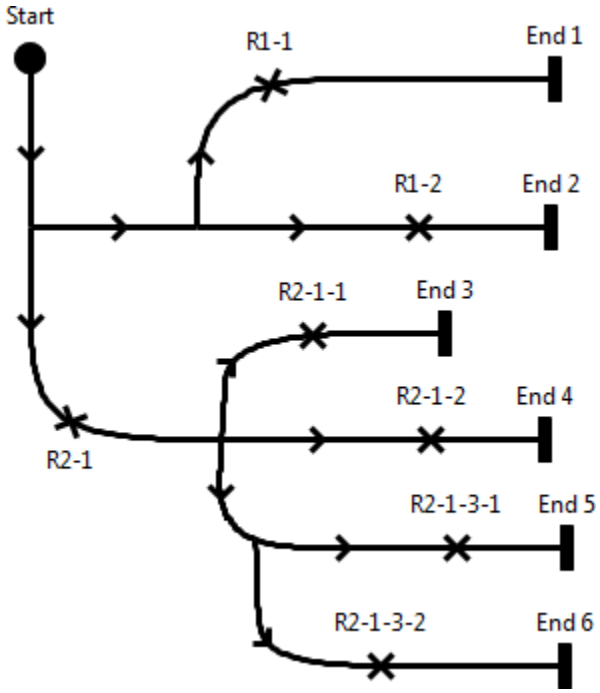


Figure 13 Or-Forks may have multiple legal outgoing connections

The UCM-GENERIC requirement is shown whenever traversal walks past a Direction Arrow, such as in the previous test cases. The UCM-OR-JOIN requirement is shown in figure 12. The UCM-LOOP requirement makes use of an Or-Fork and an Or-Join within a structure such that one or more events can be repeated, as shown in figure 14:

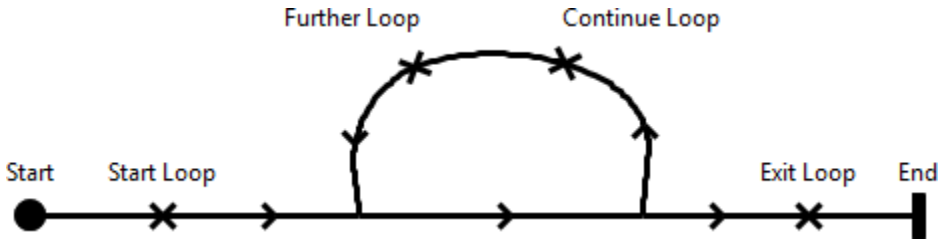


Figure 14 A loop can be formed with an Or-Fork and an Or-Join

The Or-Fork determines when to exit the loop, a choice that may be left up to the player. The UCM-AND-FORK and UCM-AND-JOIN requirements are shown in figure 15, where the player can step through events in parallel for a brief time:

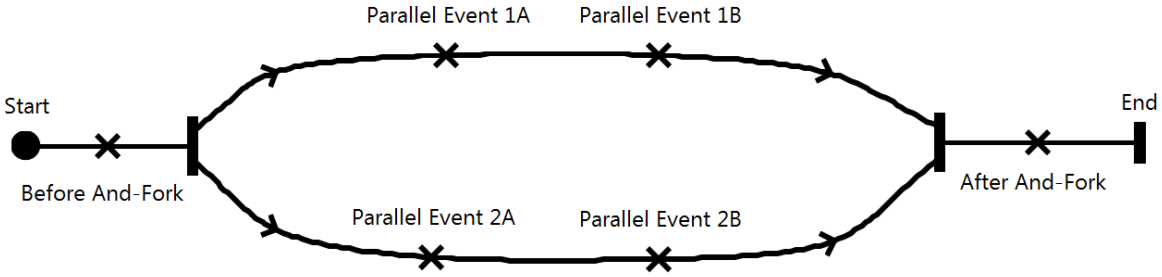


Figure 15 A test case containing both an And-Fork and an And-Join

With the above examples, we can see that test cases have been created to satisfy the requirements of covered Use Case Map features.

Moving forward, referring back to the previous examples, we have also satisfied the requirements of RGP-CREATE-PEG, RGP-STUB, RGP-END-POINT, RGP-OR-FORK, RGP-AND-FORK-RG, RGP-AND-JOIN, RGP-AND-JOIN-RG-ALL, RGP-EVENT-CALL-RG, RGP-FOLLOW-PEGASUS, and RGP-EVENT-CALL-PEG.

The EVENT-CALL requirement is satisfied with a test case for duplicate events, where a setting dictates how to handle such a case and changes the results. See figures 16 and 17:

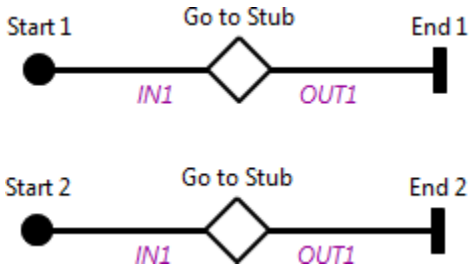


Figure 16 Top level diagram of a UCM for a duplicate event test case

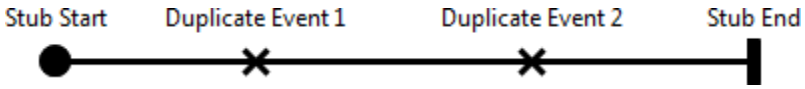


Figure 17 Sub-map for a duplicate event test case

In this example, we manually changed code in our testing tool to pass along a setting to determine which path (or both) to be updated. This feature is not presented to external testers as only one case made use of it.

To satisfy the requirements of RGP-AND-JOIN-PEG, RGP-AND-JOIN-RG-NOT, RGP-DESTROY, and RGP-FOLLOW-PEGASUS-AND-JOIN, we created a 'concurrent loop' test case, which was seen earlier in section 3.4.1 with figure 7.

To satisfy the requirements of RGP-AND-FORK-PEG and RGP-FOLLOW-PEGASUS-AND-FORK, we created the following example (figure 18):

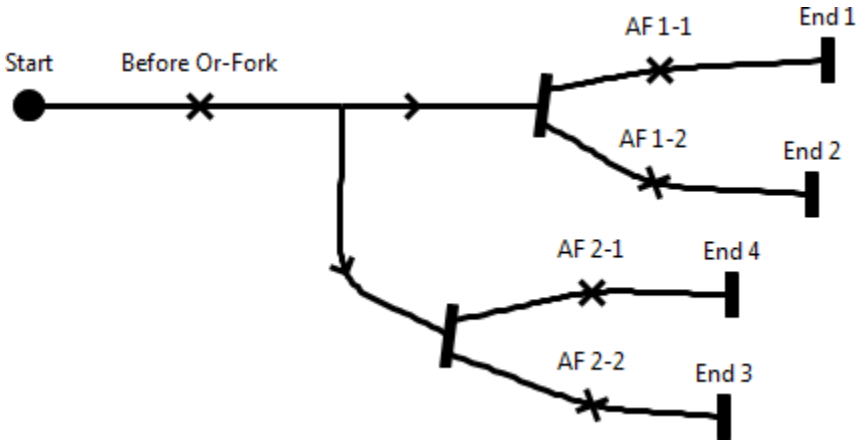


Figure 18 A test case showing And-Forks after an Or-Fork

Finally, to satisfy our last two requirements—RGP-FOLLOW-PEGASUS-STUB and RGP-FOLLOW-PEGASUS-END—we created a test case where a Pegasus can enter a stub and also exit a stub, as shown in figures 19 and 20:

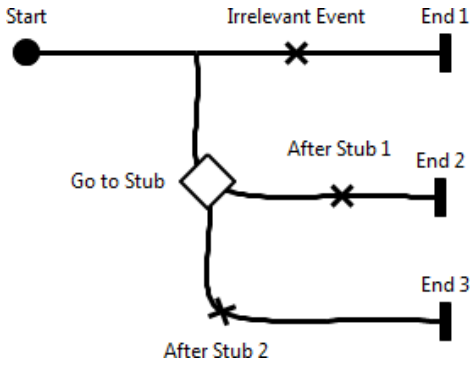


Figure 19 A Pegasus can enter a stub

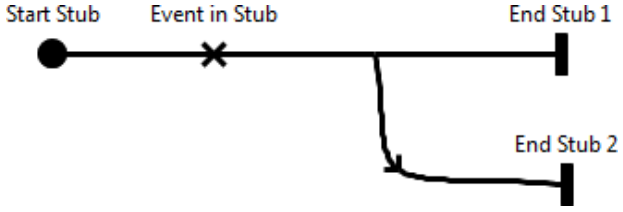


Figure 20 A Pegasus can leave a stub

It is important to note that we have excluded and thus not accounted for a specific case affecting And-Joins. Should an And-Join be placed within a stub that can be accessed through multiple paths at the same time, then the And-Join will make no distinction as to which 'thread' its incoming edges satisfy. As a result, this case will lead to errors either through allowing sequence breaking to occur or failure to update the legal set of events. This scenario was not handled as we postulate that it is an obscure case whose resolution would require extensive effort to handle.

