The Dissertation Committee for Lane Thomas Holloway
certifies that this is the approved version of the following dissertation:

Modeling and Formal Verification of Gaming Storylines

Committee:

Donald Fussell, Supervisor

Anne C. Elster, Co-supervisor

Jacob Abraham

Craig Chase

Risto Miikkulainen

Modeling and Formal Verification of Gaming Storylines

by

Lane Thomas Holloway, B.S.E.E.; M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2016

Dedicated to my family.

Acknowledgments

I wish to thank the multitudes of people who helped me. First and foremost my two advisors: Profs. Donald S. Fussell and Anne C. Elster who have put up with me and my stubbornness and taught me the black arts of academic research over what seems to be a lifetime. Next, the rest of my committee: Profs. Craig Chase, Jacob Abraham, and Risto Miikkulainen participating on the committee and providing valuable feedback.

I'd like to thank colleagues from the gaming industry whom include: Royal McGraw for insight and details into how writers think about and test storylines in production games. Rich Vogel for answering questions at the beginning of my quest toward a PhD that allowed this research to move forward. Matt Boudreaux for providing a developer's perspective on how story affects game development. Jason Frueh, a former QA developer for BioWare who helped refine and enlighten me to the software development life cycle of large games as such *Star Wars: The Old Republic*.

I'd like to thank Nadeem Malik, Walid Kobrosly, and Fred Hudson (my Master's advisor) for pushing me down the path of becoming a PhD.

Last, but not least, my family for putting up with me during this colossal undertaking.

Modeling and Formal Verification of Gaming Storylines

Publication No. _____

Lane Thomas Holloway, Ph.D.
The University of Texas at Austin, 2016

Supervisors: Donald Fussell
Anne C. Elster

Video games are becoming more and more interactive with increasingly complex plots. These plots typically involve multiple parallel storylines that may converge and diverge based on player actions. This may lead to situations that are inconsistent or impassable. Current techniques for planning and testing game plots involve naive means such as text documents, spreadsheets, and critical path testing. Recent academic research [1] [2] [3] examines the design planning problems, but neglect testing and verification of the possible plot lines. These complex plots have thus until now been handled inadequately due to a lack of a formal methodology and tools to support them.

In this dissertation, we describe how we develop methods to 1) characterize storylines (SChar), 2) define a story line description language (SDL), and 3) create a storyline verification tool based in formal verification techniques (StoCk) that use our SDL as input. SChar (Storyline Characterization) help game developers characterize the category of story line they are working on

(e.g. linear, branching and plot) through a tool that give a set of guided questions. Our SDL allows its users to describe storylines in a consistent format similar to how they reason about storylines, but in such a way that it can be used for formal verification. StoCk accepts storylines, described in SDL, to be formally verified using SPIN for errors. StoCk is also examined in three common use cases found in the gaming industry used as a tool 1) during storyline creation 2) during quality assurance and 3) during storyline implementation. The combination of SChar, SDL, and StoCk provides designers, writers, and developers a novel methodology and tools to verify consistency in large and complex game plots.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the past five years, the video game market has grown from $65 billion dollars in total revenue to $83.6 billion dollars in 2014 with a 2015 estimate exceeding $100 billion dollars[1] [2]. The cost of producing a video game is on the order of $100 million and involves teams of over 100 (sometimes up to 600) developers, artists, managers, and marketers [3] [4]. From the mainframes of the past to the video game consoles of today, games have always taken advantage of computational power for improved, refined, and realistic graphics, physics, and gameplay; however, the storylines have been left in the dark ages [4].

1.1 Motivation

The advent of massively-multiplayer online (MMO) games, the evolution of role-playing games, and the addition of role-playing elements into often linear storylines has spawned larger complex stories and worlds involv-

[1] http://www.statista.com/statistics/278181/video-games-revenue-worldwide-from-2012-to-2015-by-source/

[2] http://www.newzoo.com/insights/us-and-china-take-half-of-113bn-games-market-in-2018/

[3] http://gamers.blogs.challenges.fr/archive/2013/06/18/watch-dogs-le-tres-gros-budget-d-ubisoft.html

[4] http://documents.latimes.com/bungie-activision-contract/

ing multiple paths and choices to complete the story. However, these advances are often so complex that the game designers cannot possibly verify all possible story lines given the lack-luster tools they currently have available. My conversations with Mr. Rich Vogel, the former producer of *Star Wars: The Knights of the Old Republic* MMO for BioWare, inspired this work. He and the members of the BioWare *Mass Effect* design team along with Mr. Royal McGraw, a producer and writer at Pixelberry Studios, have all confirmed their designers currently lack the ability to manage, understand the complexity, and verify correctness of the plots they design.

1.1.1 Do players care about engaging stories?

As games such as *Mass Effect*, *The Witcher 3*, and *Fallout 4* have shown, companies are attempting to push the boundaries of interactivity in various ways: storyline, world interactivity, and scale of the world where the game takes place. However, this does not answer the question if the players themselves like or enjoy these advances. To confirm this, we created a survey that asked respondents about the current state of video games, the games they enjoy playing, and what they would think of playing games that are more dynamic in the storylines to validate the trends seen by these games.

Over two weeks, the survey (see Appendix A) received 603 responses from across the United States and a few other countries. Action, adventure, and role-playing games were the top three game types chosen and 86% of the respondents considered story to be an important part of the game experience.

The survey differentiated between static storylines and dynamic (user interactive) storylines; 95% responded that they would play a game with a dynamic storyline and 84% said they would prefer playing a game with a dynamic storyline to one with a static storyline.

When asked why they would prefer a dynamic storyline to a static storyline, users responded with three main ideas: first, dynamic storylines can allow for a more engaging experience; secondly, there is more to discover and more replayability to an interactive game; thirdly, dynamic storylines can allow players to think outside the box and allow creative thinking on how to solve problems they are confronted with during the course of the game.

Our ten questions revealed that players want more interactive stories with the ability to truly feel that their interactions meant something to the storyline. They validate the motivation behind the research and exhibit the need for tools to support the creation of such storylines. Tackling the problem facing the game developers, designers, and writers can be divided into three goals: managing design complexity, preventing inconsistencies in game plots, and integrating tools into the software development life cycle.

1.1.2 Managing Design Complexity

Tooling and robust methodologies are used by both hardware and software teams to manage the complexities of their designs. There are a few differences between the two domains in general.

In hardware design, complexity is managed using methodology and

automated tools. The methodology pushes for well-defined subsystems and well-known constraints on these systems. The automated testing tools can then verify the sub-systems and guarantee working sub-systems. However, full-system verification is still a major issue [5].

Software, on the other hand, has methodology but very limited tools. The use of formal verification for a piece of software is often impossible and various additional tools are used in their place. Software often lacks well-defined sub-systems and is not as easily constrained and understood as the hardware domain [6] [7]. Video games have large amounts of variability based upon their game type, hardware, human interactions, and stories [8]. Some of the variability is controlled through the use of game engines. However, there is still a large amount of complexity and variability associated with each game [9].

The game designer's methods and tools attempt to address this problem but are still failing [10]. In order for designers to contain the complexity, they implement ideas such as foldbacks and instance dungeons within the game. They also attempt to track all the storylines using Microsoft Excel and Microsoft Word to design, model, and version their stories and conditions for storylines to branch or open new lines of dialog for users (BioWare PAX Panel, pers. comm.). They rely on the typical software engineering processes and quality assurance (QA) testing the critical path of the storylines (R. Vogel, pers. comm.). The designers lack the proper tooling to master the complexity of non-trivial storylines and worlds in which the games occur.

1.1.3 Preventing Inconsistent Game Plots

The game industry is including more user interactivity within their stories, which can cause issues when the player does something unexpected. On a small scale or a story with very little branching these problems can be caught with typical software processes, such as quality assurance and unit testing although many inconsistencies are still missed.

An example of an inconsistent plot can be seen in the well-received Action-Role Playing Game, *Star Wars: Knights of the Old Republic*[5] [6] [7]. It exhibits quite a few instances but an encounter that occurs early on in the game is representative of multiple inconsistencies found throughout the game.

As players arrive on a new planet they are warned that two prominent families in the area (the Sandrals and the Matales) are feuding, which might overflow into a violent conflict. When speaking to the head of the Sandral estate, Nurik, he relates his sadness over the disappearance of the young Matale heir, Shen; however, he claims to know nothing about the event. Nurik also states his own son, Casus, has been missing for some time and speculates the two might have met similar fates. Nurik then dismisses the player.

Shortly thereafter, the user meets Nurik's daughter, Rahasia. Rahasia reveals that her father has kidnapped Shen because she and Shen were in love.

---

[5] BioWare, *Star Wars: Knights of the Old Republic*, Console, 2003.

[6] `https://grandtextauto.soe.ucsc.edu/2008/02/06/ep-33-an-example-star-wars-knights-of-the-old-republic/`

[7] `http://www.metacritic.com/game/xbox/star-wars-knights-of-the-old-republic`

The player receives a key to rescue Shen. Once Shen is rescued, he refuses to leave without Rahasia, which leads to a confrontation between both families' patriarchs and their battle droids. After some discussion, the two lovers run off to safety.

Later, while exploring the planet, the player comes upon the Matale compound where the guard droid grants the player an audience with the patriarch (who had last watched his son, Shen, elope with Rahasia Sandral). He then asks the player to find his son who he believes was kidnapped by the Sandral family.

A second example occurs afterward in the samequest. As the player is wandering the world, he happens upon the body of Casus Sandral, who was killed by wild animals in a dangerous area. Heading back to the Sandral estate with Casus' diary, the player finds the compound shut down entirely with no possibility of completing the quest.

These two inconsistencies occurred because the designers assumed the player would experience the world in a particular sequence and did not plan any deviations. There are even more examples within the game *Knights of the Old Republic*, and it isn't from poor work. The inability to track and test all the interactions is a problem as the size, interactivity, and scale of the stories increase.

1.1.4 Game Development and Tooling

Game development is a large process often consisting of many teams with disparate skill sets. The last goal is to create a model and tools that can be used by everyone and not just someone with the knowledge of programming, computer science or engineering. This means creating a plot model that can succinctly model all possible storylines and be understandable by all but also translatable into another form for verification. Next, there needs to be a tool that can fit into the typical software development lifecycle used by teams.

1.2 Research Questions

Our research questions are as follows:

- RQ1: Can we model storylines generically used in industry and academics?
  - RQ1.1: What are the defining characteristics of each storyline type?
  - RQ1.2: What characteristics and concepts can be used in a domain specific language or model?
- RQ2: Can an automated system be made that prevents inconsistent storylines?
  - RQ2.1: What approach makes the most sense?
  - RQ2.2: How do we transform a model for understanding storylines onto it?

- RQ2.3: Can the system fit into the software development lifecycle used in industry?

1.3 Contributions

The goals of this dissertation are to show that formal verification of video game plots can prevent inconsistencies in the plots, manage design complexity by integrating into the software development process, and can be done using a model game designers and developers can understand. Speaking with many writers, designers, quality assurance managers, testers, and developers in the industry about this problem, they all said it could not be solved and what they are doing is the best way to mitigate the problem. After learning of this proposed solution, they were very interested to learn more and see this solution come to fruition. To allow designers to create larger and more complex worlds, we propose a tool that applies formal verification to the domain of game worlds and storylines. The workflow is to translate stories into a graph then validate the states and transitions. This tool can be used at multiple levels: with the designers themselves and within the software development process handled by developers and quality assurance. Once complete, we believe stories can be dramatically more engaging, interactive, and grander in scale.

The proposal addresses the inconsistent state problems found in current and future games. We propose the following contributions:

Table 1.1: Research Questions to Contributions Matrix

	RQ1.1	RQ1.2	RQ2.1	RQ2.2	RQ2.3
RC1	X				
RC2			X	X	
RC3		X			
RC4			X	X	X

- RC1: A storyline categorization framework to categorize storyline type
- RC2: Map interactive game storylines and worlds to formal verification
- RC3: Define an abstract game plot model that can be used and understood by the layperson for storyline verification
- RC4: Create a working embodiment of the system

1.4 Outline

The rest of this dissertation is outlined as follows:

- **Chapter 2**: provides background information on storyline models and implementations, formal verification techniques, and game development
- **Chapter 3**: provides background information on modeling techniques
- **Chapter 4**: introduces our storyline characterization framework, storyline description language, and case studies on implementing our storyline description language in a formal verification program
- **Chapter 5**: discusses the implementation of StoCk

- **Chapter 6**: provides three case studies that are typical game development tasks
- **Chapter 7**: summarizes the dissertation and discusses potential future work
- **Appendix A**: the exploratory survey concerning games with storylines
- **Appendix B**: the history of storytelling
- **Appendix C**: provides an ontology of video games

Chapter 2

Background and Related Work

This dissertation addresses the challenges of preventing inconsistent storylines in narrative-driven games and preventing inconsistent storylines. In this chapter we highlight the main disparate subjects this thesis is built on: storyline models, formal verification, the typical software development lifecycle, and how storylines are implemented in practice. Chapter 3 will discuss the most relevant modeling techniques.

