Use Case Map Traversal against Sequence Breaking

Matthew Shelley and Jean-Pierre Corriveau
School of Computer Science
Carleton
Ottawa, Canada
jeanpier@scs.carleton.ca

Wei Shi
Faculty of Business and I.T.
University of Ontario, Institute of Technology
Oshawa, Canada
Wei.Shi@uoit.ca

***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 honored, as per the designers' intentions. We postulate that sequence breaking most often arises through bypassing geographic barriers, cheating, and misunderstanding on the player's behalf. In this paper, 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 a testing tool, and then show its feasibility through a game we created, concluding that our solution is both sufficient and feasible.**

Keywords—use case maps; sequence breaking; narrative; video games; traversal

I. INTRODUCTION

Sequence Breaking is a subset of *Feature Interaction*, which is a well-known problem in the field of Computer Science [1, 2, 3]. In short, a feature conflict is said to have occurred when there is "unwanted interference [among] two [or more] features," [1] where a feature is a unit of specific, verifiable functionality that provides value to the end-user of an application [4]. In the realm of video games, such conflicts arise when two or more features lead to 'undesirable behavior' including unwinnable situations, visual anomalies, and 'inaccurate' storylines (e.g. earning rewards earlier than intended).

Sequence breaking exists in the domain of game narrative when a predefined storyline is not followed as per the game designers' intentions. When a 'narrative' feature, hereafter referred to as an *event*, is called outside of the game's set of predefined narrative sequences, there exists unwanted interference with the storyline—the game's subsuming 'narrative' feature—as its integrity has not been honoured. When the player starts an invalid sequence of events, they are breaking the predefined narrative sequence.

Video games have been subject to sequence breaking since their inception. In 1986, Enix's *Dragon Quest* expected the player to rescue a princess to acquire an item, but the item could be found early in the game effectively skipping the rescue entirely [5]. In 1994, Sega's *Sonic 3 & Knuckles* inadvertently allowed the player to fly over some mid-boss areas, to skip such battles; doing so would later cause glitches [5]. In 2011, Nintendo's *The Legend of Zelda: Skyward Sword* became impossible to finish if players completed tasks in a certain order [6]. In summary, sequence breaking conflicts have existed for decades and continue to exist to this day.

Such conflicts are detrimental to players in two ways. First, as skilled players are often the ones to perform sequence breaking, for the sake of cheating or their own enjoyment, other players may fall victim to such unfair advantages. For example, in *Pokémon Red Version* and *Blue Version*, the player could follow an unusual sequence to battle a glitch Pokémon, which caused items to be duplicated [7]. Second and worse, when sequence breaking occurs without the player's knowledge, the game's story may make no sense, leading to confusion, or the game may become unwinnable. Because the player's experience is reduced, sequence breaking poses a significant problem for an industry that relies on creating 'fun' in order to sell its products.

In this paper, we provide a solution to preventing sequence breaking, verify its behaviour, and then explore its feasibility. In section 2, we first begin with a short example to further clarify what is meant by sequence breaking. Second, we define terms that will be used throughout this paper. We then briefly discuss how we choose to represent the elements of a narrative. In section 3, we overview our solution to preventing sequence breaking and present the details of our algorithm in the next section. Then, in section 5, we discuss verification and feasibility of our solution before concluding in section 6.

II. BACKGROUND

A. A Short Example

First, suppose a player is required to explore a dungeon and defeat a boss 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.

This setup expects the sequence:

a) Enter Dungeon

978-1-4799-2030-3 /14/$31.00 ©2014 IEEE

b) Fight Boss to Receive Key Item

c) Exit Dungeon, Unable to Return

d) 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, resulting in an unwinnable situation as the player cannot return to acquire the missed key item.

This outcome gives the actual sequence:

a) Enter Dungeon

b) Exit Dungeon, Unable to Return

c) Unable to Enter Next Dungeon

The player cannot proceed in any way, thus preventing the intended experience: sequence breaking has made the game unplayable.