Beyond these test cases and others created to further demonstrate specific details, we also created three Use Case Maps based on the narrative structures of *The Legend of Zelda: A Link to the Past*, *Super Mario 64*, and *Grand Theft Auto 4*. With *A Link to the Past*, the first set of dungeons before the dark world must be performed in sequence, but the seven dungeons within the dark world can be performed in almost any order provided the player chooses to collect key items from earlier dungeons to access later dungeons. It is arguable if this feature would be considered sequence breaking, but our representation treats these cases as though they were legal. While the *Super Mario 64* Use Case Map is non-exhaustive in that it does not include every potential star that could be collected, it does show how levels become unlocked as stars are collected or Bowser is defeated; in this sense, we demonstrate the major narrative progress and not the specific details. Similarly, the Use Case Map for *Grand Theft Auto 4* only includes mandatory missions and not the individual details of each mission, though these elements could be added through stubs. These additional test cases add credibility to Use Case Maps as a representation of game narrative, as it is indeed possible to take commercial examples and convert them to a format that could be used to prevent sequence breaking.

Tables 3 and 4 summarize the requirements of Use Case Maps satisfied per test case.

Test Case	**Start**	**End**	**Event**	**Arrow**	**Or-Fork**	**Or-Join**
01-simple_event_path.jucm	X	X	X			
02-simple_or_fork.jucm	X	X	X	X	X	
03-deep_or_forks.jucm	X	X	X	X	X	

04-or_fork_or_join.jucm	X	X	X	X	X	X
05-simple_and_fork.jucm	X	X	X	X		
06-and_fork_and_join.jucm	X	X	X	X		
07-simple_loop.jucm	X	X	X	X	X	X
08-extended_loop.jucm	X	X	X	X	X	X
09-concurrent_loop.jucm	X	X	X	X	X	X
10-simple_stubs.jucm	X	X	X			
11-complex_stubs.jucm	X	X	X	X	X	X
12-conditional_or_fork.jucm	X	X	X	X	X	X
13-waiting_place.jucm	X	X	X			
14-conditional_or_fork_2.jucm	X	X	X	X	X	X
15-multiple_start_points.jucm	X	X	X			
16-sidequest.jucm	X	X	X			
17-duplicate_events.jucm	X	X	X			
18-pegasus_and_fork.jucm	X	X	X	X	X	
19-pegasus_stubs.jucm	X	X	X	X	X	
GTA4_Missions.jucm	X	X	X	X	X	
Super_Mario_64.jucm	X	X	X	X	X	X
A_Link_to_the_Past.jucm	X	X	X	X		
demo_game.jucm	X	X	X	X	X	X

Table 3 Features of Use Case Maps supported by test cases (1 of 2)

Test Case	And-Fork	And-Join	Waiting Place	Stub	Variable	Condition
01-simple_event_path.jucm						
02-simple_or_fork.jucm						
03-deep_or_forks.jucm						
04-or_fork_or_join.jucm						
05-simple_and_fork.jucm	X					
06-and_fork_and_join.jucm	X	X				
07-simple_loop.jucm						
08-extended_loop.jucm						
09-concurrent_loop.jucm	X	X				
10-simple_stubs.jucm						
11-complex_stubs.jucm				X		
12-conditional_or_fork.jucm						X
13-waiting_place.jucm	X		X		X	X
14-conditional_or_fork_2.jucm					X	X
15-multiple_start_points.jucm						
16-sidequest.jucm			X		X	X
17-duplicate_events.jucm				X		
18-pegasus_and_fork.jucm	X					

19-pegasus_stubs.jucm				X		
GTA4_Missions.jucm	X	X				
Super_Mario_64.jucm				X	X	X
A_Link_to_the_Past.jucm	X	X		X		
demo_game.jucm	X	X	X	X	X	X

Table 4 Features of Use Case Maps supported by test cases (2 of 2)

As demonstrated in the test cases shown above, we have satisfied all of the requirements covered in our selected features of Use Case Maps, our Narrative Manager, and our *Royal Pegasi Algorithm*. The final step was to then run our examples through a testing tool to ensure that their actual behaviour matched our expectations.

To verify behaviour through our testing tool, our primary focus consisted of checking the legal set of events against our expected results at any given point within a sequence. In simple scenarios, we tested every possible path to ensure correct results along the way. In complex examples, where the number of possible paths could be very high, we tested numerous random and specific paths by restarting the test case between iterations. It should also be noted that some events were assigned scripts that are not accessible (in an obvious way) to testers, as they serve only to increment integers or set Booleans to true and the associated events are typically obvious. In one example, a setting to handle duplicate events was entered manually by altering our tool's code. Furthermore, in other test cases, variables within a narrative representation cannot be viewed without access to a debugging tool such as Firebug. Despite three drawbacks, our testing tool was sufficient enough for us to verify our solution.

Having compiled a list of requirements, satisfied those requirements through test cases, and then verified the expected behaviour of each test case, we believe that our solution has been verified based upon our expected behaviour of it.

4.3 Feasibility of Solution

Beyond verifying the behaviour of our Narrative Manager, it is important to illustrate its ability to prevent sequence breaking within a game environment at run-time. To handle this task, we created a simple game called *Dungeon Explorer* with a narrative structure that captures many of the features handled in our solution. In addition, we allow the player to cheat in attempt at sequence breaking. Our goal is to ensure that our solution to preventing sequence breaking works at run-time with unnoticeable delay, and that it is practical to create a game with Use Case Maps that describe narrative structure. The results of play-testing our game show that our solution is feasible, though shortcoming will be provided in our conclusion as part of future work (see chapter 5).

4.3.1 *Dungeon Explorer*

To illustrate the feasibility of preventing sequence breaking with our Narrative Manager at run-time, it was deemed necessary to create a game: *Dungeon Explorer*[7]. In this game, the player (represented by a dark blue circle with direction indicator) must navigate a dungeon (comprised of dark grey walls) to collect coins (yellow circles), step on switches (blue squares) to unlock paths by removing barriers (grey squares with an X shape), and make their way to the exit, which requires all coins to be found before unlocking. Optionally, the player can collect torches (orange triangles) to provide more 'light' around their character, as most of the screen is masked in black, making navigation difficult. By associating switches and coin pickup with events—some of which run scripts—we created a Use Case Map to specify an intended sequence of progression. Furthermore, switches corresponding to legal events are enabled, while switches corresponding to illegal events (unless they have already been activated) are disabled, showing our preload and unload methods. When the intended sequence is broken through in-game cheats, the player is notified and returned to the starting position. This demo contains all covered features of Use Case Maps and our Narrative Manager and many aspects of our *Royal Pegasi Algorithm* to show the practical application of our solution.