2.1 Storyline Models

Given the multitude of ways in which to implement storylines, they can be placed into roughly a few storyline models. Much has changed in video games since the first flashing screen of *Pong*[1] appeared over 40 years ago; the interactive nature of games allows players varied stories that can be as shallow or as deep as they would like. The storyline models presented are refinements and additions based on storyline models discussed in previous works [13], [14], [15], [16], [4], and storytelling (see Appendix B). The table below, Table 2.1, shows some of the most popular games in recent history with their game style

[1] Atari Inc., *Pong*, Arcade, 1972.

Table 2.1: Recent games and their story line models

Game	Release Year	Style	Model
Final Fantasy XIII	2010	RPG	Linear
Half-Life 2	2004	FPS	Linear
DooM	1993	FPS	Linear
Super Mario Bros.	1985	Platformer	Linear
Zork	1980	Puzzle	Branching and Foldback
Fallout 3	2008	Action RPG	Threaded
Grand Theft Auto V	2013	Action-adventure	Dynamic Hierarchical
Sim City	1989	Simulation	Emergent
Gran Turismo 5	2010	Racing	Emergent

(see Appendix C for descriptions of game styles) and storyline models.

2.1.1 Linear Storyline

Linear storyline representation is the simplest to understand. It is like reading a book: there is a single plot line and the player plays through the story from beginning to end. An abstracted linear plot figure can be seen in Figure 2.1; each node can be thought of as a plot point and the edges as the players path to the next plot point. The story never branches and is straightforward. There can be goals and sub-goals within the game, such as "press all switches to open a door," however, the player cannot continue the story without completing the task given to them.

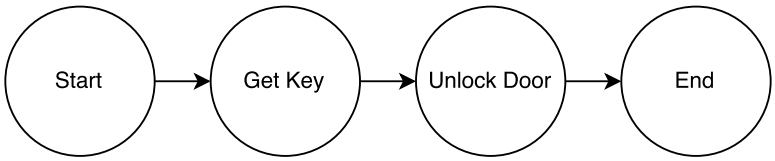

Figure 2.1: Linear Plot Example

Linear storyline games are some of the most popular even today. *Final Fantasy*[2] and *Lost Odyssey*[3] are examples of games with a linear story line in the role-playing games (RPGs) genre where the player controls multiple characters within a group that travel together along a linear story. *Half-life*[4], *Doom*[5], and *Quake*[6] are popular first person shooter action games that have linear stories. Finally, some of the oldest style of games are the side-scrolling platformers with linear storylines such as: *R-Type*[7], *Streets of Rage*[8], and *Super Mario Brothers*[9].

2.1.2 Branching and Foldback Storyline

Branching schemes are the next logical extension to the linear storyline. These are analogous to the Choose Your Own Adventure style of books. At points in the story, the players are allowed to make a choice that changes the

[2]Square Enix, *Final Fantasy*, Console, 1987.
[3]Mistwalker, *Lost Odyssey*, Console, 2008.
[4]Valve Software, *Half-Life*, CD-ROM, 1998.
[5]iD Software, *DooM*, Floppy, 1993.
[6]iD Software, *Quake*, CD-ROM, 1996.
[7]IREM Software, *R-Type*, Arcade, 1987.
[8]SEGA, *Streets of Rage*, Console, 1991.
[9]Nintendo Entertainment, *Super Mario Brothers*, Console, 1985.

story. In a branching story, the tree becomes larger and larger as the choices expand [13]. A foldback scheme works to contain the expansion of the tree by allowing various paths to join at a later point in the game. An example of branching and foldback representation can be seen in Figure 2.2. In these story graphs, all transitions from one plot event to another are explicit. There are goals and sub-goals that can contain multiple choices and, depending upon the users choice, the next plot point can change. The foldback of the plot would be when the branches merge at a common plot point in the future. The best known example of branching storyline games are the *Zork* series[10] where the player controls a nameless adventurer looking for treasure in a labyrinth.

2.1.3 Threaded Storyline

The threaded storyline involves multiple paths that develop on their own regardless of what else is happening within the game. As the game is nearing the end of the story, threads converge to create the final events. Often, there is a single thread that represents the main story and additional threads that constitute additional quests the player can choose to complete. These additional threads are often called side-quests since they do not affect the main story and are not necessary for the completion of the game; however, completing them can change the game in not-so-subtle ways. Figure 2.3 shows a possible threaded storyline. The main story is the center linear path with side-quests that can occur after the second and third plot points.

[10]Infocom, *Zork*, Disk, 1980.

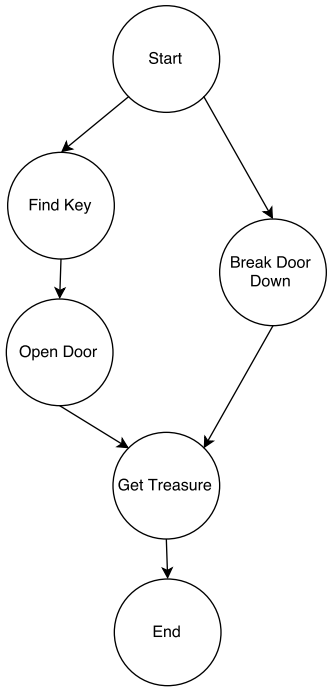

Figure 2.2: Branching and Foldback Storyline Example

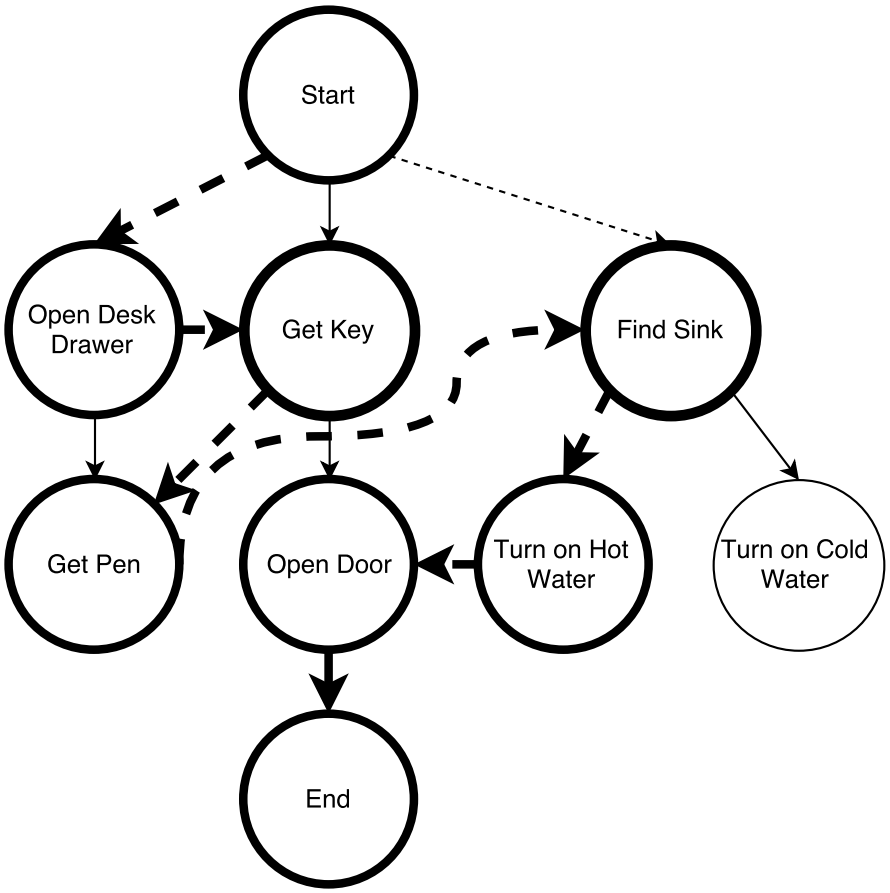

Figure 2.3: A Threaded Storyline Example

There are many threaded storyline style games that are very popular and some recent examples include the following: *Fallout*[11], *The Elder Scrolls*[12], and *Discworld Noir*[13]. In each of the games, the player is given a simple backstory and a starting point. From this starting point missions and quests appear for the player to complete. The player can complete the missions and quests and continue the storyline, or they are free at any point to stop completing the quests and interact with the world around them until they wish to continue the story.

2.1.4 Dynamic Hierarchical Storylines

Dynamic hierarchical storylines are an extension of threaded storylines in that there is an abstraction to group certain story elements together to help manage some of the story complexity. Each level of the hierarchy is usually small and manageable, but still make up a larger complex storyline when assembled. The nesting of sections could be very deep, but in practice it is normally only two levels deep. Figure 2.4 shows a possible two level dynamic hierarchical game. In the academic space, SBDM and Facade would be considered a dynamic hierarchical storyline.

Assassin's Creed[14] and *Grand Theft Auto*[15] are two popular games that use a dynamic hierarchical storyline. As certain quests are completed new

[11]Interplay, *Fallout*, CD-ROM, 1997.
[12]Bethesda Softworks, *The Elder Scrolls*, Disk, 1994.
[13]Perfect Entertainment, *Discworld Noir*, CD-ROM, 1999.
[14]Ubisoft, *Assassin's Creed*, Console, 2007.
[15]Rockstar Games, *Grand Theft Auto*, Disk, 1997.

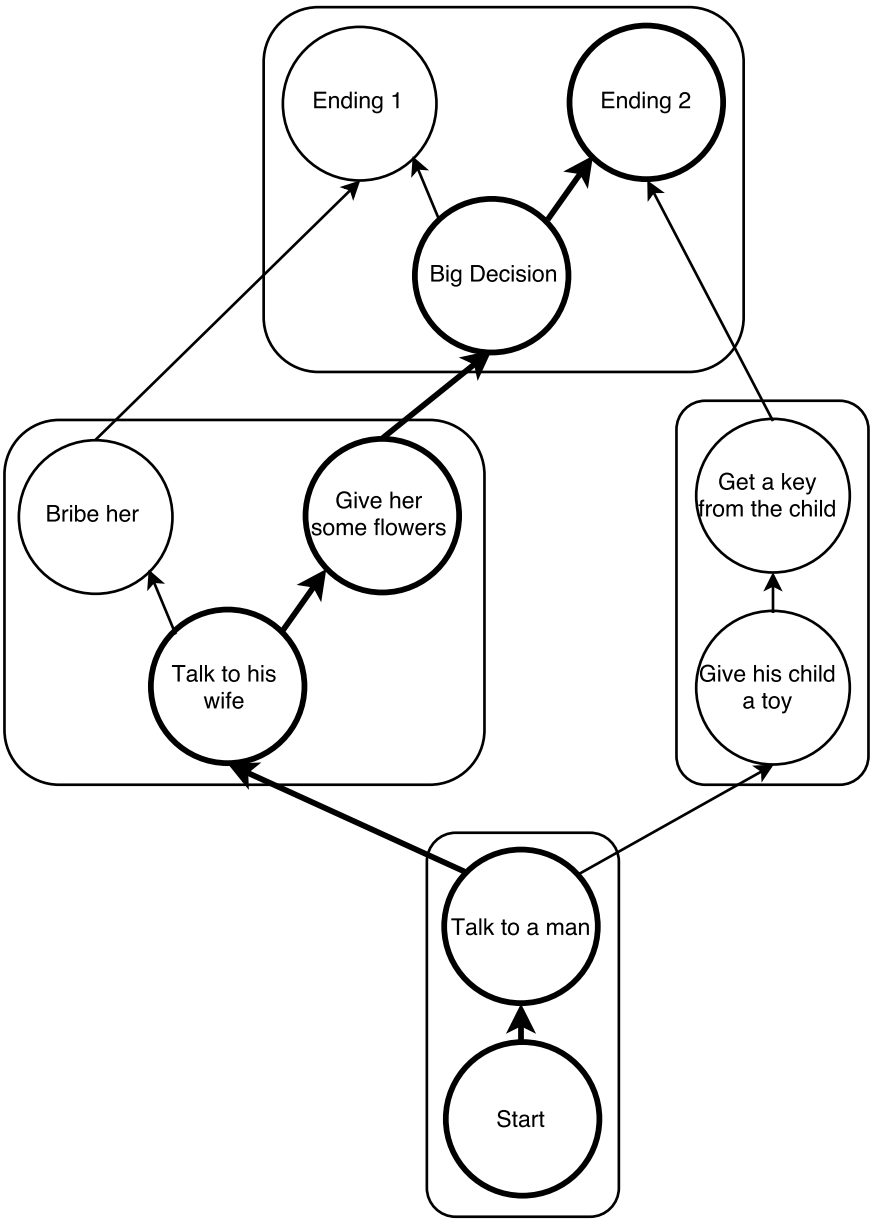

Figure 2.4: A Dynamic Hierarchical Example

areas of the world are opened up and new quests are added to the game for the user to complete. *Dragon Age: Origins*[16] also uses this style but in a different manner: at given points in the game, the user is given the choice as to which city they travel to, and when they arrive at the city, new quests are provided to the player.