B. Definitions

Table 1 defines common terms that will be used in this paper.

TABLE I. LIST OF TERMS WITH DEFINITIONS

Term	**Definition**
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 command temporarily, moving entities, etc.
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	an uninterruptable script that serves the purpose of furthering the story when called and requires additional legality checking to ensure that it is performed in correct sequence (as specified by the designer).
Event Identifier	an integer or string that uniquely identifies an event.
Geography	a medium (such as the game world containing towns, dungeons, and paths) in which entities reside, and where the player moves characters 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 can be called next based on the player's progress within the narrative representation.
Narrative	a set of pre-determined events and of event sequences, where an event sequence 'moves' the player through a game's story arc or intended means of progression.
Narrative Representation	a knowledge representation, which describes valid event sequences in a narrative, such as a diagram or graph.
Progress	a set of 'positions' (i.e., indicators assigned to 'nodes' within a narrative representation) that serves to record events that have recently occurred in addition to 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 between lines), conditional branching, and comments.
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.

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, resulting in an unwinnable situation as the player cannot return to acquire

C. On Narrative Elements

A narrative element of a game "communicates aspects of [the] story to the player" [8]. 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 denoted using events.

Two forms of narrative can be present in any game: Embedded narrative is "pre-generated narrative content that exists prior to a player's interaction with the game," 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" [*Ibid*]. Emergent narrative, alternatively, is the player's experience with the game that can "[vary] from session to session, depending on [the] user's actions" [*Ibid*]. While a good "game design involves employing and balancing the use of these two elements," [*Ibid*] the focus of our work is purely on embedded narratives. In fact, we further restrict our scope here to single-player adventure-type games since such games center on narratives but allow only for a simple form of sequence breaking (in contrast to massively-multiplayer online games). Majewski [9] discusses different approaches for creating

embedded game narratives. In all these, a design specifies valid sequences of events in order to build a story.

Some solutions to sequence breaking have been attempted. For example, Eladhari [10] proposes the creation of causal relationships between events, as well as "Object Oriented Story Construction". The latter requires that all entities of a game be given knowledge of the game's story in order to "make them more intelligent with respect to the overall narrative goals". The main disadvantage with existing research in narratives however remains that it is essentially theoretical: existing proposals are not implemented in an actual game. Consequently, unfortunately, the verification of a computational version of such proposals is generally not addressed.

In [11], we review at length work on game narrative, feature interaction, game development, and game testing. We also discuss five notable game titles[1] from between 1992 and 2011 in order to understand their means of handling narrative progression. We identify the following mechanisms to control the traversal of a narrative:

a) *Geographic barriers:* which refer to obstacles placed within the geography to 'physically' prevent the player's character from progressing. Such barriers may be removed either through gaining a new skill or item, or by the game itself after an event.

b) *Narrative barriers,* on the other hand, require some condition to be true before they can be overcome.

c) *Central hubs* may be common geography the player visits in order to access different areas of the game. A world map is an example.