With the masking feature enabled, the player's view is very limited (figure 21):

[7] *Dungeon Explorer* is located at: http://www.scriptedpixels.com/content/mcs-thesis/dungeon-explorer.htm

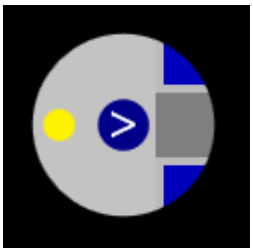


Figure 21 Limited visibility in *Dungeon Explorer*

If the masking feature is disabled, the entire map becomes visible, as shown in figure 22:

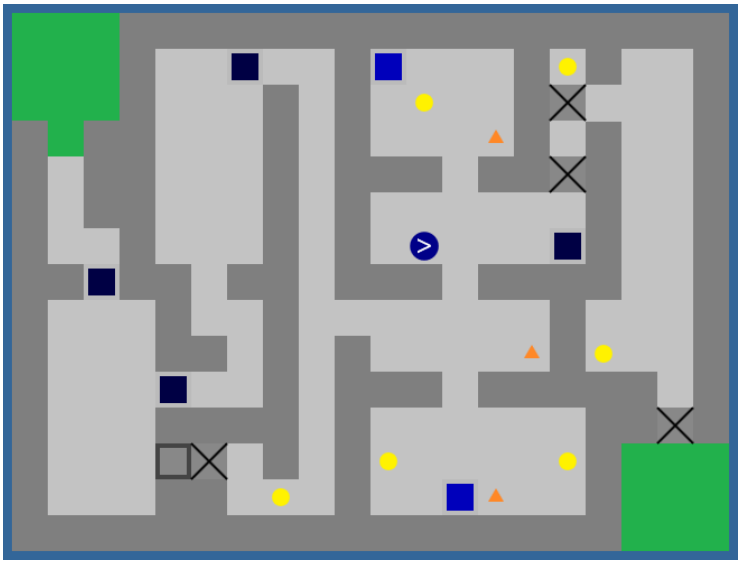


Figure 22 Screenshot from *Dungeon Explorer* without masking

The controls of *Dungeon Explorer* are simple:

- Arrow keys move the player's character up, down, left, and right
- Z closes an open textbox (once all of its text has been displayed)

In addition, the player may also enable and/or disable cheats:

- Q toggles the masking feature, allowing the entire map to be visible
- W toggles collision detection, allowing the player to walk through barriers
- E toggles 'always-enabled' switches, allowing illegal switches to be stepped on in attempt to activate them – though sequence breaking is detected at this point, and the switch is not activated

By disabling collision detection and setting switches to 'always be enabled,' the player can attempt to sequence break by stepping on a switch that would otherwise be disabled,

thus trying to call an illegal event. However, if a switch was already activated, it cannot become activated again, as per the states of switches.

There are three states for switches: disabled, enabled, and activated. All switches begin in the disabled state, meaning their associated event is illegal to call. When the associated event becomes legal to call, the switch becomes enabled, meaning the player can step on it to make the switch activated, calling the event. Activated switches always stay in that state. When the associated event for a switch becomes illegal and the switch is enabled but not activated, then the switch reverts back to its disabled state. The transition between enabled and disabled states is handled by the preload and unload methods for associated events, respectively. The 'always-enabled' cheat forces disabled switches to stay in their enabled state, though they may only become activated if the associated event is legal. Thus, illegal events call may be attempted but are prevented. Table 5 shows these states:

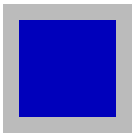
Disabled / 'Illegal'	**Enabled / 'Legal'**	**Activated**
		

Table 5 States of switches in *Dungeon Explorer*

A Use Case Map was created to represent the player's intended progression through the dungeon, as shown in figures 23 and 24. In particular, each switch is associated with an event, while each coin adds to a counter that will satisfy the condition for the exit.

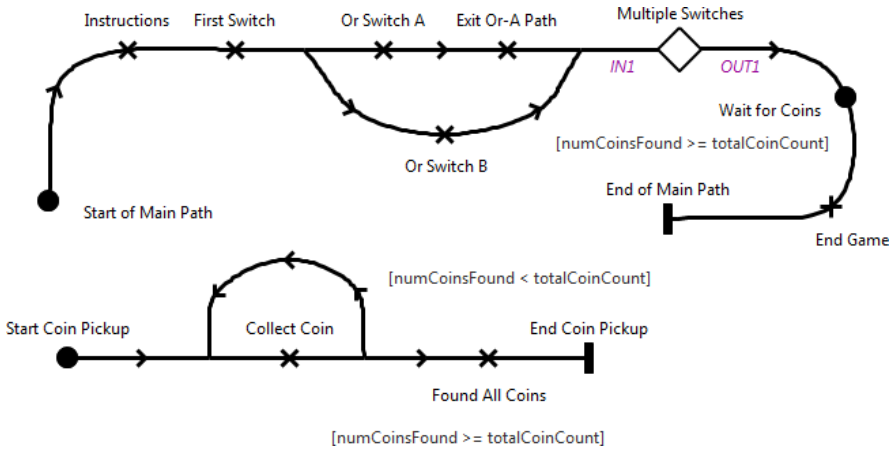


Figure 23 Top Level Use Case Map for *Dungeon Explorer*

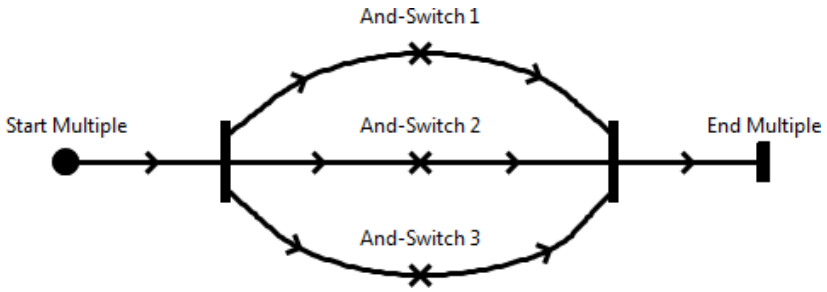


Figure 24 Sub-Map for "Multiple Switches" stub

Notice that two variables were created to be used in this representation: `numCoinsFound` and `totalCoinCount`. `numCoinsFound` stores the number of coins found by the player, and is incremented every time the player picks up a coin through the event "Collect Coin," while `totalCoinCount` serves as a constant to indicate how many coins are in the dungeon. With these two variables, we can enforce a condition for the exit.

As can be seen in figures 23 and 24, this game supports linear sequences of events, Or-Forks with multiple legal outgoing edges, Or-Forks with exactly one legal outgoing edge, Or-Joins, loops, And-Forks, And-Joins, Waiting Places, Static Stubs, multiple paths, variables, and conditions. Thus, all covered UCM features that we support are included.

It is our belief that *Dungeon Explorer* is an adequate candidate for showing the practical application of our Narrative Manager (and its *Royal Pegasi Algorithm*) within a game environment for the prevention of sequence breaking at run-time. Given that all covered features of Use Case Maps that we support are handled, including 'concurrency' and split paths; preloading, calling, and unloading of events; and, handling sequence breaking when an illegal event call is attempted; it is reasonable to conclude that this example fully demonstrates the major aspects of our Narrative Manager within a game environment.

4.3.2 Feasibility and Development Insight

In creating a game to demonstrate the practical application of our Narrative Manager, we have determined that our solution is indeed feasible for preventing sequence breaking at run-time within an unnoticeable amount of time. To argue this point we will describe and show features of our Narrative Manager through screenshots, and record the time of each update to the player's narrative progress. We conclude that our solution performs the

desired functionality, as specified in chapter 1, to prevent sequence breaking, within a game environment, within a time that cannot be detected by a human.

Recall from chapter 1 that our solution to sequence breaking must include: checking the legality of an attempted event call given its unique identifier; updating the player's narrative progress when a legal event is called by keeping track of the legal set of events, and then preloading newly-legal events and unloading newly-illegal events; and, rejecting illegal event calls with an optional callback function to resolve the conflict. Each of these requirements has been satisfied in *Dungeon Explorer*, as demonstrated in screenshots.

Checking the legality of an attempted event call given its unique identifier occurs whenever the player picks up a coin, steps on a switch, or tries to exit the dungeon. When a called event is legal, most notably shown through switches, the game world updates to preload access to newly-legal events and unload access to newly-illegal events. We can see the legality check, preloading of events, unloading of events, and update of narrative progress within figures 25, 26, and 27 where the player steps on switches:

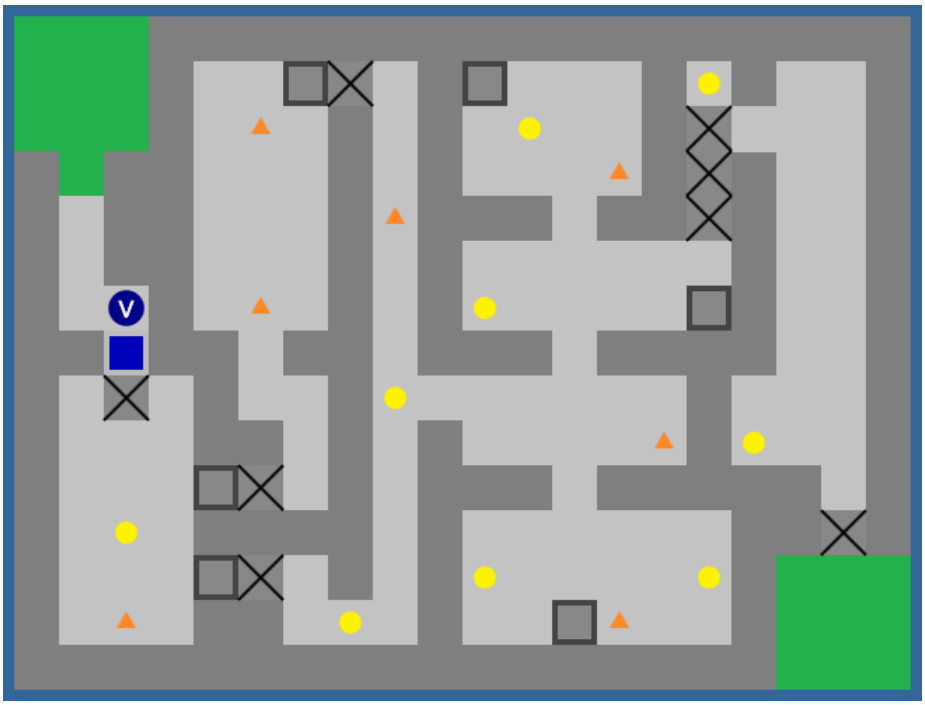


Figure 25 Before stepping on the first switch

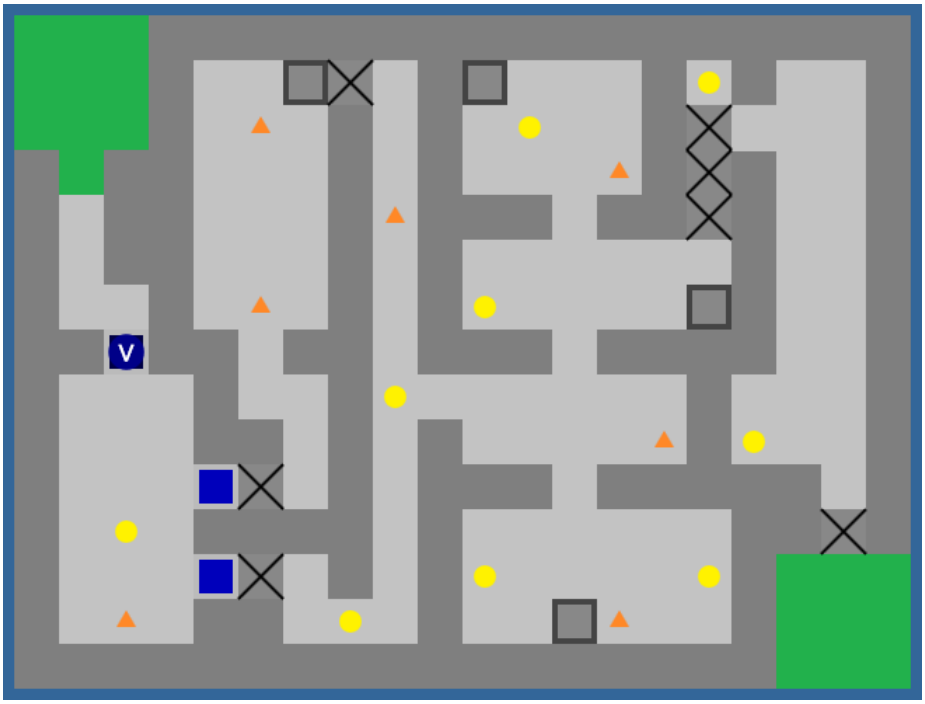


Figure 26 Upon stepping on the first switch, two new switches become enabled

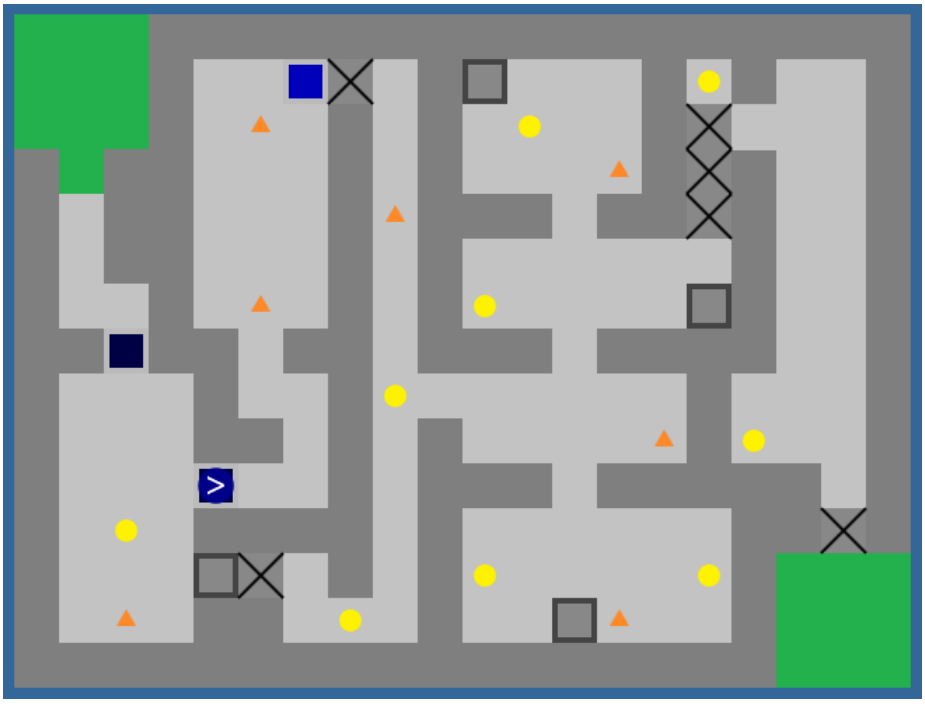


Figure 27 Stepping on an 'Or-Fork' switch disables the alternative path

As can be seen in the above screenshots, the player calls an event when they step on an associated switch, to remove barriers allowing for further exploration in the dungeon.

When an event is preloaded, through the game's `preloadEventByName()` method, which takes in an event name and handles the preloading for that event, the switch (if any) connected to that event is set to be enabled. When an event is unloaded, through the game's `unloadEventByName()` method, which similarly takes in an event name and handles unloading for that event, the connected switch (if any) is set to its disabled state, provided the switch has not been activated. In order for the switches to change their state and for barriers to be removed, the event associated with the switch must be legal to call – thus we partially satisfy the checking requirement, but still need to prove that illegal calls are rejected. We can demonstrate such sequence breaking, through the calling of illegal events, by activating cheats for "always-enabled" switches and walk-through-barriers, resulting in the detection of sequence breaking as seen in figures 28 and 29:

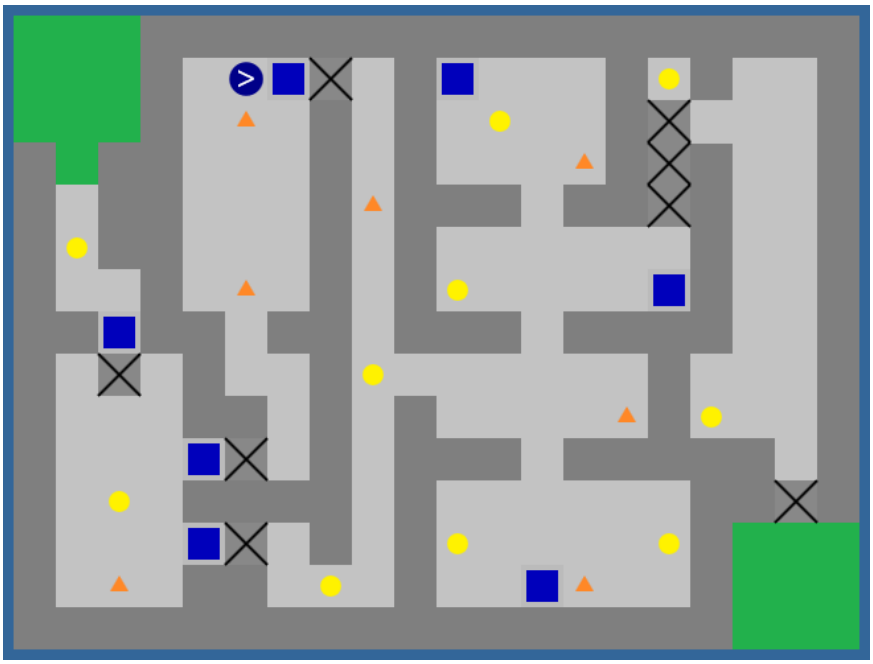


Figure 28 Screenshot of *Dungeon Explorer* before sequence breaking

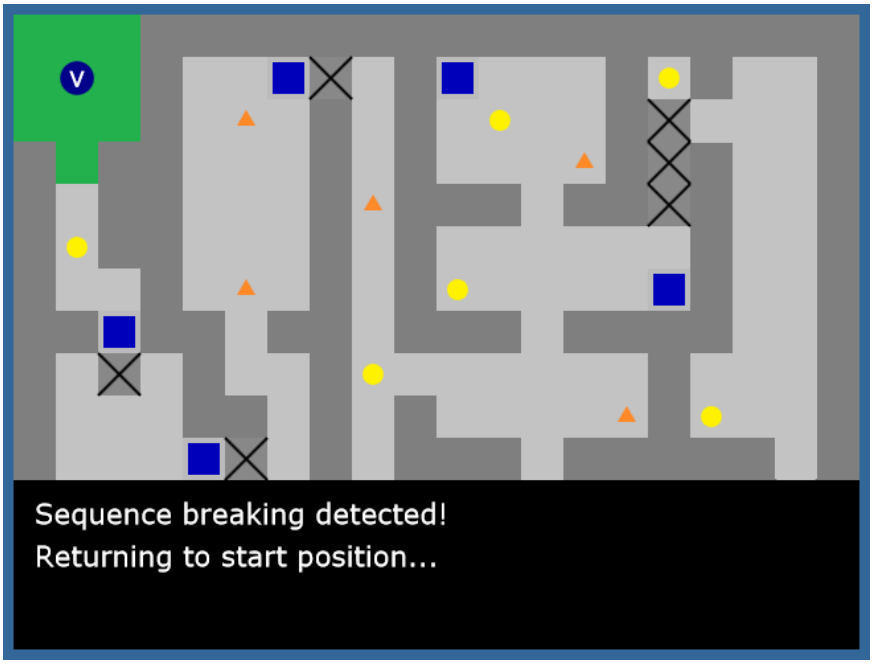


Figure 29 Screenshot of *Dungeon Explorer* with sequence breaking detected

Through these screenshots we have shown that *Dungeon Explorer*, through our Narrative Manager, is capable of checking the legality of attempted event calls, updating the player's progress, preloading events, unloading events, and most importantly rejecting calls to illegal events, effectively preventing sequence breaking.

Having demonstrated the requirements of our solution within a game environment, our focus now turns to suggesting that our Narrative Manager can perform the necessary operations within an unnoticeable amount of time, such that the player's experience is not affected. For this purpose, we have instrumented a timer into each 'update' caused by a call to a legal event [97]. As detecting sequence breaking is performed in O(1) time, by checking if a key exists within an associative array, it is not necessary to count the time for illegal calls or even the legality check as it is negligible. Instead, we focus our timer on the traversal of our game's Use Case Map in addition to the preload and unload callbacks. The average update time, adjusted average update time (where updates less than 1 millisecond in duration are rounded up), maximum update time, and minimum update time are given for five play-throughs where events were called in different legal sequences. The results of our performance testing are recorded in table 6:

Iteration / Statistic	Iteration 1	Iteration 2	Iteration 3	Iteration 4	Iteration 5
Average	1.39 ms	1.5 ms	1.44 ms	1.28 ms	1.67 ms
Adjusted Average	1.56 ms	1.56 ms	1.5 ms	1.39 ms	1.72 ms
Maximum	3 ms	6 ms	4 ms	3 ms	4 ms
Minimum	<1 ms	<1 ms	<1 ms	<1 ms	<1 ms

Table 6 Performance of *Royal Pegasi Algorithm* within *Dungeon Explorer*

(Extended details of these iterations are provided in Appendix B.)

From these results, it is clear to see that our average update time is small, while our maximum traversal was at most 6 milliseconds. Even if every update in our example took 6ms, this time is not at all detectable by a human being – as can be demonstrated by the fact that computers have a refresh rate between 60 – 120 Hz (or 8ms to 16ms) [98]. Our algorithm takes less than a 'frame' at most for our game, and even then the processing occurs within a separate thread. Furthermore, the occurrence of the computation by our Narrative Manager is only when a legal event completes, which is relatively uncommon in comparison to the number of 'frames' where events are not completed. Thus, we can suggest that time spent towards updating the player's narrative progress, preloading events, and unloading events is not significant enough to disrupt the player's experience.

As our Narrative Manager was able to prevent sequence breaking at run-time, within a game environment, while performing the required operations, within a period of time that is unnoticeable to a human, we believe it is reasonable to conclude that our solution is indeed feasible. As we conclude this dissertation in the next chapter, we will recapitulate our contributions and discuss the shortcomings of our solution.

5 Chapter: Conclusions and Future Work

Throughout this dissertation, we presented a means of managing game narrative with the aid of Use Case Maps for the purpose of preventing sequence breaking at run-time. We verified that our solution behaved as expected and showed that it was feasible.

To reiterate, sequence breaking is a type of feature interaction conflict, akin to invocation order, that occurs when the player accesses an in-game event outside of its intended order, effectively causing unexpected behaviour in the game's subsuming feature—its storyline—as the story may be presented incorrectly. While sequence breaking is often performed by skilled players for their own enjoyment, such cheating could be detrimental to other players who fall victim to these unfair advantages. Worse still, when a player inadvertently sequence breaks, the story of the game may lose its integrity or the game may become unwinnable. In providing a means of preventing and handling sequence breaking at run-time, we reduce the need to determine how a player might access events out of order and instead focus on resolving such conflicts when they occur.

To find a solution to sequence breaking, we decided that a representation scheme for valid sequences of events was necessary, so that we could track the player's narrative progress and verify the legality of events. First, we reviewed literature on game narrative and observed the narrative structures of five commercial games to develop a list of necessary features to represent; we selected Use Case Maps as we deemed the format to be sufficient and easy to understand. Second, we created our Narrative Manager and *Royal Pegasi Algorithm* to monitor the player's narrative progress and perform legality checks on attempted event calls, to prevent sequence breaking, in addition to preloading events when they become legal and unloading events when they become illegal, to reduce calls to events that should be inaccessible. Third, we compiled a list of requirements for the features of Use Case Maps that we support along with the intricate details of our Narrative Manager and its *Royal Pegasi Algorithm*, and then verified the actual behaviour of our implementation against the expected behaviour through test cases supporting these requirements. Finally, we created a game called *Dungeon Explorer* to test the feasibility of preventing sequence breaking at run-time with our approach. In following these steps,

we were able to create a solution that we determined to be sufficient for representing game narrative, to produce expected results, and to be feasible.

Unfortunately, an arguably major drawback of our solution exists for which we do not see an easy answer. As updating of the player's progress occurs only when a legal event is called, it is possible for conditions, such as those on waiting places, to become satisfied outside of an event call. Our solution assumes that narrative-related variables are updated through events, hence the need to have an event for "Collect Coin" in *Dungeon Explorer*.

However, as an inherent benefit, we allow game programmers the ability to fetch the current set of legal events: a feature that could be helpful in producing seemingly more-intelligent non-playable characters, such as through improved dialogue.

To conclude, we believe that using Use Case Maps as a representation scheme for game narrative leads to a feasible solution for the prevention of sequence breaking in video games. Furthermore, in finding a solution to this problem, we believe that we have made a significant contribution to the greater and well-known topic of feature interaction.

5.1 Future Work

Should we continue our work in the future, we would like to support additional features of Use Case Maps, improve upon our code, resolve shortcomings of our solution, and allow both saving and loading of the player's narrative progress. As our focus was on the feasibility of Use Case Maps for the prevention of sequence breaking at run-time, our solution was limited to features strictly necessary to represent game narrative based on common techniques found in literature and structures of commercial games.

For Use Case Maps, we would like to add support for preconditions on start points, dynamic stubs, end points connecting to waiting places, timers with timeout paths, end points connecting to timers, components, and expressions stored within responsibilities. While including most of these additional features would simply fulfill the remaining requirements of Use Case Map traversal as specified by Amyot and Mussbacher [85], components could be used to assign events to specific objects, locations, or actors to provide clarity, and expressions stored within responsibilities could be treated as post-conditions for events, allowing game designers to include this logic directly in their

narrative structures instead of through scripts in an external file. In supporting these additional features of Use Case Maps, we can better take advantage of this representation scheme for game narrative.