2.1.5 Emergent Storyline

In emergent storyline games, there is little or no back-story and often no predefined quests or goals. The player is interacting within a simulation; the best example of this style of game is *Sim City*[17]. The starting state of the game is a blank world and the rules of the simulation. The rules or goals of the game can change as the users meet certain requirements. Other games with emergent storylines are racing games such as *Gran Turismo*[18]; puzzle games like *Tetris*[19], *Columns*[20], or *Candy Crush Saga*[21]; and arcade games such as *Asteroids*[22], *Pong*[23], and *Q*bert*[24]. In the academic research space, OPIATE and PaSSAGE are emergent storylines since their storylines are generated as a response to the how the user interacts with the world and characters.

[16]BioWare, *Dragon Age: Origins*, CD-ROM, 2009.
[17]Maxis, *Simcity*, Disk, 1989.
[18]Polyphony Digital, *Gran Turismo*, Console, 1997.
[19]Nintendo, *Tetris*, Console, 1984.
[20]SEGA, *Columns*, Console, 1990.
[21]King, *Candy Crush Saga*, Internet, 2012.
[22]Atari, *Asteroids*, Arcade, 1979.
[23]Atari Inc., *Pong*, Arcade, 1972.
[24]Gottlieb, *Q*bert*, Arcade, 1982.

2.2 Formal Verification

Formal verification is the process of determining the correctness of programs or systems using mathematical constructs. The most common methodologies for formal verification that are possible in our use case are: theorem provers, SAT solvers, and model checking.

2.2.1 Theorem Provers

Theorem provers are programs that attempt to prove mathematical theorems by derivating sound conclusions from a set of facts. These systems can be human-directed or automated [39]; however, in the confines of the proposed research, only automated techniques that operate on first-order logic and are not covered by SAT solvers or model checking, which are covered in later sections will be described.

First-Order Resolution

First-order resolution with unification is one of the oldest techniques of theorem proving proposed by Robinson in 1965 [40]. The resolution principle works on sets of first-order logic equations by combining the steps of performing substitutions of terms for variables and applying truth-functional analysis to the resulting equations into a single step. Continually applying the resolution rule to a set of propositional statements and a statement to verify creates and resolves new statements until a determination can be made about the statement under test. This technique can be automated using a search

algorithm following a few simple steps. First, all axioms and the negation of the statement to be proved are conjunctively connected. The resulting statement is transformed into conjunctive normal form where a set of clauses is formed. The resolution rule is then continually applied until an empty clause is derived or there are no new clauses. If there is an empty clause, the original statement to be proven is true; or if no empty clause can be derived and there are no new clauses to which to apply the resolution rule then the statement to be proven is invalid. First order resolution has a few issues that can occur such as Godels incompleteness theorems, the halting problem, an explosion of generated clauses, and the possibility of looping in an infinite branch without finding a contradiction[41] [42].

Model Elimination

Model elimination [43] [44] is another technique for theorem proving. It was designed to prevent the explosion of created clauses that can occur through resolution by attempting to prune clauses that will not result in verification of the statement under test. Model elimination attempts to lessen the number of candidate statements created by searching through partially false statements already contained within the clauses. From these partially false statements, it further searches these for a completely false statement by refining the partially false statements. However, like the resolution procedure it faces the same problems, although the explosions of generated clauses does not expand as rapidly. Model elimination is the basis for the Selective Linear Definite clause

resolution [45] procedure found within Prolog.

Lean Theorem Proving

Another technique to solve theorems, lean theorem proving, uses the specifications and abilities of the implementing language to create theorem provers in a minimal amount of code [46]. These are often implemented using Prolog due to its grounding in first-order logic and backtracking engine and can be implemented in a few lines of code. The issues for lean theorem provers change based upon the language in which it is implemented.

Analytic Tableaux

Analytic tableaux [47] [48] methods continually apply a set of rules to a formula and the resultant sub-formulas creating a tree-like structure that determines the satisfiability of a given first-order logic formula. A branch on the tableaux is considered closed if a path contains a literal and its negation. If all branches are closed then the formula is not satisfiable; otherwise it can be satisfied. An issue that can arise with analytic tableaux methods is that certain tableaus cannot be closed even when they are non-satisfiable when handling first-order logic.

Superposition Calculus

Superposition calculus is considered state-of-the-art for many theorem provers [49] [50] [51] [52] and like the other methods described, it is a set of

operators operating on first-order logic. However, it utilizes Knuth-Bendix completion in addition to first-order resolution [53] [54] [55]. The operators used are: deduction, deletion, and simplification. Deduction adds clauses that logically follow the given clauses, deletion removes clauses that are composites of other clauses, and simplification allows for a subset of clauses to be created from multiple sets of clauses. When applied programmatically it is refutation-complete, meaning that given unlimited resources and a fair derivation strategy every unsatisfiable clause set can eventually be proven unsatisfiable.

2.2.2 SAT Solvers

SAT Solvers, or satisfiability solvers, are programs that determine if the variables of a boolean formula can be assigned in a way to evaluate to true. There are three main approaches for modern SAT solvers: brute-force, look-ahead, and conflict-driven. Brute-force and look-ahead solvers use a breadth-first search algorithm, normally based on the DPLL algorithm, as their base; whereas, conflict-driven SAT solvers use a depth-first search.

Brute Force

Brute force is the most naïve solution, and grows exponentially as the SAT equation increases in size. Brute force creates the entire truth table for the equation and attempts to find the first solution that is true. This attempt at solving SAT problems is unrealistic because of the computational time and is easily surpassed by the look-ahead and conflict-driven methods discussed

next.

Look-Ahead

DPLL, an acronym for Davis-Putnam-Logemann-Loveland [56] and the basis for two methods of SAT solving, is a backtracking-based search algorithm for determining the satisfiability of propositional logic in conjunctive normal form as an extension to an algorithm by Davis and Putnam [57]. The algorithm runs by choosing a literal, assigning a truth value to it, simplifying the formula, and recursively checking if the simplified formula is satisfiable. If not, the same recursive check is done with the opposite truth value. There are additional rules that enhance the normal backtracking algorithm: unit propagation and pure literal elimination. Unit propagation occurs when there is a single unassigned literal and there is only one choice to make the clause true. Pure literal elimination occurs when a propositional variable only has one polarity to make all clauses containing them true. Brute-force algorithms complete the tree naïvely looking for a solution, whereas look-ahead algorithms use heuristics to drive the search down branches that are more probable to have solutions [58].

Conflict-Driven

The third method, conflict-driven SAT solving, which is currently the most successful SAT solving architecture, takes a different approach. It performs a random depth-first search and when a conflict occurs a heuristic is

invoked and a backjump is performed to an assignment further up the depth-first search where the depth-first search restarts. Some of the most popular conflict-driven SAT solvers are: *zChaff* [59], *Minisat* [60], and *Rsat* [61].

2.2.3 Model Checking

Model checking, a field of research pioneered by [62] [63] and [64], is the act of proving correctness of a system or algorithm with respect to a set of properties using formal mathematical methods. Some areas in which formal verification is used are protocol, software, and hardware validation and verification [6]. Software applications and systems, however, present a problem to current model checking techniques. The large and often times unconstrained, state-space make verifying fully a piece of software within a development time-line impossible [65].

Compared to theorem provers, model checking requires no human reasoning; once a model is to be created, it creates a decidable problem [66]. Current research in model checking has focused on the state-space explosion problem and there has been a large improvement in the past years using new techniques such as bitstate hashing, BDD, on-the-fly, compositional, and partial order techniques. These developments reduce the memory requirements and allow the model checking software to move from the realm of toy to real-life tool.

Bitstate Hashing

Bitstate hashing is a technique used to increase the quality of verification by reachability analyses that normally fail due to their size. It can perform high coverage verification within a memory space that may be orders of magnitude smaller than what is needed for an exhaustive verification. Each state is represented by a number and is then passed to a hash function whose result is then passed to a bit field to store if the state has been checked. There are trade-offs that must be made due to hash-collisions in the hashing function, the size of the memory in which to operate, and the number of states that must be checked [67].

Binary Decision Diagrams

Another method of reducing memory usage is to represent Boolean functions efficiently using ordered binary decision diagrams (BDDs). Boolean functions are turned into binary trees rooted at a given variable and having other nodes as their leaves. At the bottom of the tree are terminal nodes that specify the output of the function. Then two methods are applied to the binary trees: first, all terminal nodes are merged and second, all isomorphic subgraphs are merged. The reduced binary decision diagrams are functionally equivalent to the original function, but can often be many times smaller than the original representation [68] [69].

On-the-Fly

The next technique used in reachability analysis is on-the-fly algorithms. This style of algorithm only expand the state-space as needed and reduce the amount of randomly accessed memory used and instead use sequentially accessed memory such as stacks. The largest problem with on-the-fly algorithms is that they do not always check the entire state-space, meaning that not every state is checked [70].

Compositional

Compositional algorithms view the problem as a composition of smaller finite-state machines such that only FSMs that are needed are kept in memory at one time. As these smaller combinations are verified their results can be used within the larger machine. It decouples independent states and collapses states that behave similarly. Additionally, in some cases the verification questions can be answered without involving all of the machines [71] [72].

Partial Order Reduction

Partial order reduction exploits the fact that concurrently executed transitions can result in the same state when executed in different orders. The algorithms determine a representative subset of transitions for the concurrently executed transitions to prevent the entire state-space having to be explored [73] [74] [75].

2.3 How Storylines are Implemented in Current Games and Interactive Dramas

Video games and interactive dramas implement storylines models using various techniques. These are driven by constraints of time, scope, and goals of the game. Examined below is a cross section of interactive storytelling and real-time storyline generation. Interactive storytelling attempts to create meaningful stories based upon user inputs for dramatic purposes. Drama is achieved by maximizing metrics either as an overall value or as a local value. The initial design of an interactive storytelling system was by Laurel [76] and the first attempt at such a system was the search-based drama management work by Weyhrauch. Real-time storylines are those created by OPIATE and PaSSAGE which use an underlying theory such as Propp's Morphology for fairy tales or the hero's journey to create a never-ending storyline based on the methodology.

2.3.1 Search-based Drama Management and Facade

Search-based drama management (SBDM) was the first drama manager used to control a story and is based on two fundamental assumptions: an evaluation function can encode an author's aesthetic and a search mechanism can be used effectively in guiding a storyline [77]. Weyhrauch proved the two assumptions in his dissertation with the simple interactive game, Tea for Three. The drama manager (in this research, MOE) uses these two assumptions to place values upon plot points within the plot by abstracting the plot into

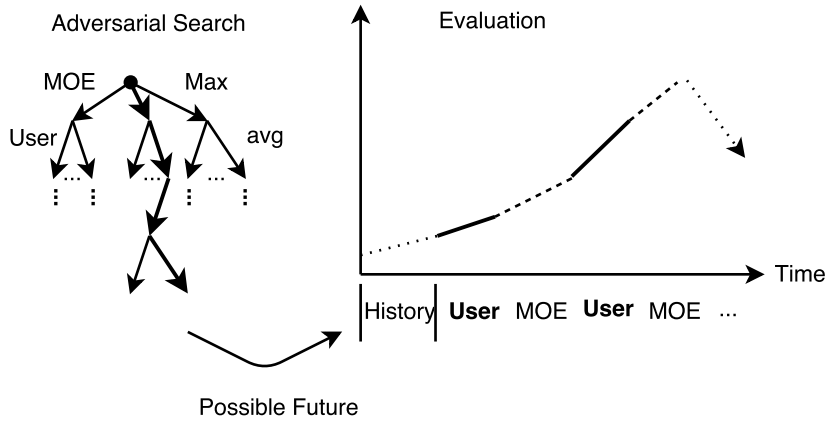

Figure 2.5: Example of Search Based Drama Management Algorithm based on [77]

a series of user and drama manager moves. The drama manager views the story as an adversarial game with the user and drama manager making moves toward an end game. The algorithm is a modified minimax algorithm used in many checkers, chess, and tic-tac-toe games to guide the story. The evaluation function uses multiple factors that each have their own ideal graph for the story. Figure 2.5 shows an example of each factor of the algorithm. SBDM has been examined and extended by various research groups [78], [79] and found to be lacking in usefulness. However, the overall method was refined in Facade.

Mateas' dissertation introduced Facade [80], the first fully implemented interactive drama system. It offers a complete, real-time dramatic experience with a highly interactive, character-driven story. Facade places the user in the role of a friend of Grace and Trip who have invited the player over to dinner. It becomes apparent that Grace and Trip's marriage is crumbling. The user can

interact with Grace and Trip through dialog and simple actions, although the emphasis is on dialog. Facade's story is broken down into beats, that are the smallest bit of drama that causes a change in dramatic tension and a change in a character. The beats tightly integrate the story and actions for the artificial intelligence (AI) controlled characters. Unlike SBDM, Facade views each beat as an individual object having a set of conditions that determine when it can be enacted, as opposed to a set of moves to be taken by the player and the AI director. A director agent coordinates the behavior of the AI characters with the beat enacted and makes the decisions on which beat to enact next by attempting to have the story follow an ideal dramatic tension graph using the beats available to it at a given time. Figure 2.6 shows how a completed story might look when the ideal line versus the actual line are graphed. Magerko [81] notes some of Facade's problems; the user can provide non-sequitur inputs and the story will still go toward a logical conclusion, ignoring the users input. Its scalability is poor because of the high entanglement between AI characters, plot associated with the characters, and the authorial burden to generate the numerous beats to create a story.