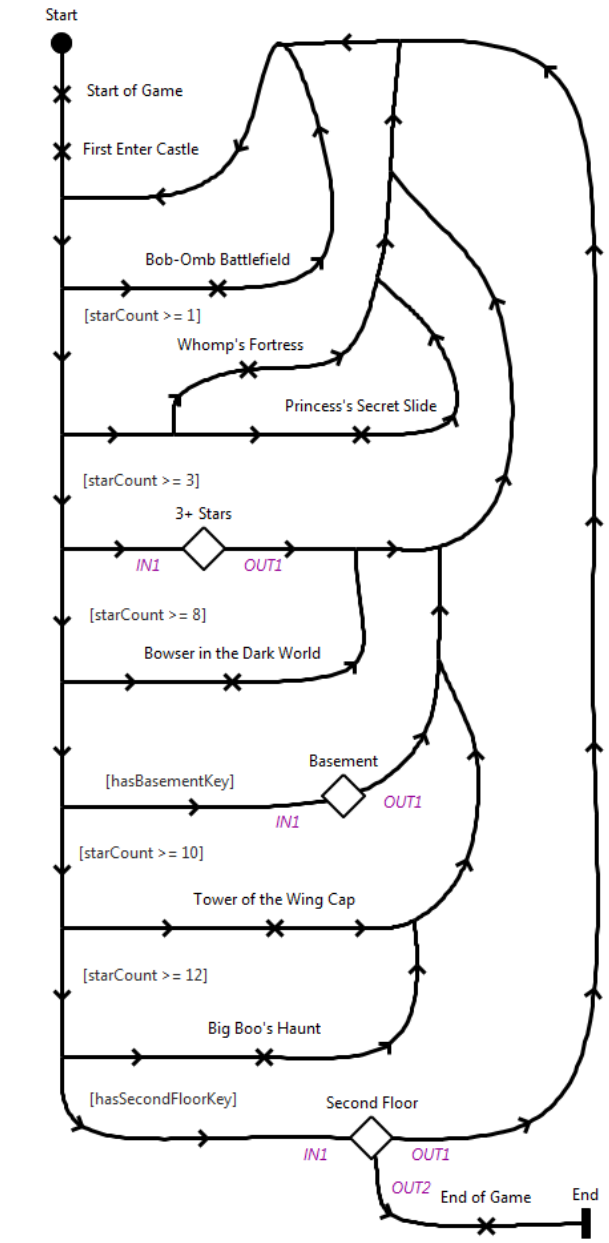
Narrative barriers are significantly more difficult to break than geographical barriers, while central hubs are 'transitions' between events. In general, video games make use of event sequences to represent narrative progression along with geographic and narrative barriers to enforce the intended progression. Our proposal caters to this approach by allowing constraints (such as preconditions) on events and by handling both entities and triggers within geography.

Our study of these five games also suggests that, typically, the narrative structure of single-player adventure-type game can be captured through a small set of 'scenario elements' found in many scenario notations such as Petri Nets [12] and Use Case Maps [13] (hereafter UCMs). Given our familiarity with the latter, we support this claim by creating a UCM representation of the narrative of each of these five games (to be used, among others, as test cases for our approach to sequence breaking). As an example, we show the UCM for the top level and one sublevel of Super Mario 64 in Figure 1.

The claim that representing the narrative of single-player game only requires a small set of 'scenario elements' is further supported by the work of Martineau [14] who develops a high level computer narrative language, the Programmable Narrative Flow Graph (PNFG), built atop Narrative Flow Graphs (NFG), which are simple Petri Nets. (His work is however limited to only purely textual games.)

[1] namely: The Legend of Zelda: A Link to the Past, Super Mario 64, Pokémon: SoulSilver Version and HeartGold Version, Grand Theft Auto 4 and Bastion

a) top level



b) 3+ star subdiagram

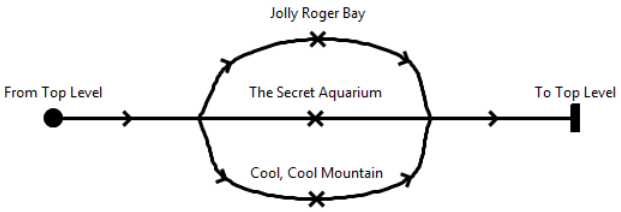


Figure 1. Partial UCMs for Super Mario 64

The small set of scenario elements we use is given in Figure 2. (in order to enable the straightforward interpretation of Figure 1.). It is further discussed in section 4 where we discuss the details of our proposal.