For improving our code, we would like to optimize our algorithm such that events are only preloaded or unloaded once per event call as currently an event may be preloaded, unloaded, and then preloaded again (or some variant). We could fix this redundancy by only preloading or unloading events after we have determined the new set of legal events instead of during traversal as we currently do now, by examining changes to the set.

For resolving shortcomings, we would like to handle more-complex variable conditions, as we only support simple comparisons (i.e., variable vs variable or variable vs constant); include a callback function for game programmers when an event completes its script; and, consider a better alternative to preloading and unloading events through external means, as our approach relies on game programmers to look up what do to given an event name, possibly in O(n) time (where n is the number of events). We would also like to reduce our assumptions, such as with stub bindings, to be less strict.

For allowing saving and loading of the player's narrative progress, we would like to save the locations of Royal Guards and the values of narrative variables to a file (or possibly a cookie in a web browser), which could later be loaded and used in another play session.

Moving forward, we would be interested to see our solution as less a means of strictly preventing sequence breaking and more a means of managing game narrative through Use Case Maps. We believe that our solution's ability to monitor and update a player's narrative progress through a predefined narrative structure in real-time without negatively affecting the player's experience can be of benefit to game programmers and designers, as was briefly touched upon with fetching of the player's current set of legal events.

Bibliography

[1] S. Tsang and E. H. Magill, "Detecting Feature Interactions in the Intelligent Network," University of Stratchclyde, 1993.

[2] K. Czarnecki, Generative Programming, Ilmenau, Germany: Technical University of Ilmenau, 1998.

[3] Y. Jia and J. M. Atlee, "Run-Time Management of Feature Interaction," in *ICSE Workshop on Component-Based Software Engineering (CBSE6)*, 2003.

[4] ICFI 2012, "About ICFI," 29 July 2011. [Online]. Available: http://www27.cs.kobe-u.ac.jp/wiki/icfi2012/index.php?AboutICFI. [Accessed 15 December 2012].

[5] IGN, "Pokemon Report: OMG Hacks," 24 November 2008. [Online]. Available: http://ds.ign.com/articles/933/933126p1.html. [Accessed 11 May 2012].

[6] tvtropes.org, "Sequence Breaking," 9 May 2012. [Online]. Available: http://tvtropes.org/pmwiki/pmwiki.php/Main/SequenceBreaking. [Accessed 11 May 2012].

[7] RFSmediaproductions, "Pokemon Blue: "The MissingNo. Trick" revealed," 4 January 2007. [Online]. Available: http://youtu.be/-tOyBahKkbk. [Accessed 18 March 2013].

[8] Kotaku, "Game-Breaking Skyward Sword Bug Confirmed. Here's How to Avoid It.," 7 December 2011. [Online]. Available: http://kotaku.com/5865426/game+breaking-skyward-sword-bug-confirmed-heres-how-to-avoid-it. [Accessed 15 May 2012].

[9] J. D. Hay and J. M. Atlee, "Composing Features and Resolving Interactions," in

ACM International Symposium on the Foundations of Software Engineering (FSE), 2000.

[10] K. P. Pomakis and J. M. Atlee, "Reachability Analysis of Feature Interactions: A Progress Report," in *International Symposium on Software Testing and Analysis (ISSTA)*, 1996.

[11] P. K. Au and J. M. Atlee, "Evaluation of a State-Based Model of Feature Interactions," *Feature Interactions in Telecommunications Networks,* vol. IV, p. 153, 1997.

[12] K. H. Braithwaite and J. M. Atlee, "Towards Automated Detection of Feature Interactions," in *Second International Workshop on Feature Interactions in Telecommunications Software Systems*, 1994.

[13] ICFI 2009, "AboutICFI - ICFI 2009," 7 October 2008. [Online]. Available: http://www27.cs.kobe-u.ac.jp/wiki/icfi/index.php?AboutICFI. [Accessed 13 June 2012].

[14] S. Carless, Gaming Hacks: 100 Industrial-Strength Tips and Tools, Sebastopol, CA: O'Reilly Media, 2005.

[15] GTA Network.com, "GRAND THEFT AUTO IV - Mission Walkthroughs," 2009. [Online]. Available: http://www.gta4.net/missions/index.php. [Accessed 15 December 2012].

[16] TASVideos, "TASVideos / Welcome To TAS Videos," 8 April 2012. [Online]. Available: http://tasvideos.org/WelcomeToTASVideos.html. [Accessed 16 May 2012].

[17] D. Amyot and G. Mussbacher, "User Requirements Notation: The First Ten Years, The Next Ten Years," *Journal of Software,* vol. 6, no. 5, pp. 747-768, 2011.

[18] J. Whitehead, February 26 2007. [Online]. Available: http://classes.soe.ucsc.edu/cmps080k/Winter07/lectures/narrative.pdf. [Accessed 18 January 2013].

[19] J. Juul, "Games Telling stories?," Game Studies, July 2001. [Online]. Available: http://www.gamestudies.org/0101/juul-gts/. [Accessed 18 January 2013].

[20] MobyGames, "Genre Definitions," [Online]. Available: http://www.mobygames.com/glossary/genres. [Accessed 18 January 2013].

[21] J. Majewski, Theorising Video Game Narrative, Bond University, 2003.

[22] A. Nhlabatsi, R. Laney and B. Nuseibeh, "Feature interaction: the security threat from within software systems," *Progress in Informatics,* pp. 75-89, 2008.

[23] M. Shehata, A. Eberlein and A. O. Fapojuwo, "A taxonomy for identifying requirement interactions," *Computer Networks,* pp. 398-425, 2007.

[24] M. Nakamura, T. Kikuno, J. Hassine and L. Logrippo, "Feature Interaction Filtering with Use Case Maps at Requirements Stage," in *Sixth International Workshop on Feature Interactions in Telecommunications and Software Systems (FIW'00), IOS Press*, 2000.

[25] M. Nakamura, P. Leelaprute and T. Kikuno, "Deriving Interaction-Prone Scenarios in Feature Interaction Filtering with Use Case Maps," in *Proceedings of the Seventh International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, 2002.

[26] H. M. Chandler, The Game Production Handbook, Infinity Science Press Llc, 2009.

[27] J. Ward, "What is a Game Engine?," 29 April 2008. [Online]. Available: http://www.gamecareerguide.com/features/529/what_is_a_game_.php. [Accessed 19 January 2013].

[28] H. Korhonen, "Comparison of Playtesting and Expert Review Methods in Mobile Game Evaluation," in *Proceedings of the 3rd International Conference on Fun and Games*, 2010.

[29] T. Mahlmann, A. Drachen, J. Togelius, A. Canossa and G. Yannakakis, "Predicting Player Behaviour in Tomb Raider: Underworld," in *Symposium on Computational Intelligence and Games (CIG), 2010 IEEE*, 2010.

[30] K. Hullet, N. Nagappan, E. Schuh and J. Hopson, "Data Analytics for Game Development (NIER Track)," in *33rd International Conference on Software Engineering (ICSE)*, 2011.

[31] A. Drachen and A. Canossa, "Towards Gameplay Analysis via Gameplay Metrics," in *Proceedings of the 13th International MindTrek Conference: Everyday Life in the Ubiquitous Era*, 2009.

[32] A. Drachen and A. Canossa, "Analyzing Spatial User Behaviour in Computer Games using Geographic Information Systems," in *Proceedings of the 13th International MindTrek Conference: Everyday Life in the Ubiquitous Era}*, 2009.

[33] PBSoffbook, "The Art of Glitch | Off Book | PBS," 9 August 2012. [Online]. Available: http://youtu.be/gr0yiOyvas4. [Accessed 20 January 2013].

[34] kazztawdal, "Let's Play San Francisco Rush," 25 June 2009. [Online]. Available: http://youtu.be/OJyCppPNEeM. [Accessed 20 January 2013].

[35] S. Redon, A. Kheddary and S. Coquillart, "Fast Continuous Collision Detection between Rigid Bodies," *Computer Graphics Forum,* pp. 279-287, 2 June 2009.

[36] tvtropes, "Unwinnable by Mistakes," 20 August 2012. [Online]. Available: http://tvtropes.org/pmwiki/pmwiki.php/Main/UnwinnableByMistake. [Accessed 20 January 2013].

[37] J. Newman, "(Not) Playing Games: Player-Produced Walkthroughs as Archival Documents of Digital Gameplay," *International Journal of Digital Curation,* vol. 6, no. 2, pp. 109-127, 2011.

[38] J. Newman, Playing (with) videogames, New York: Routledge, 2008.