2.3.2 AI Controlled Emergent Storylines

Another interactive storytelling approach is introduced by Fairclough's research (OPIATE), which is different from both SBDM and Facade because it has no notion of overall story. It uses a database of templated encounters to create an emergent story for the user [82]. These encounters are based

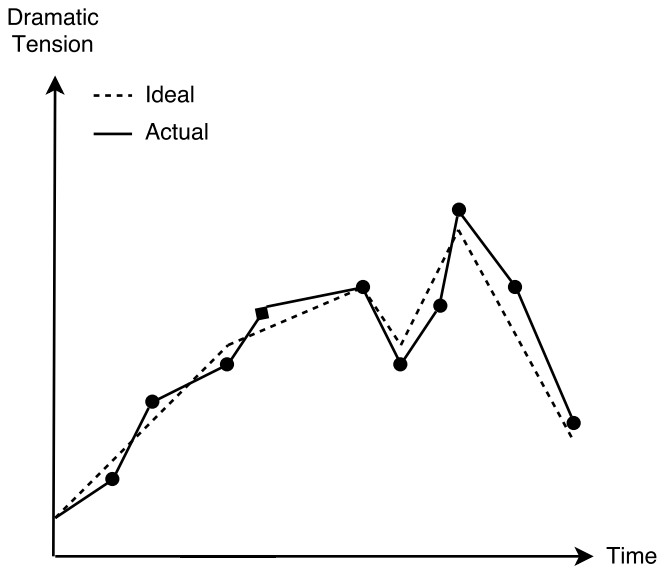

Figure 2.6: Example Facade Graph based on [80]

upon Propp's 31 functions and five character archetypes found in his analysis of Russian folk tales [83]. The virtual world contains non-player characters (NPCs), items and different settings. The NPCs have an array of likeness variables that describe how the NPCs perceive the user (sometimes called Player Character or PC) and the other characters. The NPCs also have a simple vocabulary that allows them to gossip with one another and affect each other's perceptions of the other characters. During play, a case-based reasoning system chooses an encounter based upon the state of the characters and items within the world. The reasoner compares the state of the world with its database to give the player a quest.

Bulitko, et al. introduces PaSSAGE [84], which, like OPIATE, has

an emergent story line. Users progress through the world interacting with characters and completing encounters in order to be given a new one. Unlike OPIATE, the next encounter given to the user is based upon a user model and the position in the hero's journey that the user has just completed, instead of the characters and items within the world as in OPIATE. The metrics within PaSSAGE's user model are then matched against the encounters stored within the encounter database and the encounter with the best match is selected. If the virtual world lacks NPCs that fit conditions for the encounter, the engine continues to scan the world until the conditions are met, and then the encounter is started.

2.3.3 Triggers and Actions

In the video game industry, game storylines are most often implemented by decomposing the storyline into a series of triggers with optional gating functions and actions that change behaviors of characters and global state as the storyline progresses. *Fallout 3* and *The Elder Scrolls V: Skyrim* both have threaded storylines. These storylines are decomposed into a series of triggers that activate when the user crosses a boundary, examines or picks up an object, or talks with another character within the world. These triggers then change the state of a global object that affects the game world in some way.

2.4 Software Development Process for Video Games

The creation of a video game can be split into three building blocks: story development and writing, art, and coding. All of these building blocks are held together by a software development process.

Story Development and Writing

The writers of the story team will define the world that the player will interact with in the final game. This includes creating and developing the storyline and dialog within the game. As mentioned earlier, the tools used by writers are often simplistic such as Microsoft *Word* and *Excel* and text documents shared on a company wiki page. The writers will also use the game development toolkits provided by the developers (a.k.a. coders) e.g., GECK or a scripting language that defines the dialog (pers. comm. Royal McGraw, Pixelberry Studios).

Art

The artwork is needed throughout the game development process. Without the artwork, there would nothing interesting to display on the screen. Artists use a variety of art programs depending on the style of game being created. However, drawing and animation tools such as *Maya*[25], *zBrush*[26],

[25] `http://www.autodesk.com/products/maya/overview`
[26] `https://pixologic.com/`

and *Photoshop*[27] are standard tools seen in the industry[28]. These tools are used to create and edit the artwork as well as handle modeling, animation, and applying texture to the models. The artwork is thus placed into the game in order to be tested and verified during the development process.

Coding

The creation of video games often have three levels of code: scripting, gameplay, and engine code. Scripting and gameplay code define the high-level behavior of the game and the game engine provides the building blocks for them to work.

Scripting The scripting code is often the most accessible and used by the majority of developers and writers involved to create what most people consider the game. This often includes anything related to the storyline such as: narration, conversation, and quests. Narrative-driven games, such as point-and-click adventure games and many mobile games are written using scripting languages such as Lua[29], JavaScript[30], or a custom language like GECK Script[31] used in *Fallout 3* that abstract the low-level hardware interactions such as rendering images on the screen and higher level functions such as object clipping and physics from becoming a concern.

---

[27] `http://www.adobe.com/products/photoshop.html`

[28] `http://travestychandler.kinja.com/video-game-artists-what-do-i-use-821327309`

[29] `http://www.lua.org/`

[30] `http://www.ecmascript.org/`

[31] `http://geck.bethsoft.com/index.php?title=Scripting_for_Beginners`

Gameplay A level below the scripting code is the gameplay code that often handles gameplay mechanics such as controlling non-player characters (NPCs), populating the game world, and handling collision detection. Unlike the scripting code, gameplay code could be written in a programming language such as C. As an example, *Unreal*[32] allows the use of C, C++, and their own scripting language called Blueprints.

Game Engine The game engine handles input and output, physics, and the low level processing for the game to run on the given device. Game engines often come with most of the related tools needed to create a game and provide a game creation pipeline to handle managing art assets, code, and deployment. *Unity3D*[33] and *Unreal* come bundled with visual tools that can act as an integrated development environment (IDE) for new game development. *Playmaker* is used by *Unity3D* to provide a visual editor for AI behavior, animation, interactive objects, cut scenes, prototypes, and interactive walkthroughs. It provides an environment that novices can quickly grasp but also provides the ability to dig deeper into the scripting language as necessary for more advanced users. The *Unreal* engine provides a user interface for users to create games with the system along with the ability to create games for a variety of platforms all using one workflow. *Gamemaker: Studio*[34] and *RPGMaker*[35] are tools that provide easier interfaces and the ability to create

[32] `https://www.unrealengine.com/what-is-unreal-engine-4`
[33] `https://unity3d.com/`
[34] `http://www.yoyogames.com/gamemaker`
[35] `http://www.rpgmakerweb.com/`

cross-platform games quickly, although they are focused on games that are simpler than those that can be created by *Unity3D* or *Unreal*.

2.4.1 Software Development Life Cycle

In the majority of software development firms they have a date at which the software must be released to stakeholders and customers, and because of this, a release date is given. This release date forces a timeline for a feature complete program that is in direct conflict with a strict agile methodology that believes in only solving immediate problems and not doing any long term planning. As such, milestones for development are put into place. These milestones provide incremental goals in a waterfall-style process. Between these milestones, the development teams use an agile process that uses scrum with kanban boards and a backlog. Once the product is released, it turns into an agile process using bugs and additional features as the backlog for the agile process.

As an example, we'll use a hypothetical game that is scheduled to be released a year from now. The initial planning meetings for the game would involve the heads of the respective departments: writing, art, development, test, development operations (dev ops), and stakeholders. This meeting would have high-level goals set for the project. In this case, it would be goals that must be met by the end of each quarter. The groups would then set deliverables for each quarter as in Table 2.2.

Table 2.2: Development Plan Goals Matrix

	Q1	Q2	Q3	Q4
Writing	Create story outline, characters , and world	Flesh out story, work with dev. to implement story	fine tune story, add lore	fix grammar, spelling, and bugs found by test
Art	Create placeholder assets and design characters, worlds, and weapons	implement assets	finish assets	touch up any assets
Development	create base code (menu, world, fighting)	refine code and work with DevOps and test for pipelines	complete engine code, begin testing	debug and performance
Test	create testing frameworks and determine team involvement	setup testing pipelines, begin testing for game	play testing and installers	test critical path
DevOps	determine hardware needs, deployment processes	large scale testing on platforms	provide support for test and development, prepare distribution	prepare for release

First Quarter

Initially, test, development, and dev ops must determine how they will collaborate with one another. In some cases, it might be better to collaborate with one another; in other cases, working alone then integrating before a checkpoint date might be a better choice. In our hypothetical case, development operations sprinting separately from test and development is the correct decision since the dev ops group must plan on purchasing hardware and determining how to deploy the game to the various test instances. Development and test sprint together because test will be working on testing frameworks that will be tied to the interfaces developed by the developers and also planning for the game play test in the fourth quarter. As the first quarter is coming to a close, the teams meet again and begin to adjust the plans for the second quarter milestones based on their work in the first quarter. In this example, we'll assume that the first quarter goals are all met.

Second Quarter

Testing efforts in this quarter will need to start merging and multiple teams must coordinate with one another to get pipelines in place for this testing effort. Development, testing, art, and dev-ops must all work together to accomplish multiple tasks while still striving toward finishing the game. At this point, the development team will be spread out interacting with these teams in various ways. It means that team members coordinating with other teams will attend multiple status meetings and work toward a common goal.

Additionally, since these shared goals often run into road blocks, the backlog of work will have to be decomposed in fine enough granularity that the developers can take work off the backlog and complete it while waiting to become unblocked. In this quarter, once multiple pipelines are set up, each department can begin to see results of their work. The pipelines allow departments to test independently of one another and then push completed work to the other pipelines once they are released at the end of a sprint.

Third Quarter

Now, some of the teams will work closely together while others will begin preparing for the launch of the game. Development and test will continue working closely together and art will be working with the development team to ferret out graphical glitches in the engine based on the artwork. The dev ops team will keep the pipelines in working order while preparing auxiliary systems for the expected load to be placed on their servers during the launch of the game. Test will spend time testing the game of various hardware combinations and work with development to profile and enhance the speed of the game. In the third quarter many of the tasks are winding down in creating the game as testing and debugging tasks begin to spin up for the large push that occurs before the game goes gold, or releases, in the fourth quarter.

Fourth Quarter

Everyone's effort is placed into finding bugs and quickly fixing them. The writers comb over all dialog and make sure everything has been recorded. The art department verifies all art assets are complete and the final graphical tweaks have been made. Development wraps up the game engine work and starts focusing solely on bug reports sent to them by the test department. The test department adds a large amount of temporary labor to help them playtest the game. The playtesters play through the game beginning to end attempting to finish the game. Any place the playtesters find an error, bugs are submitted to the correct teams and triaged immediately by the scrum masters. The dev-op teams implements the strategy for how the game is to be delivered to the users: Steam, disc, ordered and downloaded from the company's store, or downloaded using a purchase key. These strategies all need enough hardware to scale horizontally during peak times. The groundwork done in the third quarter to determine the extra load is tested early in this quarter with 4x load tests against the hardware and software platform to make sure the distribution platform performs correctly. As the fourth quarter comes to a close, all teams are working furiously, often long into the night to make the release candidate deadline. Once the release candidate has been accepted the game is considered completed and sent off for manufacturing and to the services selling the game.

Chapter 3

Modeling Techniques

Modeling is a very important aspect of computer science since the ability to abstractly represent an idea or process using a model allows for deeper understanding and analysis. When deciding how to model problems encountered in this dissertation a few common models kept arising: finite state machines, petri-nets, condensed graphs, UML state machines, and SAGA.

3.1 Finite-State Machines

Finite-state machines (FSMs), or finite-state automata, are a model of computation used in computer science and engineering. They are consistent, easily debugged, allow better understanding of processes, and represent a way of thinking about computation. There are two types of FSMs: deterministic and non-deterministic. Deterministic means there is one and only one transition given an input to a next state where with non-deterministic, there are multiple possible next states.

FSMs at the formal level are defined as a 5-tuple consisting of a set of states, an alphabet, transition functions, a starting state, and a set of accepting states [41]. Graphically, a FSM is a graph containing arcs, labeled arcs, and

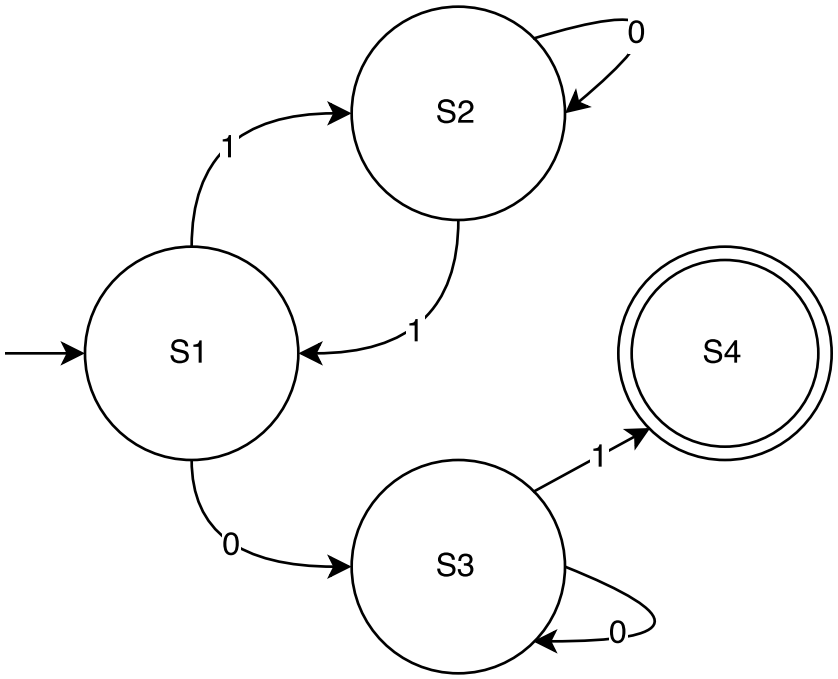

Figure 3.1: Example Finite State Machine

circles. An arc is a transition, a labeled arc is a transition with its definition for transition defined, and circles are the various states. A state made of an outer circle and an inner circle is the accepting state.