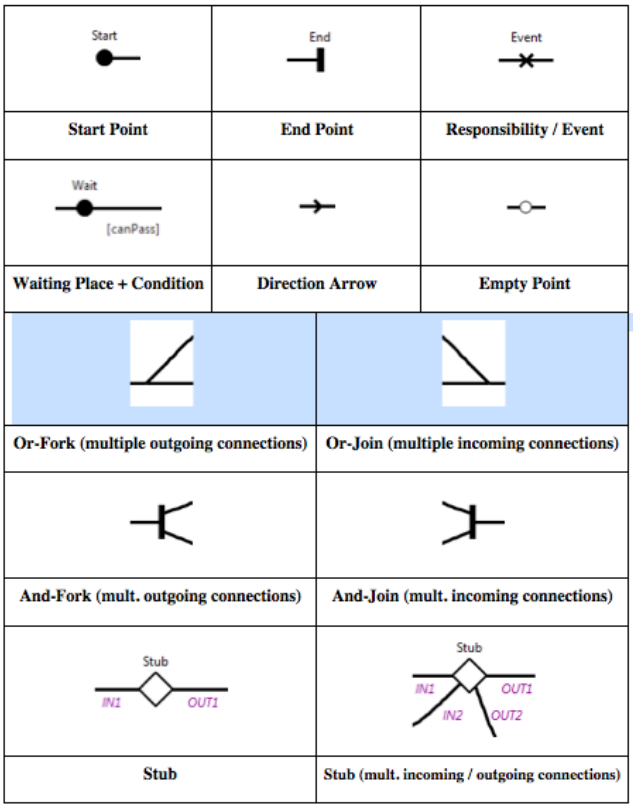


Figure 2. UCM Elements used for Narrative Representation

Let us first overview the latter.

III. Overview of Our Solution

In this paper, we propose a 'narrative manager' and traversal algorithm in order to prevent sequence breaking within a game environment at run-time. We represent a set of valid narrative sequences using UCMs; keep track of the player's progress along such a representation; and then, check if attempted event calls are legal using this combined knowledge. The narrative manager stores the narrative representation, player's progress, and set of legal events. This manager can determine if attempted event calls are legal by checking against the legal set of events. The traversal algorithm accepts an event identifier for a legal event, and then updates both the player's progress and the set of next legal events. We can prevent sequence breaking by rejecting any attempted calls to illegal events.

It is important to note that since the nature of sequence breaking involves the unintended sequential ordering of events, it cannot be assumed that any arbitrary event can be guaranteed to not be called. Instead, we have to assume that any event can attempt to be called regardless of the player's progress. Therefore, it seems to be necessary to validate at least $O(n^2)$ sequences, where *n* is the number of events, prior to the game's execution (e.g. at design-time or during play-testing), but such validation is intractable. It is unreasonable to expect designers or play-testers to consider an exponential number of sequences as such an approach simply does not scale. Thus, it is necessary in our opinion to offer a run-time solution. Indeed, instead of testing and resolving (not only pairwise but) all sequences of events to ensure only valid sequences are allowed, it is preferable to create a means of checking if a requested event is legal based on the player's progress within the storyline at a given point in the game. When an attempted event call is deemed to be legal, it may proceed as expected; otherwise, the event call is rejected (and an appropriate warning can be given to the player). In this approach, designers need only consider valid sequences, while game testers verify that this preventative procedure works.

We believe sequence breaking to be preventable at run-time when, in a time undetectable by the player and within a game environment, we can perform the following routine, given an event identifier, narrative representation, and the player's progress:

a) Determine if the event associated with the given event identifier is legal to call at this point of the narrative.

b) If 'yes', return a 'success' value, call the associated event, update the player's progress, and then:

- Update the game's world to allow the player to call events that are now considered legal; and,
- Update the game's world to prevent now-illegal events from being called, effectively preventing the player from accessing illegal events.

c) If 'no':

- Ignore the attempted event call; and,
- Return a 'failure' value, so an optional developer-defined function can be triggered to handle the detected sequence breaking, possibly by using the current set of legal events.

Furthermore, our run-time approach to sequence breaking must work within the limitations of a game's environment *without* hindering the player's experience, as we discuss in section 5.