[39] S. Carless, Gaming Hacks: 100 Industrial-Strength Tips & Tools, Sebastopol, CA: O'Reilly Media, Inc., 2005.

[40] M. Eladhari, Object Oriented Story Construction in Story Driven Computer Games, Stockholm, Sweden: Stockholm University, 2002.

[41] GameFAQs, "The Legend of Zelda: A Link to the Past," [Online]. Available: http://www.gamefaqs.com/snes/588436-the-legend-of-zelda-a-link-to-the-past/data. [Accessed 19 January 2013].

[42] OptiGamer, "Zelda sales charts and sequel announced," 22 February 2005. [Online]. Available: http://web.archive.org/web/20050223002315/http://www.optigamer.com/news/?id=733. [Accessed 19 January 2013].

[43] G. Mueller, "The Greatest Games of All Time: The Legend of Zelda: A Link to the Past - GameSpot.com," [Online]. Available: http://www.gamespot.com/features/the-greatest-games-of-all-time-the-legend-of-zelda-a-link-to-the-past-6145817/. [Accessed 20 January 2013].

[44] tvtropes, "Sequence Breaking," 18 January 2013. [Online]. Available: http://tvtropes.org/pmwiki/pmwiki.php/Main/SequenceBreaking. [Accessed 19 January 2013].

[45] ShadowPikachu420, "How To Beat Link A To The Past In 5 Minutes," 18 April 2012. [Online]. Available: http://youtu.be/1bjYuzDqhHU. [Accessed 19 January

2013].

[46] Gamespot, "Super Mario 64 Related Games," [Online]. Available: http://www.gamespot.com/super-mario-64/related/platform/n64/. [Accessed 19 January 2013].

[47] CuttingTheEdge, "All Time Top 20 Best Selling Games and More," 21 May 2003. [Online]. Available: http://web.archive.org/web/20060221044930/http://www.ownt.com/qtakes/2003/gamestats/gamestats.shtm. [Accessed 19 January 2013].

[48] Wiej, "GameFAQs: Super Mario 64 (N64) FAQ/Walkthrough by Wiej," 22 October 2009. [Online]. Available: http://www.gamefaqs.com/n64/198848-super-mario-64/faqs/58059. [Accessed 6 March 2013].

[49] R. Harrison, "Super Mario 64 Walkthrough - IGN FAQs," 1 January 2006. [Online]. Available: http://faqs.ign.com/articles/566/566275p1.html. [Accessed 20 January 2013].

[50] kingddd, "Super Mario 64 Stair Glitch," 15 January 2006. [Online]. Available: http://youtu.be/cb-7NZoNaBg. [Accessed 20 January 2013].

[51] S. Totilo, "'Grand Theft Auto IV' Posts Record First-Week Sales," 7 May 2008. [Online]. Available: http://www.mtv.com/news/articles/1586971/grand-theft-auto-iv-posts-record-first-week-sales-report.jhtml. [Accessed 20 January 2013].

[52] K. Orland, "Grand Theft Auto IV Passes 22M Shipped, Franchise Above 114M," 14 September 2011. [Online]. Available: http://www.gamasutra.com/view/news/37228/Grand_Theft_Auto_IV_Passes_22M_Shipped_Franchise_Above_114M.php. [Accessed 20 January 2013].

[53] Nintendo Co., Ltd., "Financial Results Briefing for Fiscal Year Ended March 2011," [Online]. Available:

http://www.nintendo.co.jp/ir/pdf/2011/110426e.pdf#page=6. [Accessed 20 January 2013].

[54] B. Molina, "'Pokemon' titles sell 1 million on launch day," 9 March 2011. [Online]. Available: http://content.usatoday.com/communities/gamehunters/post/2011/03/pokemon-titles-sell-1-million-on-launch-day/1. [Accessed 20 January 2013].

[55] magmar1x, "Pokemon HeartGold & SoulSilver Walk Through Walls Action Replay," 20 September 2009. [Online]. Available: http://youtu.be/v5SIDyv8YM0. [Accessed 21 January 2013].

[56] PrimalGrovyle, "Pokemon HG/SS: Walk Through Walls part 1," 12 October 2010. [Online]. Available: http://youtu.be/Btl4UvHHNAU. [Accessed 21 January 2013].

[57] 1UP, "1UP's Best of 2011 Awards: Editors' Picks," 22 December 2011. [Online]. Available: http://www.1up.com/features/1up-2011-awards-editors-picks?pager.offset=2. [Accessed 21 January 2013].

[58] C. Brom and A. Abonyi, "Petri Nets for Game Plot," in *Proceedings of AISB artificial intelligence and simulation behaviour convention*, Bristol, 2006.

[59] C. Brom, V. Šisler and T. Holan, "Story Manager in 'Europe 2045' Uses Petri Nets," *Virtual Storytelling. Using Virtual Reality Technologies for Storytelling,* pp. 38-50, 2007.

[60] C. J. Pickett, C. Verbrugge and F. Martineau, "(P)NFG: A Language and Runtime System for Structured Computer Narratives," McGill University, Montreal, 2005.

[61] F. Martineau, PNFG: A Framework for Computer Game Narrative Analysis, Montreal: McGill University, 2006.

[62] C. Verbrugge, "A Structure for Modern Computer Narratives," McGill University,

Montreal, 2003.

[63] J. A. van der Poll, P. Kotzé, A. Seffah, T. Radhakrishnan and A. Alsumait, "Combining UCMs and Formal Methods for Representing and Checking the Validity of Scenarios as User Requirements," in *Proceedings of the 2003 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology*, 2003.

[64] J. Whittle and P. K. Jayaraman, "Synthesizing Hierarchical State Machines from Expressive Scenario Descriptions," *ACM Transactions on Software Engineering and Methodology,* vol. 19, no. 8, pp. 1-45, 2010.

[65] J. Levin, "Modeling in Software Architecture," Ottawa-Carleton Institute for Computer Science, Ottawa, 2009.

[66] R. Buhr, D. Amyot, M. Elammari, D. Quesnel, T. Gray and S. Mankovski, "High Level, Multi-Agent Prototypes from a Scenario-Path Notation: A Feature-Interaction Example," in *PAAM'98, Third Conference on Practical Application of Intelligent Agents and Multi-Agents*, London, UK, 1998.

[67] A. Alsumait, A. Seffah and T. Radhakrishnan, "Use Case Maps: A Visual Notation for Scenario-Based User Requirements," in *Proceedings of the 10th International Conference on Human-Computer Interaction*, 2003.

[68] M. Nakamura, P. Leelaprute and T. Kikuno, "Deriving Interaction-Prone Scenarios in Feature Interaction Filtering with Use Case Maps," in *Proceedings of the Seventh International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, 2002.

[69] E. A. Billard, "Patterns of Agent Interaction Scenarios as Use Case Maps," *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics,* vol. 34, no. 4, pp. 1933-1939, 2004.

[70] D. Callele, E. Neufeld and K. Schneider, "Emotional Requirements in Video Games," in *Requirements Engineering, 14th IEEE International Conference*, 2006.

[71] R. Arrabales, A. Ledezma and A. Sanchis, "Towards Conscious-like Behavior in Computer Game Characters," in *IEEE Symposium on Computational Intelligence and Games*, 2009.

[72] D. Llansó, M. A. Gómez-Martín, P. P. Gómez-Martín and P. A. González-Calero, "Explicit Domain Modelling in Video Games," in *Proceedings of the 6th International Conference on Foundations of Digital Games*, New York, NY, 2011.

[73] J. Torrente, Á. del Blanco, G. Cañizal, P. Moreno-Ger and B. Fernández-Manjón, "<e-Adventure3D>: An Open Source Authoring Environment for 3D Adventure Games in Education," in *Proceedings of the 2008 International Conference on Advances in Computer Entertainment Technology*, New York, NY, 2008.

[74] M. Shiri, J. Hassine and J. Rilling, "Feature Interaction Analysis: A Maintenance Perspective," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, New York, NY, 2007.

[75] R. Buhr, "Use Case Maps as Architectural Entities for Complex Systems," *IEEE Transactions on Software Engineering,* vol. 24, no. 12, pp. 1131 -1155, 1998.

[76] R. Buhr, "Making Behaviour a Concrete Architectural Concept," in *Proceedings of the 32nd Annual Hawaii International Conference on Systems Sciences*, 1999.