Examining the values in the 5-tuple more, the set of states describes the system being modeled at some point in time where inputs and transitions have occurred transitioning the system into the particular state. The alphabet, or input alphabet, is the set of inputs allowed for the system. Next is the start state, which specifies the state the system begins in. The accept states specify when the machine no longer accepts input and is done processing. Finally, the

transition functions are a set of functions with each function associated with a state and inputs accepted at the state that describe the next state for the system.

Now that a FSM has been defined formally, computation of a FSM must be defined. Assume we have a FSM 5-tuple, M, and a sequence of characters in the alphabet, referred to as w. M accepts w if there is exists a sequence of states in the FSM with the following conditions:

1. The machine starts in the start state
2. The machine goes from state to state according to the transition function
3. The machine accepts the input if the last state is in the set of accept states

A non-deterministic finite-state machine (NFA) is a super set of deterministic FSMs. The difference is that in any state a NFA could have a plurality of inputs that are acceptable for moving to different states through a transition function. When a NFA performs computation, at any point where there is a non-deterministic transition, the state machine copies itself and proceeds with computation for each copy. When a copy can no longer execute, it destroys itself and its computation. Formally, a NFA is defined the same as a deterministic finite-state machine with the exception of the transition function being a power set of the possible sets due to the non-determinism. In terms of formally defining computation, an NFA, N, with a sequence of characters in

the alphabet, w, N accepts w if there exists a sequence of states in the FSM with the following conditions:

1. The machine starts in the start state
2. The next state is in the set of allowable states for the current state
3. The machine accepts the input if the last state is an accept state

FSMs, and specifically, NFAs, appear to be a good choice to model game plots because they are the basis of computation for many concepts. FSMs can work for generating a model for the plot; however, it would be very verbose and hard for a layperson to create and understand. Take, for instance, a very simple scenario where a player may or may not find a key, and at some point in the future, having the key may open up an additional path that the player can take. In this case, there must be a doubling of states from the point where the player has the key, or does not have the key. As more possible paths are made based upon items or choices taken, the state machine must create copies of these nodes. Doing this a few times does not seem that bad, but one must consider, the average game contains over two megabytes of state and asking someone not skilled in the art of creating FSMs to create a FSM that can cover all possible cases becomes a very poor choice. The FSMs grow exponentially in size for what is a very simple case that appears in many video games: finding a key or object that affects the outcome of the game at a later time.

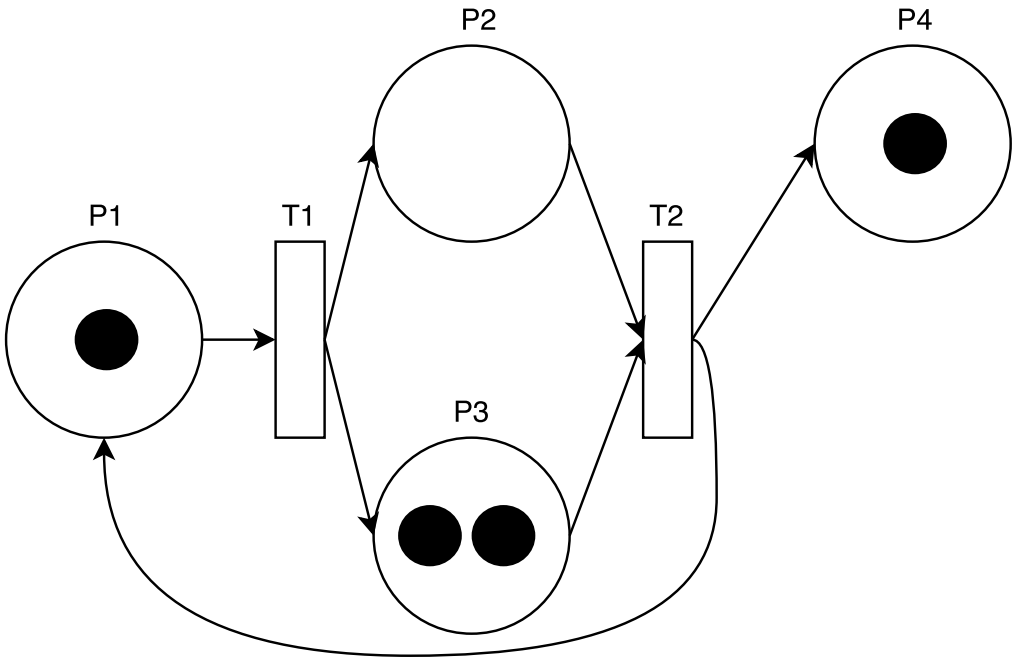

Figure 3.2: Example Petri-net

3.2 Petri-nets

Petri-nets, described by Carl Petri in his dissertation [85], are a mathematical modeling language for the description of distributed systems. Petri nets have been used to model parallel and distributed computing, workflow management, and network theory such as coordination models and theories of interaction.

Petri-nets are a directed graph containing two types of nodes: bars, representing transitions; and circles, representing places or conditions. The directed arcs associate pre- and post-conditions with the transitions. In addi-

tion to the bars, circles, and arcs, there are also tokens. Tokens are stored in the places and when sufficient tokens are contained in a pre-condition place, the associated transitions can fire, consuming the tokens and emitting a new token into the post-condition place.

In order for Petri-nets to become useful, a rule of how they are manipulated must be defined. That rule is the transition, or firing rule. This rule has three parts:

1. A transition, t, is said to be enabled if each input place, p, of t is marked with at least as many tokens as the weight of the arc between p and t
2. An enabled transition may or may not fire
3. A firing of an enabled transition t and removes the tokens from each input place of t ands them to the output place of t

A transition without any input place is called a source transition and one without any output is called a sink transition.

Petri-nets have been used to model various other systems. The most common are: finite-state machines, parallel activities, dataflow computation, communication protocols, synchronization control, producer-consumer systems, formal languages, and multi-processor systems. The wide range of modeling choices means they also have many properties that can be used for analysis of these systems, the most important in the case of formal verification are: reachability, boundedness, liveness, persistence, and fairness [86].

Comparing Petri-nets to FSMs appears to be an unfair comparison; however, even though Petri-nets can model FSMs, it does not make sense to do that. Petri-nets are more concise at modeling coordination between asynchronous systems whereas FSMs can model discrete behavior of a system over time much more concisely than a Petri-net. Petri-nets are very good at modeling interactions between systems, but aren't very good at modeling interactions within a system or modeling interactions where there are gating factors such as selecting a key or not. Describing this example in Petri-nets would cause both branches to be executed concurrently and could still not account for issues succinctly, such as where completing one quest before another could cause another quest not to complete if an invariant was invalid due to how Petri-nets execute. Changing the entire execution model then grafting additional features to handle specifics of how game plots can operate would, like in the case of FSMs, be problems for someone not skilled in the art to understand a heavily modified version of Petri-nets. Petri-nets, like FSMs, can have exponential blow up; although, it will occur when trying to describe serial and parallel quests.

3.3 Condensed Graphs

Condensed graphs provide a way to express complex dependencies in a program task graph or workflow [87] [88]. These directed-acyclic graphs consist of nodes and edges, where nodes are tasks and edges are sequencing constraints. Through some simple transformations, various execution models

such as availability, coercion, and imperative can be represented.

Condensed graphs have the notion of a computation triple that contain three prerequisite requirements for the evaluation of a function: a set of inputs, a function description, and a destination. Associating these graphs creates new condensed graphs and once the graph is executed, it represents the result of the computation triple.

Node themselves can contain other graphs; these nodes are referred to as condensed nodes. There is no limit to the nesting of the graphs in the condensed node and are helpful in abstracting computational methods being described by a condensed graph. In condensed graph terminology, these are H-graphs since they hierarchically describe an algorithmic process at various levels of abstraction.

The operations available to coerce and modify condensed graphs into the three computing models are: stemming and grafting, node deconstruction, and mutual reduction. Stemming is the process of breaking the connection between the output of a node and the input into another to change a static association into a dynamic one. Grafting, of course, is the opposite of stemming. Node deconstruction is the method of combining exact sub-graphs between graphs such that they can be shared. Finally, mutual reduction is a process to simplify the execution of condensed graphs by providing rules to combine disparate execution models.

In Figure 3.3, it shows a condensed graph, F, that represents the func-

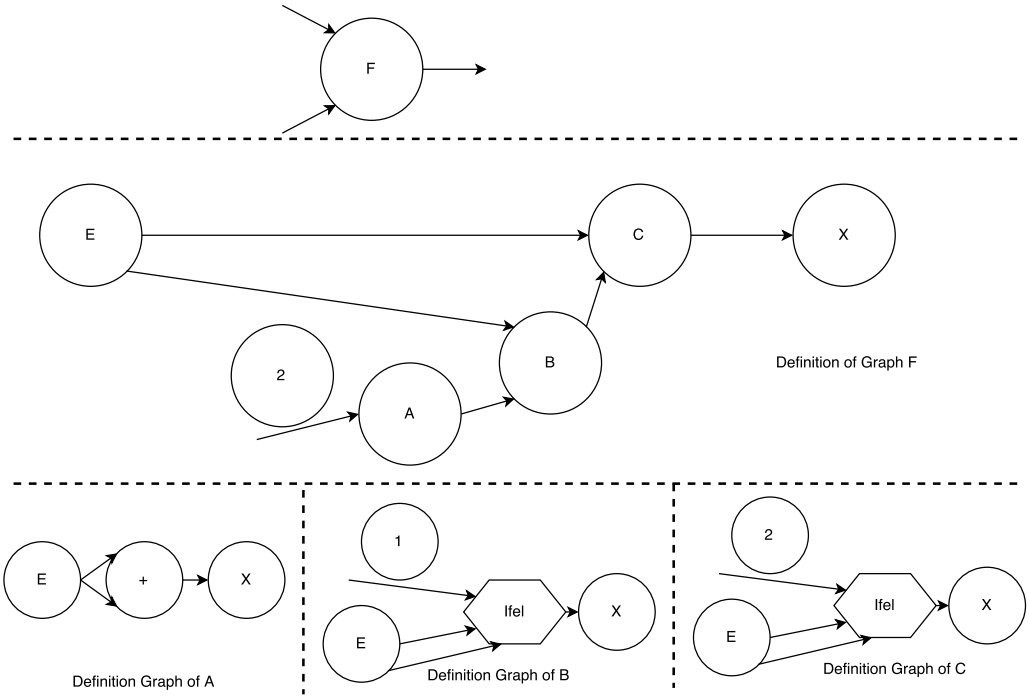

Figure 3.3: Example Condensed Graph

tions below, where F accepts two Boolean parameters and returns an integer.

$$F(x, y) = C(x, B(y, A(2))) \tag{3.1}$$

$$A(x) = x + x \tag{3.2}$$

$$B(r, s) = \text{if r then 1 else s} \tag{3.3}$$

$$C(p, q) = \text{if p then 2 else q} \tag{3.4}$$

Condensed graphs have many concepts that are helpful to modeling plots. The H-graphs allow one to create nesting quests and the execution model of the nodes provides a method to introduce quests that must be completed before the next plot point can begin. The limitations of the condensed graphs have to do with a few things, including the execution of parallel or series quests. With quests in games, there are times that not all quests have to complete in order to continue to the next plot point; three out of five would need to complete before continuing the story. When quests have to be completed serially, there is no method for easily representing a series of quests that happen sequentially versus in parallel. Finally, there is no way to easily represent contention and dependencies on NPCs or objects or a state within the storyline to check based on the plot when dealing with complex situations like side quests, parallel or serial quests or even different a branch. It could be created but for someone not skilled in condensed graphs, it would be difficult to understand condensed graphs, their execution model, and how to relate it to a plot graph. Again, like FSM and Petri-nets, seeing an exponential explosion can occur.

3.4 UML State Machines

Unified Modeling Language (UML) state machines are an extension of Harel statecharts[89] that are object-based and adapted to UML[90]. The UML state machines introduce new concepts such as: guards, hierarchically nested states, orthogonal regions, and extending the notion of an action. Additionally, they support entry and exit actions and actions that depend on the state of the machine and the triggering event. The state machine specification defines two types of state machines: behavioral and protocol. The behavioral model is used to model behavior of entities while the protocol model is used to express usage scenarios of classifiers, interfaces, and ports.