IV. Details

The major contribution of our work is a narrative manager that stores a narrative representation, updates progress, determines the set of currently legal events, and checks the legality of attempted event calls, with the goal of preventing sequence breaking. We also propose a traversal method for a UCM-based narrative, the *Royal Pegasi Algorithm*, to determine the current set of legal events based the player's progress and a given identifier for a legal event. Let us elaborate

A. Narrative Manager

The narrative manager offers three public methods, which are used for initialization and attempting to call events. The

init() method loads a Use Case Map file, stores narrative variables with their designer-defined default values, and optionally reads a file containing scripts for events. The tryToCallEventById() and tryToCallEventByName() methods, which respectively take in an event index and event name, check if the associated event is legal to call, by checking if it exists in the set of legal events, and then call the event if it is deemed legal. Using these methods, it becomes possible for a designer to store a sequence representation and check the legality of attempted event calls at run-time.

Adding to this ability, when an event is called, we preload events that have become legal and unload any events that have become illegal. To preload an event means that we add elements (entities or triggers) to the geography in order for the event to be callable by the player. For instance, the player may need to interact with a non-playable character to call an event, and thus this character will be placed in the world. To unload an event means that we remove elements from the geography in an effort to prevent calls to it. In order for our approach to run, developers provide code for preloading and unloading events. With this code, we not only try to prevent calls to illegal events, but inherently update the game's world to better serve developers from a narrative perspective.

B. Royal Pegasi Algorithm

1) An Example

The *Royal Pegasi Algorithm* serves the purposes of traversing a UCM to determine the current set of legal events. When the narrative manager is initialized, the first set of legal events is determined by following the start point(s) specified by designers. Subsequently, the legal event set is updated whenever an event is called by moving forward along the diagram, based on 'markers' referred to as Royal Guards and Pegasi, hence the chosen name. Royal Guards only traverse a UCM as far as can be certain, by relying on connections that are guaranteed to be crossable (i.e., traversable). For instance, Or-Forks (see Figure 2) may have multiple valid outgoing connections, but there is no certainty as to which ones will be crossed. Pegasi instead serve as speculative markers, which belong to either a Royal Guard or a Pegasus, in that they are created to explore paths that could *potentially* be crossed. In essence, a Royal Guard serves as a root node to a tree of Pegasi. Intuitively, our *Royal Pegasi Algorithm* can be conceptualized as a modified version of the traversal method proposed by Amyot [15] for UCMs. Details of the implementation of this algorithm can be found in [11]. Let us now instead illustrate it with an example. More precisely, in order to illustrate some of the complexity 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 given in Figure 3. 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..

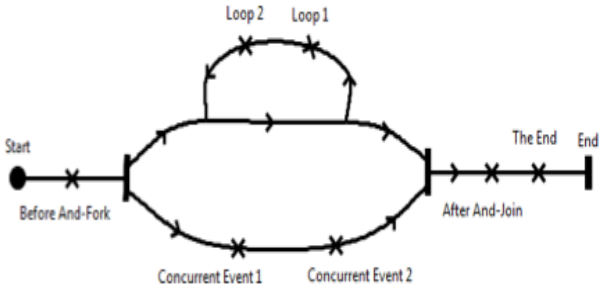


Figure 3. A small example

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 behavior 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.

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 UCM has ended and no more legal events remain. Presumably, the game has come to an end as there is no more narrative to manage.

2) Traversing a UCM

The narrative representation stored within a specification diagram of a UCM 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 behavior, require distinction, by performing a check using *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 (see next subsection) 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 behavior 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 behavior 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 behavior, 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 behavior 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) About Events

Events serve the purpose of furthering the story within a game through uninterruptable scripts when called. They 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 UCMs directly refer to events and are used to create them, with the responsibility label becoming the event name. 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 by game programmers: our solution merely calls such a programmer-supplied 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 ignore expressions assigned to responsibilities in favor 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 as supported by the jUCMNav tool[16]. 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. 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.