[77] G. Mussbacher, D. Amyot and M. Weiss, "Visualizing Aspect-Oriented Requirements Scenarios with Use Case Maps," in *First International Workshop on Requirements Engineering Visualization*, 2006.

[78] H. Handa, "Dimensionality Reduction of Scene and Enemy Information in Mario," in *IEEE Congress on Evolutionary Computation*, Okayama, Japan, 2011.

[79] S. Bakkes, P. Spronck and J. van den Herik, "Rapid and Reliable Adaptation of Video Game AI," *IEEE Transactions on Computational Intelligence and AI in Games,* vol. 1, no. 2, pp. 93-104, 2009.

[80] H. Abd El-Sattar, "A Novel Interactive Computer-Based Game Framework: From Design to Implementation," in *Visualisation, 2008 International Conference*, 2008.

[81] A. Amanatiadou, "Extending the Reference Method for Game Production: A Situational Approach," in *Conference in Games and Virtual Worlds for Serious Applications*, 2009.

[82] J. Torrente, Á. del Blanco, P. Moreno-Ger, I. Martínez-Ortiz and B. Fernández-Manjón, "Implementing Accessibility in Educational Videogames with <e-Adventure>," in *Proceedings of the first ACM international workshop on Multimedia technologies for distance learning*, New York, NY, 2009.

[83] R. M. Weerakoon, P. Chundi and M. Subramaniam, "Dynamically Adapting Training Systems Based on User Interactions," in *Proceedings of the 2011 workshop on Knowledge Discovery, Modeling and Simulation*, New York, NY, 2011.

[84] jUCMNav, "jUCMNav: Juice up your modelling!," 13 September 2012. [Online]. Available: http://jucmnav.softwareengineering.ca/ucm/bin/view/ProjetSEG/WebHome. [Accessed 21 March 2013].

[85] D. Amyot, "UCM Scenarios and Path Traversal," SG17, Geneva, 2002.

[86] ITU-T, Z.151 : User Requirements Notation (URN) - Language definition, Geneva, 2012.

[87] A. Miga, Application of Use Case Maps to System Design With Tool Support,

Ottawa: Carleton University, 1998.

[88] J.-F. Roy, J. Kealey and D. Amyot, "Towards Integrated Tool Support for the User Requirements Notation," in *System Analysis and Modeling: Language Profiles*, Springer, 2006, pp. 198-215.

[89] The jQuery Foundation, "jQuery," 2013. [Online]. Available: http://www.jquery.com/. [Accessed 25 February 2013].

[90] C. Evans, "jCanvas," [Online]. Available: http://calebevans.me/projects/jcanvas/index.php. [Accessed 25 February 2013].

[91] W3C, "HTML5 differences from HTML4," 5 April 2011. [Online]. Available: http://www.w3.org/TR/2011/WD-html5-diff-20110405/. [Accessed 25 February 2013].

[92] HTML5games.com and Aspidoff Technologies, "HTML5games.com," 2011. [Online]. Available: http://html5games.com/. [Accessed 25 February 2013].

[93] D. Galeano and D. Tebbs, "Making the Move to HTML5, Part 2," Gamasutra, 21 February 2013. [Online]. Available: http://www.gamasutra.com/view/feature/187014/Making_the_Move_to_HTML5_Part_2.php. [Accessed 25 February 2013].

[94] Little Workshop, "BrowserQuest," [Online]. Available: http://browserquest.mozilla.org/. [Accessed 25 February 2013].

[95] YoYo Games Ltd., "GameMaker: Studio™ Master Collection," 2013. [Online]. Available: http://www.yoyogames.com/gamemaker/studio/master. [Accessed 25 February 2013].

[96] Mozilla Developer Network, "Array slice method," 4 December 2012. [Online]. Available: https://developer.mozilla.org/en-

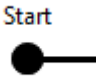
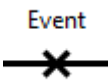
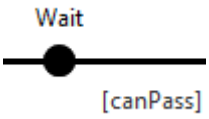
US/docs/JavaScript/Reference/Global_Objects/Array/slice. [Accessed 23 March 2013].

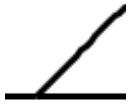
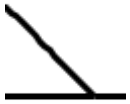
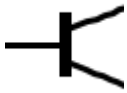
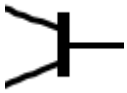
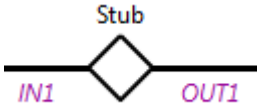
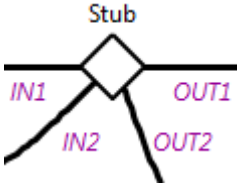
[97] Mozilla, "Firebug and Logging : Firebug," [Online]. Available: https://getfirebug.com/logging. [Accessed 17 March 2013].

[98] stackoverflow, "What's the minimum lag detectable by a human?," 30 July 2011. [Online]. Available: http://stackoverflow.com/questions/6880856/whats-the-minimum-lag-detectable-by-a-human. [Accessed 17 March 2013].

Appendices

Appendix A Types of Nodes Supported from Use Case Maps

		
Start Point	**End Point**	**Responsibility / Event**
		
Waiting Place + Condition	**Direction Arrow**	**Empty Point**

	
Or-Fork (multiple outgoing connections)	**Or-Join (multiple incoming connections)**
	
And-Fork (mult. outgoing connections)	**And-Join (mult. incoming connections)**
	
Stub	**Stub (mult. incoming / outgoing connections)**

Appendix B Detailed Results of Playthroughs for *Dungeon Explorer*

The following tables list the sequence of event calls taken for five different playthroughs (i.e., iterations) of *Dungeon Explorer*. Each playthrough / iteration recorded the sequence of events and time (in milliseconds) per legal event call to determine the new set of legal events. At the end of each table, we have included the average time per legal event call, maximum time, minimum time, and adjusted average time (rounding 0ms times to 1ms, for more-accurate results).

Iteration 1 / Events	Time (ms)	Iteration 2 / Events	Time (ms)
Instructions	1	Instructions	1
Collect Coin	2	Collect Coin	1
First Switch	1	First Switch	2
Collect Coin	2	Collect Coin	2
Or Switch B	2	Or Switch A	1
Collect Coin	2	Exit Or-A Path	1
Collect Coin	2	Collect Coin	6
Collect Coin	2	Collect Coin	1
And-Switch 1	1	Collect Coin	1
Collect Coin	1	And-Switch 1	1
Collect Coin	1	Collect Coin	2
Collect Coin	3	Collect Coin	2
And-Switch 3	1	Collect Coin	2
And-Switch 2	0	And-Switch 3	0
Collect Coin	2	And-Switch 2	1
Collect Coin	2	Collect Coin	1
Found All Coins	0	Collect Coin	1

End Game	0	Found All Coins	1
		End Game	0
Average	1.39	**Average**	1.5
Maximum	3	**Maximum**	6
Minimum	0	**Minimum**	0
Adjusted Average	1.56	**Adjusted Average**	1.56

Iteration 3 / Events	**Time (ms)**	**Iteration 4 / Events**	**Time (ms)**
Instructions	1	Instructions	1
Collect Coin	1	Collect Coin	1
First Switch	2	First Switch	1
Collect Coin	2	Collect Coin	1
Or Switch A	0	Or Switch B	2
Exit Or-A Path	4	Collect Coin	3
Collect Coin	1	Collect Coin	2
Collect Coin	1	And-Switch 1	1
And-Switch 2	2	Collect Coin	2
Collect Coin	2	Collect Coin	2
And-Switch 3	1	Collect Coin	1
Collect Coin	2	And-Switch 2	2
Collect Coin	2	And-Switch 3	1
And-Switch 1	1	Collect Coin	1
Collect Coin	1	Collect Coin	1
Collect Coin	1	Collect Coin	1

Collect Coin	1	Found All Coins	0
Found All Coins	1	End Game	0
End Game	0		
Average	1.44	**Average**	1.28
Maximum	4	**Maximum**	3
Minimum	0	**Minimum**	0
Adjusted Average	1.5	**Adjusted Average**	1.39

Iteration 5 / Events	**Time (ms)**
Instructions	1
Collect Coin	2
First Switch	2
Collect Coin	2
Or Switch A	1
Exit Or-A Path	2
Collect Coin	4
Collect Coin	2
Collect Coin	2
And-Switch 1	2
Collect Coin	2
Collect Coin	1
And-Switch 3	1
Collect Coin	2
And-Switch 2	1

Collect Coin	1
Collect Coin	2
Found All Coins	0
End Game	0
Average	1.67
Maximum	4
Minimum	0
Adjusted Average	1.72