Of interest to us is the behavioral model since it captures the dynamics of a computer program. UML state diagrams are directed graphs with nodes that are states and vertices that are transitions. A state is represented with a rounded rectangle and transitions are represented as arrows that are labeled with triggers for the event and a list of executed actions. Guard conditions are boolean expressions based on values of extended state variables and event parameters. An action is enabled when the guard transition evaluates to true. The state machines uses a run-to-completion model that assumes the state machine finishes processing each event before another event can occur.

UML state machines are the best at modeling software processes we care about when compared to the previous modeling techniques. The largest problem with UML state machines is the pure size of the specification and verbosity in cases when there are multiple actions occurring on an event. Ad-

ditionally, it is a visual formulation for complex systems so it cannot easily be separated from it's graphical representation. It also requires a large amount of textual information as well to truly understand the system. Finally, the UML notation and semantics make it geared toward computerized UML tools and not a true formal model, although the specification does make a distinction between the notation and the state machine semantics.

3.5 SAGA - Story as an Acyclic Graph Assembly

SAGA, introduced by [14], is a domain-specific language (DSL) for story management that views storylines as an acyclic graph. It attempts to provide a language for story designers that is easy to use and integrate into existing games. SAGA provides the users with a simple syntax that operates at the plot point level in an attempt to shield the designer and writer from the implementation details and focus only on the higher level story events. SAGA was designed with the help of an unnamed game studio who validated some of the design choices.

SAGA was designed for representing storylines at a high level when writing a game. Due to this, the domain specific language is ill-suited to handle the lower-level abstraction that is required to model a storyline that involves interactions between characters, objects, and overall game state.

Chapter 4

Categorizing and Describing Storylines

In this chapter, we describe how to categorize storylines in video games quickly though our tool SChar. We also present our Storyline Description Language (SDL), which allows us to abstract the storyline from the implementation. These will then be used as input to our storyline checker, SToCk, presented in Chapter 5. Our methods and tools target narrative-driven games with complex storylines such as *Fallout 3*, *The Elder Scrolls V: Skyrim*, and *Grand Theft Auto V* that may let the player affect the direction of the storyline.

4.1 SChar - Storyline Characterization Framework

Our storyline characterization framework, SChar, draws upon the characteristics of the storyline models in order to separate implementation from the model that exists, and categorizes them into the model they most closely resemble. Our framework consists of simple true/false questions that help categorize the games. Figure 4.1 shows the decision flow for categorizing a game's storyline model, based on the storyline models described in Chapter 2.

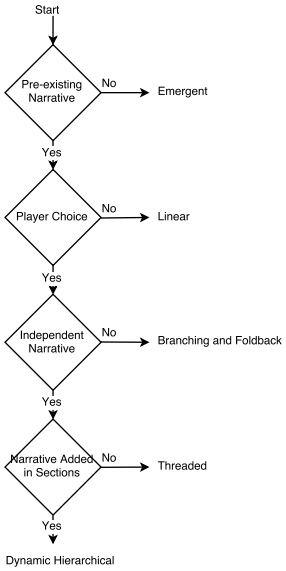

Figure 4.1: SChar Decision Flowchart

As shown in Figure 4.1 there are four main questions:

- **Pre-existing Narrative.** *SimCity* and racing games such as *Gran Turismo* do not have narratives since there is no history affecting the gameplay or story. Similarly, with OPIATE and PaSSAGE the story starts when the player starts the game. On the other hand, in games such as *Super Mario Brothers*, the game does have a pre-existing narrative e.g., Bowser has captured the Princess and Mario hears her cry for help.

- **Player Choice.** If the player's choices do not impact the storyline or there are no choices, the story is a linear story.

- **Independent Narrative.** Independent narrative exists when there are multiple storylines that progress at different speeds and allows the player

choice in how the story completes. However, if the stories can merge after a choice, then it is categorized as **branching and foldback**. For instance, the *Mass Effect* series is a very good example of a branching and foldback storyline whereas *Discworld Noir* is an example of independent narrative.

- **Narrative is Added in Sections.** The last aspect is how narrative is added to stories: if selected sections of the story are added based on a player choice, then the storyline model is dynamic hierarchical; if it is not, it is a threaded story. *Discworld Noir* and *Fallout 3* are examples of threaded storylines whereas games such as *Grand Theft Auto* and *Dragon Age: Origins* are dynamic hierarchical because additional narrative is added by a triggering function, such as competing a story in one section of the world.

Examples of Categorized Games

Iteratively refining the framework for determining storyline models through simple questions required us to categorize games. In our study, we categorized over 85 games to ensure that our framework was valid, in Table 4.1 we've listed some of the representative games. The ability to succinctly characterize all storyline models provides a starting point for the next next step: a language that can model storylines free of the storyline model or implementation context.

Table 4.1: Categorized Games

Game	Storyline Model
Grand Theft Auto V	Dynamic Hierarchical
Fallout 3	Threaded
Final Fantasy	Linear
Mass Effect	Branching and Foldback
PaSSAGE	Emergent
Super Mario Brothers	Linear

4.2 Our Storyline Description Language

Discussions with developers, testers, writers, and designers in the industry have led to the creation of the Storyline Description Language (SDL). It represents a storyline as a directed graph and uses non-deterministic finite state machines (NDFSMs) as its basis. The edge and vertex graph is augmented with additional constructs to succinctly describe a variety of plots. These constructs are used to handle cases where certain vertices are or aren't available based on a set of conditions. These conditions use plot objects that are boolean values describing a global state within the game. There are actions that can modify a boolean object when exiting a vertex. Additionally, when a player must be constrained to a subset of vertices and edges, an atomic construct is introduced.

4.2.1 Arc

The arc, or edge, represents a directed transition between plot points and also specifies the start points of the game plots and side quests. The specifics of the arc will be discussed more in depth with the various scenarios

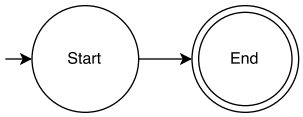

Figure 4.2: Example Arc and Plot Point

that can be created in the game plot graph.

Figure 4.2 shows a starting arc and an arc connecting two plot points together. The first plot point is a standard plot point while the second plot point is an ending plot point.

A second type of arc that will be used is the implicit arc. An implicit arc, is an arc not specified by the user but is determined by the rules of movement through the plot description model. An implicit arc is visually represented as a dashed arc between two plot points. These dashed arcs can be seen in the description of the parallel quest below.

The third style of arc is one that connects a plot object to a condition. It is represented in the same fashion as the first arc, as a normal line with an arrow pointing toward the condition or conditions associated with it. This style of arc is described more in the conditions and plot objects sections.

4.2.2 Plot Point

The plot point is the centerpiece of the plot model. It represents a point in the storyline where the user interacts with an NPC, item, or boundary that causes the story to move forward. All plot points will have arcs entering and

leaving the plot point, with the exception of ending plot points which mark the end of the story. This concept is drawn from the state of a state machine, the place in a Petri-net, and a node in a condensed graph. The plot point can be annotated with additional information such as the number of times a player can enter the plot point, the node type, and if the plot point is an ending plot point. The node type is used when describing branches, serial quests, and parallel quests. The ending plot point is shown in Figure 4.2. Visually, it is a plot point with an additional smaller circle inside of it. Figure 4.2 shows the typical plot point visual representation with the entry and exit arcs.

4.2.3 Plot Object

Plot objects are objects or NPCs in the game world that are modified by a transition into or out of a plot point. The plot objects are Boolean and are combined in various ways using the conditions object in the graph model. The plot object draws inspiration from condensed graph's idea of static inputs into nodes.

In Figure 4.3, the plot object is visually described as a box containing the name of the plot object and its default value. Arcs are used to connect conditions to plot objects and they are described in more detail in the next section.

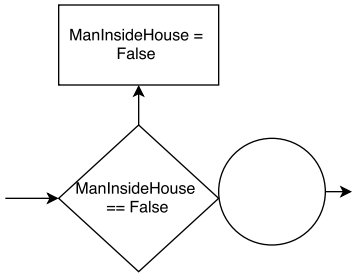

Figure 4.3: Example Plot Object and Condition

4.2.4 Conditions

Conditions came from the idea of place tokens needed to fire a transition in Petri-nets and also as a way to simplify the state machine situation of modeling having picked up a key or not at some point in the past. Conditions are tied to resources and specify a gating Boolean function of the associated resources or the modification of resources along an arc. The two cases are, of course, pre- and post-conditions. In the pre-condition state, the conditions check a Boolean function created using resources. As a post-condition, they modify a resource or resources. In the graph model, a pre-condition state is modeled as a 45-degree rotated rectangle located before the plot point, accepting all incoming arcs. A post condition is a 45-degree rotated double rectangle occurring directly after a plot point and is the source for all outgoing arcs.

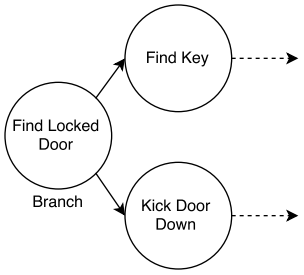

Figure 4.4: Example Branch

4.2.5 Branch

A branch decision occurs when a player has the choice of which plot point to transition to next. Described by the model, a branch is described by multiple arcs leaving a plot point and the plot point being annotated as a "branch". If the plot graph is created in a way that a cycle occurs with the main branch node, an additional annotation specifying the number of times the user can enter the node can be added, otherwise it assumes the user can enter it an infinite number of times. The branch can be thought of as a state in an NFA with a few enhancements to make it easier for a person not skilled in the art of state machines to model a game plot. An example of a branch can be seen in Figure 4.4.

4.2.6 Quest

In games, there are times where multiple quests are given to the user and the user has the ability to finish them in any order they see fit. From this there are two possibilities: the user can complete them in parallel switching

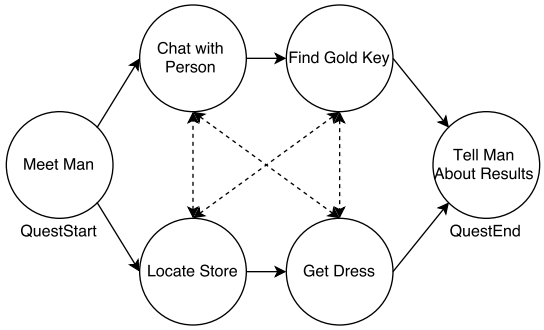

Figure 4.5: Example Parallel Quest

between the tasks of each quest or they must choose one, complete it, and start on the next one. The simple modeling case is modeling parallel quests; serial quests require the use of atomic sections (described in a section below). All quests must end at a common point denoted as a quest endpoint.

Figure 4.5 depicts a simple parallel quest and the implicit transitions that can take place when completing a parallel quest. In the case of a series quest, as each quest is completed the player must return to the starting plot point and begin from there again. Whereas, in a parallel quest, the user can visit all the plot points and complete all the quests before returning to the start plot point to receive all the rewards and quest completions before moving onto the plot point after all quests are done. In the serial quest case, it is still the same as the parallel case except that each quest in the serial quest is enclosed in an atomic section. Figure 4.6 depicts a serial quest.

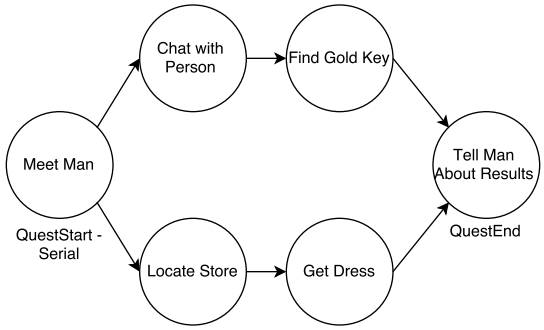

Figure 4.6: Example Serial Quest

4.2.7 Side Quests

Side quests are a single quest or set of quests that do not pertain to the main plot line. These quests normally become available after a certain plot point or criterion are met within the game and can be completed until another point or criterion has been fulfilled in the main plot line. A side quest is specified just like the main plot graph; however, the starting arc is annotated with name or number of the plot point that must be completed in order to start following the side quest plot graph and the arc exiting the last plot point of the side quest is labeled with the plot point name that the side quest must be completed by. As an example, in Figure 4.7, the side quest can be started after B has completed and D must be completed before C is completed, otherwise, the side quest will be shut off from the user.

Side quests are very much like a condensed graph in that it is another graph that can be executed and is self-contained when discussing itself. It can still have effects on plot objects in the main quest.