Finally, we remark that in creating a UCM 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 captured. As a result, the designer of a UCM for a narrative 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

V. VERIFICATION AND RESULTS

In this section, we address the procedures used to both verify our solution and check its feasibility, and then discuss the results of these methods. With respect to verification, we challenged our solution against a set of test cases that cover the UCM subset that we support, our narrative manager, and our *Royal Pegasi Algorithm*. With respect to feasibility, we created a simple game where the player proceeds through a sequence of events that must be followed in an intended order, while being permitted to cheat in attempt to break the sequence. We start by describing our test procedures in greater depth, and then discuss our results.

A. Descriptions of Experiments

Our experimental procedures apply to both the verification of our solution and its feasibility for preventing sequence

breaking at run-time. For the former, we created a webpage where numerous test cases (i.e., UCMs) can be traversed to verify our expected behavior. For the latter, we created a game to show that our solution runs in a time undetectable by the player with the proper environment. We detail these procedures to better provide an understanding of our experiments.

Verification of our solution is handled through a testing tool[2] (here a web page) shown in figure 4, which allows for a traversal of one of a set of test cases that *cover* the UCM semantics that are supported by our solution. Upon selecting an example, the tester can view the associated UCM (and its sub-diagrams, if any), try to call arbitrary events, view the current legal set of events, and call an event from the current legal set. As the tester proceeds through a test case, by calling a legal event (either by clicking an event identifier or entering one in the specified textbox), the legal set of events updates, allowing other events to be called until no more remain. Any scripts associated with events are also run (e.g., to increment variables). When an illegal event is attempted, through the textbox, a notification of sequence breaking will be shown. One can verify the legal set of events generated by following a sequence of events against the expected behaviour, by examining the provided diagrams. We have created enough test cases to cover the supported UCM semantics *and* the specifics of our narrative manager and its *Royal Pegasi Algorithm*.

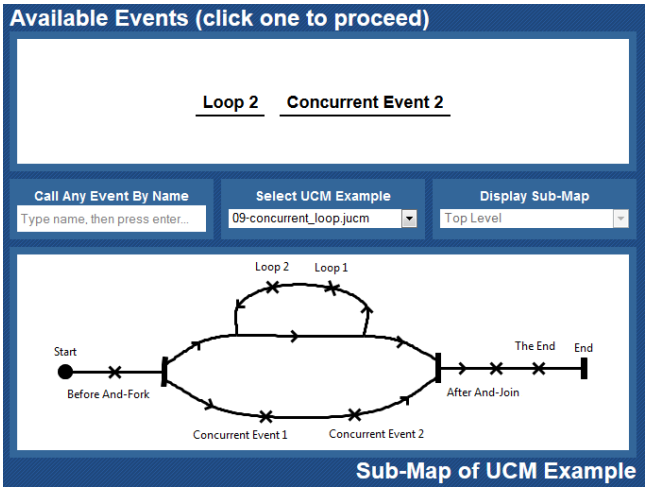


Figure 4. Verifying our solution through test cases

Feasibility of our solution is demonstrated through a game we created called *Dungeon Explorer*[3] (see figure 5). In this game, the player navigates a dungeon with limited light to collect coins, which will unlock the exit when all have been found. Torches may also be picked up to increase the visible area around the character.

As the player progresses they will activate switches to unlock more areas of the dungeon, granting access to more coins and switches. We created a UCM to represent the valid sequences of events (e.g., switch presses) that can be followed by the player. Cheats are provided to optionally allow for: walking through barriers, attempting to activate disabled switches, and viewing the entire map. If the player tries to sequence break, by stepping on a switch out of order or exiting the dungeon before collecting all coins, the game will detect the conflict and then move the player's character to resolve it.