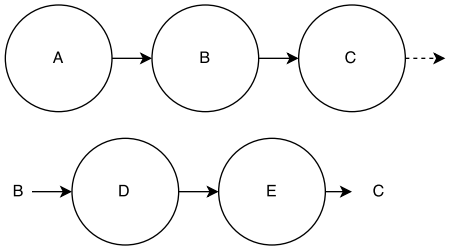

Figure 4.7: Example Side Quest

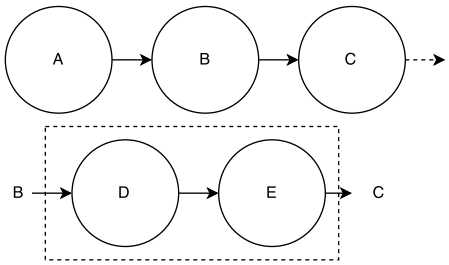

Figure 4.8: Example Atomic Section

4.2.8 Atomic Sections

There are some cases where once a player begins a quest or section of the story, they should not be able to interact with other plot points from side quests or a parallel quest. In order to visually show an atomic section, as in Figure 4.8, a box is drawn around the section of the plot graph that must be completed before the player can interact with other valid plot points.

In the example above, once the user completes the first side quest plot point, D, they must continue with the side quest and complete E before returning to the main plot line.

Atomic sections can be thought of like a more powerful H-graph seen

in condensed graphs. They are a sub section of computation, or in this case, plot, that the user can interact with and must finish before continuing on in further computation.

4.3 Implementation Issues

The two most obvious solutions are XML and JSON (JavaScript Object Notation). After again speaking to many testers and engineers, we chose to implement our SDL based on JSON file, a lightweight data exchange format, that is user readable and easy for machines to parse and generate. It is a well-understood format used in many tools within the industry and allows for simple debugging when problems occur.

The JSON file has two top-level elements: a string element name and an object named plot. The name corresponds to the name of the plot and the plot object contains objects for plot points, plot objects, conditions, and atomic sections.

Plot Point

The plot point is the base element of the plot model and in this representation encapsulates other concepts in the plot model described in the previous chapter. A plot point has the previously described values: attributes, pre-conditions, post-conditions, outgoing arcs, the number of times the user can re-enter the plot point, and, if it is a side quest plot point, the gating node names as to when it can be accessed. Additionally, it has name, description,

start point, and end point attributes so the user of the tool can annotate the plot point with additional information and describe a plot point as the starting point of the story or an acceptable end to the story.

Attributes is a list of attributes for the plot point and it can have the following values: normal, branch, quest start, quest end, loop back, start, end, side quest start, and side quest end. If there are no attributes defined, it is assumed to be a normal plot point.

The pre-conditions and post-conditions are an array of strings that are the names of conditions defined in the conditions subsection. The outgoing arcs are defined as an array of strings that contains the names of each plot point that is a possible next plot point. Finally, the number of times the user can re-enter a plot point is considered to be one unless the type of the plot point is changed to serial or parallel, which at that time it becomes equal to the number of outgoing arcs from the plot point.

The side quest start gate plot point and side quest end gate plot point attributes are only needed if the plot point is going to be involved in a side quest. In the standard case of a game not containing any side quests, these values will not be set.

Arcs

An arc is represented inside of the plot point as an array of strings. Each explicitly defined next plot point for the current plot point is placed into this array with the name outgoing.

Branches

Creating a branch for the branch and foldback is very simple. All that has to be done is provide the explicit next nodes for the branch in the outgoing attribute and set an attribute for branch. If there is a case where the user could constantly go through this plot point multiple times, but the game requires it to be constrained, the number of times allowed attribute will have to be set to prevent it from being able to be an infinite loop-back.

Quests

Quests are defined by a quest start and quest end nodes. There are two types of quests that must be supported: parallel quests and serial quests. A parallel quest is represented by defining a quest start point and a quest end point then providing the quest plot points between the quest start and quest end points.

A serial quest is defined in much the same was as the parallel quest, except each quest becomes part of an atomic section. The plot points are much the same as the parallel quest but the addition of atomic sections creates the serial quests.

Plot Object

A plot object, as described, is an object that has a value of true or false. When representing one using JSON, it must have a name and a value. Optionally, it can contain a description. It is then placed within the plotObjects

subsection.

Condition

Pre- and post-conditions are both defined within the condition subsection. They become pre- or post-conditions when they are defined in the plot point precondition or postcondition attribute. Conditions have four attributes: name, description, check, and action. Name and description are both strings that provide a name and human readable description to the condition. Check is an array of strings that provides a list of statements that must all be true. The statements are boolean statements that reference plot objects defined in the plot objects subsection. Finally, action is an array of strings with simple assignment statements such as `manIsOutsideTheHouse == false`.

Side Quests

Side quests are represented through a side quest object defined in the JSON file. Side quests are objects that contain the following information: name, description, a starting gate plot point, an ending gate plot point, and a list of plot points that represent the side quest. The starting gate and ending gate plot points are plot points that specify when the side quest becomes available and when it is no longer available.

Atomic Section

The last concept that has to be modeled is atomic sections. Atomic sections are defined within another sub section called atomic. Each subsection is defined as arrays of string that contain the list of plot points and other atomic sections associated with the atomic section. Atomic sections cannot have circular references.

4.4 SDL Output

The output of our SDL is a JSON file that will be used as input into our formal verification tool for complex game plots, discussed in the next chapter. An example out of SDL is shown below and is representative of the *Star Wars: Knights of the Old Republic* quest which has been referenced within this dissertation.

```
{
    "name": "Star Wars Example",
    "plot": {
        "plotPoints": {
            "startPoint": {
                "attributes" : ["START","QUEST_START"],
                "name" : "Starting Point",
                "description" : "A man gives two quests",
```

```
    "outgoing" : ["findDaughter", "findSon"]
},
"findDaughter": {
    "name" : "Found daughter",
    "outgoing" : ["reportDaughter"]
},
"reportDaughter": {
    "name" : "Report Daughter",
    "postcondition" : ["moveManInsideHouse"],
    "outgoing" : ["endPoint"]
},
"findSon": {
    "name" : "Found son",
    "outgoing" : ["reportBackAboutSon"]
},
"reportBackAboutSon": {
    "name" : "Report Son",
    "precondition" : ["isManOutsideHouse"],
    "outgoing" : ["endPoint"]
},
"endPoint": {
    "name" : "End Point",
    "attributes" : ["END", "QUEST_END"]
```

```
            }
        },
        "plotObjects": {
            "manIsOutsideTheHouse" : {
                "name" : "Man is outside the house",
                "value" : true
            }
        },
        "conditions" : {
            "isManOutsideHouse" : {
                "name" : "Man is outside his house",
                "check" : ["manIsOutsideTheHouse == TRUE"]
            },
            "moveManInsideHouse" : {
                "name" : "Move man inside his house",
                "action" : ["manIsOutsideTheHouse = FALSE"]
            }
        }
    }
}
```

Chapter 5

StoCk: Storyline Checker

In this chapter, we analyze several formal verification techniques and discuss our choice of methodology for verifying complex game plots and how we map it to our problem domain. We then use this methodology to create a Storyline Checker (StoCk) tool based on SPIN, a well-known open source model checker.

The story lines are described by the story line description language (SDL) presented in the previous chapter. StoCk takes the storyline description language SDL files to formally verify that the storyline does not contain inconsistencies. This chapter describes: choosing a formal verification technique, how to represent an SDL to the formal verification tool, and how the StoCk ties all the previous work together into a usable tool for the gaming industry.

Finally, we present a case study for representing our storyline description language in SPIN[91] using a storyline sample from *Star Wars: Knights of the Old Republic*.

5.1 Formal Verification Methods and Their Trade-Offs

As described in Chapter 2, the following three techniques for formal verification seemed the best choice for storyline verification: theorem proving, SAT solving, and model checking.

Theorem provers are often human-driven and the automated methods are often slow due to the methods used to solve the equations. However, theorem provers do not map easily to our problem domain. So using them would be a very poor fit. Additionally, theorem provers rely upon equations to be solved; if these equations are written incorrectly, they become either impossible to solve in a reliable amount of time or become unsolvable.

SAT solvers are not human-driven and are automated, but they do operate on the same formulas as the theorem prover techniques discussed above. Although they can be faster, SAT solvers can become much slower if the equations are created in a particular method that does not align to the strength of SAT solvers. Again, like theorem provers, SAT solvers are not designed to solve pathfinding problems easily; which leaves model checking.

Model checking has been used to solve many problems in the software and hardware domain. These problems, specifically distributed and parallel software execution verification, map nicely to the plot description model. Additionally, model checkers work by providing a counter-example when they fail, which provides the user with a path that proves the logic does not work. The model checker's ability to work well within constrained problem spaces

has also been proven through many domain spaces such as: protocol verification, algorithm verification, parallel and distributed system verification, and software execution. Additionally, since model checkers operate on finite state machines, the use of a model checker for validating storylines makes more sense because storylines are often modeled as directed graphs or flowcharts. On the downside, much like SAT solvers and theorem provers, model checkers are based on heuristics and can explode under some circumstances. However, model checkers work on finite state machines and match very closely the storyline description language. Model checkers provide a solid foundation upon which to build a storyline verification tool.

The SPIN model checker is our formal verification tool of choice because the problems SPIN verifies: parallel and multi-threaded computation and software execution, align closely with the storyline models we want to verify. SPIN uses state-of-the-art model checking techniques such as *on-the-fly, partial order reduction*, and *BDD-like state storage* to handle exploding state space[1]. It also provides a modeling language, **Promela**, that matches the typical execution of a storyline since it is user-driven and thus nondetermenistic. Finally, SPIN has a large support community, is actively developed, and can run on any computing environment which is important for development tools.

[1] `http://spinroot.com/spin/what.html`

5.2 Direct Finite State Machines Representation versus Leveraging Pomelea's FSM Abstractions

Formal verification of protocols, models, and algorithms are often done to verify correctness and whether the systems are stable under all inputs. In the case of storyline verification, this means being able to complete the storyline no matter the path taken through the story. This section describes how to represent storylines described in SDL in SPIN. This can be done either directly as finite state machines (FSM) or leveraging Pomela's higher-level state descriptions.

5.2.1 Using a Non-deterministic Finite State Machine

In order to prove that storyline validation is possible, we will examine the example problem from the introduction modeling it as a non-deterministic finite state machine that will be used as input into the SPIN model checker. The SPIN model checker uses partial order reduction, bit-state hashing, and bounded context switching to prevent the state space of the problem from becoming too large and constraining the state space to a computational space that can be searched in a time acceptable to the majority of companies within the hardware and software verification domain. Using a non-deterministic finite state machine should be straightforward since model checkers operate on finite state machines.

The example is quite simple. It starts with a parallel quest where the quest giver gives the player two quests: find his daughter and find his son.

Looking at Figure 5.1, the upper path is the quest of finding the quest giver's daughter. In the plot point Find Daughter the user has found the daughter, which then gives the player the ability to return to the quest giver where a check is done to make sure the quest giver is outside his house. If the check passes, the user is given a cutscene where the quest giver expresses gratitude for finding his daughter then goes inside the house. Leaving the cutscene, the resource of the man being outside is set to false, where it was initially true. In the second quest, the player is asked to find the man's son who has mysteriously disappeared. Following the man's request, the player eventually will find the mans son in the desert dead from a nasty fall and will want to report back to the man. Before beginning the cutscene revealing the fate of the man's son, there is a check to see if the man is outside the house. If he is not, the cut scene cannot move forward. Finally, when both quests are complete, the man tells the user of another person they will want to speak to so they can leave the village safely.

With the parallel quest understood, the next step is to convert the plot model into a non-deterministic finite state machine. The resultant state machine is shown in Figure 5.2. This state machine is modeled in Promela using a do-od loop with a gating check on previous states and potential next states that non-deterministically chooses the next statement to execute based on available statements.

Running the encoded FSM results in the following snippet seen in Figure 5.3, showing that an accepting state cannot be found and where the prob-

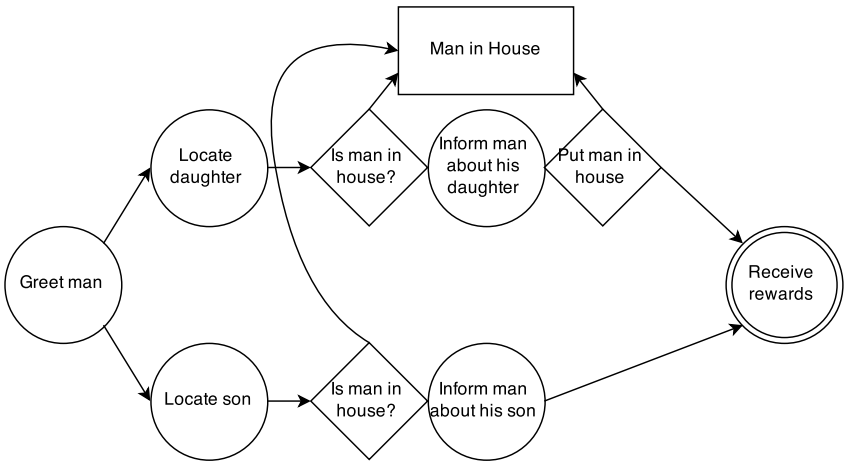

Figure 5.1: Parallel Quest Example modeled in the SDL

lem occurs.

The image above shows the output after running the FSM through the SPIN model checker, in which there is an error and a trail file is created. Invoking SPIN again with an option to read the trail file produces the shown output in Figure 5.4.

Figure 5.4 shows the eight steps it took to find an invalid endpoint in the finite state machine. Of course, in the state machine, it moves from state 1, to state 2, to state 4, to state 3, at which point it could not go any further. This example shows that our plot model can be encoded as a FSM, checked for validity, and errors found.