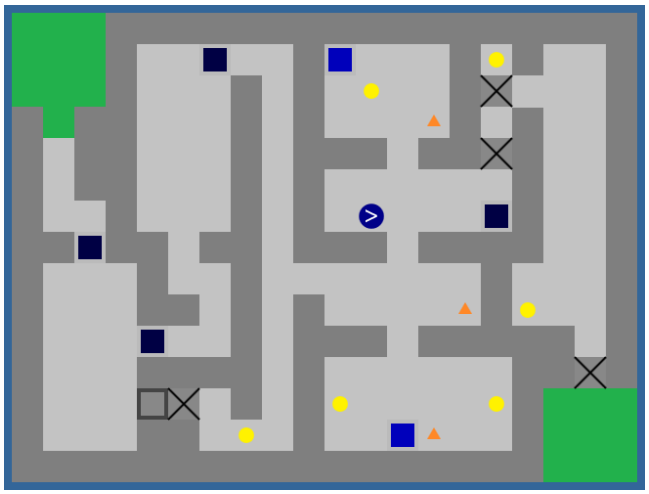


Figure 5. Screenshot of Dungeon Explorer

This game demonstrates that our solution can prevent sequence breaking at run-time without hindering a player's experience. Together, the testing tool we have created and *Dungeon Explorer* serve to verify the expected behavior of our solution and demonstrate its feasibility, respectively.

B. *Verification of Solution*

In order to verify our solution, we took several steps. First, we compiled a list of requirements to satisfy, by referring to the intricacies of our solution and the UCM semantics that we support. Second, we created UCMs as test cases to support these requirements. Third, we iteratively verified the actual behavior of each test case against its expected behavior. When a problem was detected with a test case, we resolved the problem, and then started testing all over from the first test case. We continued this process until all test cases passed. In all, we created UCMs for 19 specific examples, and 3 structures from commercial games for added credibility, as well as our *Dungeon Explorer* game.

For the sake of brevity, we will not detail each requirement we aimed to satisfy or provide an example of each requirement being satisfied, as such information is available in [11].

C. *Feasibility of Solution*

Beyond verifying the behavior of our solution, it is important to illustrate its ability to prevent sequence breaking within a game environment at run-time. For this reason, we created *Dungeon Explorer*, which included a UCM that covered all of our supported features, and then instrumented a timer into each update caused by a call to a legal event [11], which includes preload and unload callbacks. (As we are able

[2] The testing tool may be viewed at: http://www.scriptedpixels.com/content/mcs-thesis/ucm-testing-tool.htm

[3] *Dungeon Explorer* may be played at: http://www.scriptedpixels.com/content/mcs-thesis/dungeon-explorer.htm

to detect sequence breaking in O(1) time, by checking if a key exists within an array, it is not necessary to count the time for illegal calls as the legality check is negligible.) Over several playthroughs, we recorded the average update time, adjusted average update time (where updates less than 1 ms are rounded up), maximum update time, and minimum update time. Our feasibility results are recorded in Table 3:

TABLE II. PERFORMANCE OF ALGORITHM IN DUNGEON EXPLORER

Iteration / Statistic	It. 1	It. 2	It. 3	It. 4
Average	1.39 ms	1.5 ms	1.44 ms	1.28 ms
Adjusted Average	1.56 ms	1.56 ms	1.5 ms	1.39 ms
Maximum	3 ms	6 ms	4 ms	3 ms
Minimum	<1 ms	<1 ms	<1 ms	<1 ms

From these results, it is apparent that our average update time is small, while the maximum update time was at most 6 ms. Even if our solution were to run at 6 ms for every update call, such a time would still not be detectable by a human being – as evidenced by the fact that computer monitors have refresh rates between 60 and 120 Hz (or 8 ms to 16 ms per frame). To add to this point, our algorithm runs in a separate thread and only when a legal event is called: a relatively uncommon occurrence in comparison to the number of frames where a legal event is not called. Thus, it is reasonable to suggest that our solution is sufficiently efficient as the time to preload events, call a legal event, unload events, and determine the new set of legal events is not significant enough to disrupt the player's experience.

VI. CONCLUSIONS

In this paper, we presented an approach for managing a game's narrative for the purpose of preventing sequence breaking at run-time. From reviewing the literature and studying five popular games, we compiled a list of narrative elements that appear to be necessary and sufficient to represent storylines. We then observed that a subset of Use Case Maps (UCMs) could readily capture these narrative elements and represent the valid sequences of a narrative (in order to compare a player's progress against the designer's set of valid storylines). We created a narrative manager and a UCM traversal algorithm in order to monitor the player's progress and prevent sequence breaking. We then developed an extensive set of test cases that address all the UCM elements we use as well as cover all the branches of our algorithm. Finally, we developed a game to demonstrate the feasibility of our proposal.

One derived benefit of our solution is that, having access to the current legal set of events, a designer may draw on this knowledge to have the game offer better context-sensitive behavior (such as improved dialogues, better behavior for non-player characters, etc.). In particular, we explain elsewhere [11] how, beyond preventing sequence breaking by ignoring attempted calls to illegal events, we try to reduce such calls by altering the game world to exclude related triggers or entities.

There is however presently a drawback in our approach: Because the player's progress is currently only updated when a legal event is called, it is possible for events that depend on conditions to not be 'moved' to the legal set before another event has been triggered. Similarly, an event that has become illegal based on a precondition may remain in the legal set of events. A solution would require rethinking the current handling of narrative-related variables. At this point in time, it is not clear whether such rethinking should occur only in the limited scope of single-player adventure-type games that we have chosen, or consider the complexities that will be unavoidably introduced if we tackle massively-multiplayer online games.

REFERENCES

[1] S. Tsang and E. H. Magill, "Detecting feature interactions in the intelligent network," in L. G. Bouma and H. Velthuijsen (editors), Feature Interactions in Telecommunications Systems. IOS Press, Amsterdam, pp. 236–248, 1994.

[2] Y. Jia and J. M. Atlee, "Run-time management of feature interaction," ICSE Workshop on Component-Based Software Engineering (CBSE6), Porland (Oregon), May 2003.

[3] ICFI 2012, "About ICFI," 29 July 2011. [Online]. Available: http://www27.cs.kobe-u.ac.jp/wiki/icfi2012/index.php?AboutICFI [Accessed 21 May 2014].

[4] K. Czarnecki and U. Eisenecker, Generative Programming, Addison-Wesley, New York, 2000.

[5] tvtropes.org, "Sequence breaking," 9 May 2012. [Online]. Available: http://tvtropes.org/pmwiki/pmwiki.php/Main/SequenceBreaking [Accessed 21 April 2014].

[6] 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 21 April 2014].

[7] IGN, "Pokemon report: OMG hacks," 24 November 2008. [Online]. Available: http://ds.ign.com/articles/933/933126p1.html [Accessed 21 April 2014].

[8] J. Whitehead, February 26 2007. [Online]. Available: http://classes.soe.ucsc.edu/cmps080k/Winter07/lectures/narrative.pdf [Accessed 21 April 2014].

[9] J. Majewski, Theorising Video Game Narrative, Master's thesis, Centre for Film, Television & Interactive Media, School of Humanities & Social Sciences, Bond University, 2003.

[10] M. Eladhari, Object Oriented Story Construction in Story Driven Computer Games, Master's Thesis, Department of History of Literature and History of Ideas Stockholm, Sweden: Stockholm University, 2002.

[11] M. Shelley, On the Feasibility of using Use Case Maps for the Prevention of Sequence Breaking in Video Games, Master's thesis, School of Computer Science, Ottawa: Carleton University, 2013.

[12] C. Brom and A. Abonyi, "Petrinets for game plot," in Proceedings of artificial intelligence and simulation behaviour convention (AISB), Bristol, April 2006.

[13] D. Amyot and G. Mussbacher, "User requirements notation: the first ten years, the next ten years," Journal of Software, vol. 6(5), pp. 747-768, 2011.

[14] F. Martineau, PNFG: A Framework for Computer Game Narrative Analysis, Master's thesis, School of Computer Science, Montreal: McGill University, 2006.

[15] J. Kealey and D.Amyot, "Enhanced use case map traversal semantics", SDL Forum, pp.133-149, Paris, September 2007.