Figure 5.2: Parallel Quest Example as an NDFSM

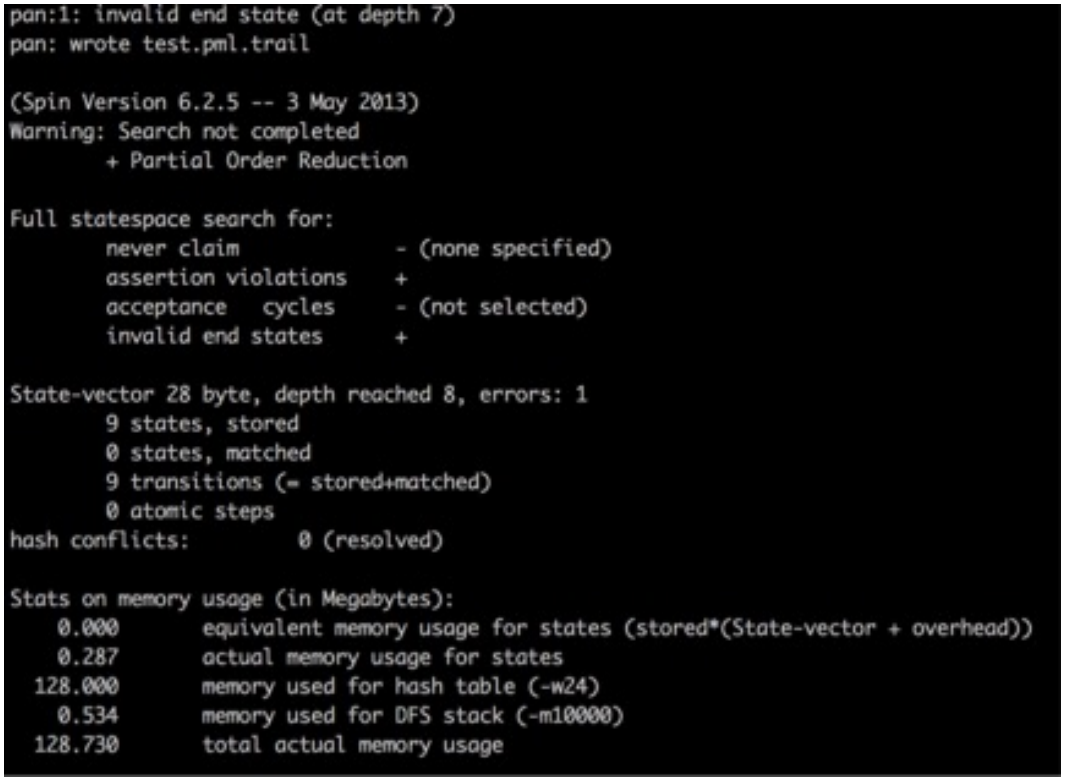

Figure 5.3: Error message for FSM

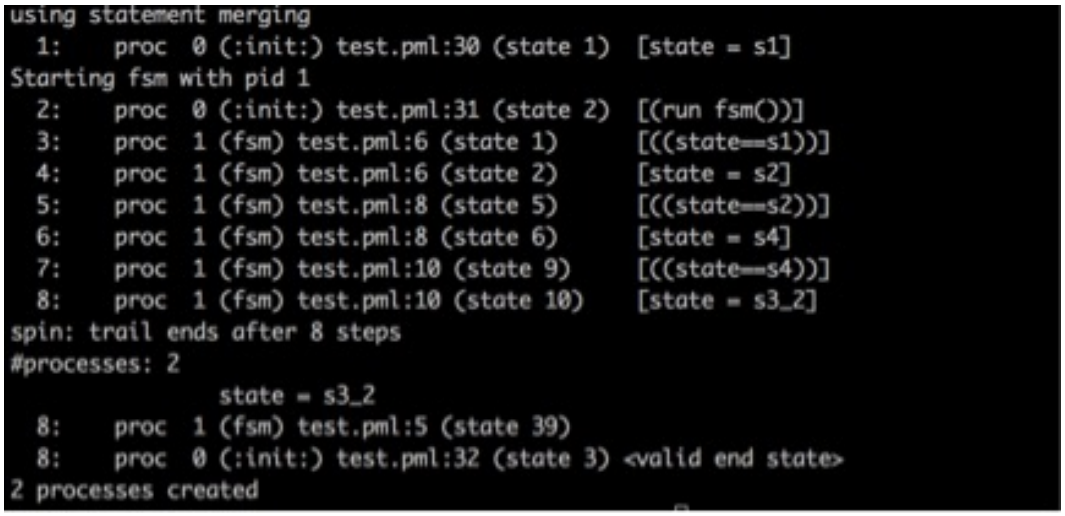

Figure 5.4: Error Trail for FSM

5.2.2 Leveraging Promela's Abstractions

In the previous section it was shown that the plot description model could be accurately described as a non-deterministic finite state machine. However, this is not the best fit for implementation since FSMs have exponential blow up in many common game situations. The SPIN model checker, used above, has a modeling language, Promela. Promela provides a syntax and constructs to more easily model more complex algorithms and processes, and a storyline can be thought of as a process.

The conversion into Promela for the software description language shown in Figure 5.1 is close to the non-deterministic finite state machine method in that a do-od loop is used to represent each state; however, the gating statement makes use of the plot objects and the actions taken in each state can involve adjusting the value of the plot object.

Running the Promela version of the scenario again shows that there is an error and that it occurs in the same manner as the first case study. This proves that storylines can be encoded using the Promela language, checked for validity, and errors found. Thus, we can convert storylines described in SDL to Promela to be formally verified using SPIN.

5.2.3 Analysis

Both methods have shown that they are valid. However, encoding our SDL directly into FSMs results in larger file size and more states that must be converted . It also does not take advantage of SPIN's Promela modeling language, which also provides a higher-level abstraction of FSM which lets our SDL be described more succinctly. We therefore chose to convert our SDL using Promela's abstractions to prevent larger file sizes, allow for easier conversion, and easier error analysis when a problem arises.

5.3 Implementation of StoCk

There are some large design decisions that need to be done before coding up a verification system. The first is choosing the programming language. After speaking with multiple testing and quality assurance engineers(pers. comm. Jason Frueh), it was found that Java is the underlying language that most of the tools they use on a daily basis are written in. We thus chose Java as the language of choice since it makes it easier for game developers to provide customizations and integrations in a language they are already familiar with.

The verification system has some important tasks: reading the plot information from disk, parsing that information into objects in memory, converting it into a file that can be used by the underlying verification system, running the verification system, and finally, reporting to the user the results of the verification. The rest of this section describes the processes in each of these steps.

Converting Storyline Description Language into Promela Code

Once a plot has been defined in the JSON format, it must be converted into a Promela file to be executed by SPIN and a C compiler. This process can be divided into two steps: 1) reading in the SDL from a file and then parsing the JSON file into the plot model and 2) creating a Promela file from the plot model data.

Reading and Parsing JSON

Reading and parsing JSON can be accomplished with a Java library named Jackson[2]. The Jackson library provides the output in a tree format that is used by the verification system to transform the branches and leaves into the various Java objects representing the plot model. The Java objects are almost exact replicas of the JSON format described above; however, a few changes have been made in order for the creation of the Promela files to be done with less hassle.

[2] `http://wiki.fasterxml.com/JacksonHome`

The first simplification is moving the conditions for a plot point into the plot point structure itself instead of representing it as an object in the path toward the plot point. This allows the code to easily reference any pre- or post-condition associated with itself when it is time to create the Promela file.

The next enhancement is to create the explicit arcs between the plot points in both incoming and outgoing directions even though the flow is purely a single direction. This enhancement allows the program to determine its position accurately enough for the creation of atomic sections and other positional attributes needed when creating a Promela file.

The final enhancement to the classes is including a reference to the name provided for each object in the JSON file to the class associated with it. Using these references, Map interfaces are used to quickly lookup the objects during processing.

Creating a Promela File

After the JSON file has been read and parsed into the plot model objects, a Promela file must be created. In order to do this, we again have to map our data model onto a model that works in Promela. The model we are using is based off of a finite state machine. Instead of determining which plot point to go to next after getting to that state, the state is determined as a set of preconditions and SPIN will randomly choose the next state based on these preconditions when formally verifying.

The Setup

In order to create the Promela file, there is a basic framework that must be set up for every file. This includes defining true and false as 1 and 0, respectively; creating a list of all plot points completed that is initialized to false and is the same size as the number of plot points in the plot model; creating an end game flag initialized to false; creating flags for each atomic section specifying if the atomic section is active; creating a `proctype` named `gameRunner` which contains the contents of our plot model to be model checked; and finally, the init block which causes our storyline definition checker to be started. Each plot point will have a row in this method and it consists of: a preconditions blocking statement, actions, and setting the plot point to being run as true.

Creating the Plot Point

Plot points, as mentioned above, are defined by a blocking statement, actions, and setting the plot point to being run as true. If it is an end game plot point, it must also set the end game to flag true. There are, of course, a few different cases for each plot point, but the process is close to the same for each type of plot point be it a starting plot point, or a branching plot point. We will examine each of the components of the plot point within the Promela file.

The first section is the pre-condition blocking statement. These statements prevent plot points from being chosen as a possible next point to be run

by SPIN when being executed. If multiple plot points are available, SPIN will randomly pick one. The first part of the blocking statement to be generated is making sure the explicitly specified incoming plot points are completed. Depending on the type of plot point, this could either be a Boolean OR or an AND. Next, atomic blocks are checked to see if the plot point is either starting an atomic block, ending an atomic block, or simply inside an atomic block. Following the atomic blocks, the pre-conditions are checked if any exist for the plot point. After that, side quest specific options are checked. If the plot point is in a side quest, then the gating plot point conditions have to be applied. Finally, a check is put in place to make sure this plot point has not already been completed.

After the blocking statement has been created, the actions that occur after the user acts upon the plot point are done. The first step is to set the plot point as being run. The next is to set the end game flag, if the plot point is an ending plot point. After this, the atomic section processing takes place. The plot point is checked if entering or leaving an atomic section and setting atomic section flags correctly. Finally, the post actions are created in the same way as the pre-conditions are done in the blocking statement.

5.4 Running StoCk

After converting the JSON file into the plot model then generating the Promela file, it must be run by SPIN. This is accomplished by using Java's ProcessBuilder classes to execute external program commands.

The first step is to write the Promela file to a temporary directory, then use the SPIN compiler to create the associated C files to create the model checker. After the C files are created, the C compiler is run to create an executable. Next, the executable model checker is run and its output stored to a Java String. If the return code for the model checker execution shows an error state, SPIN's trace facility is run against the file to generate the counter example of a condition not holding.

5.5 Returning Results to the User

StoCk has two results that must be returned to the user: the verification was successful or the verification failed. In the failure cases there are a few different cases. These cases are: failure to execute the tool, failure to execute all the steps, or failure from the verification tool. In each of these cases, the result is returned in a VerificationResult class that stores the output of the raw results to the console. The data within this object are then displayed to the user.

The next chapter provides three case studies on using StoCk in a game development setting.

Chapter 6

StoCk Case Studies

The previous chapter described the implementation of StoCk, this chapter provides case studies for the three main use cases of StoCk using a real world example. In our case: *Fallout 3*. *Fallout 3* is one of the best examples of style of games this research is aiming to help: it is well known, has many resources freely available on how to complete the game, and has many walkthroughs and guides to reference when creating the plot model. Additionally, it is an action adventure RPG with a threaded storyline with 11 quests for the main plot and 16 side quests. There are three case studies that each build upon one another: quality assurance for the writer's storyline, quality assurance for the developer's implementation of the storyline, and the development of a quest within *Fallout 3*. The last case study most closely resembles how a storyline is developed in industry and by consumers modding (editing) the game to create new content or fix bugs found within the game that the developers have not fixed.

6.1 Quality Assurance for Writers

One use of the plot verification tool is as a quality assurance tool for the writers when they are creating the storyline for the game. In this case study, we examine the complete storyline from *Fallout 3* from the writer's perspective after the work as been complete. The data for the storyline will come only from external sources and not examine the implementation of the storyline within the game. This section will describe: how data was obtained to create the storyline, how the storyline description language (SDL) files were created, and the results of the study.

6.1.1 Where the Data Was Obtained

The data to create the storyline definition language files were gathered from various sources. First, *Fallout 3* itself, since *Fallout 3* displays the name of the quests the player is currently on. Secondly, through the official strategy guide written in conjunction with the *Fallout 3* development team [92]. Finally, the multiple walkthroughs and frequently asked question guides that are found on the Internet through GameFAQs [1]. The most referenced was the Fallout Wiki[2] because it was the most up-to-date resource and contained a list of all glitches and errors found within *Fallout 3* on each platform it is available on.

[1] `http://www.gamefaqs.com/pc/918428-fallout-3/faqs`
[2] `http://fallout.wikia.com/wiki/Portal:Fallout_3`

6.1.2 How the SDL Files Were Created

The storyline description language files were created by transcribing a single quest at a time through first using the official walkthrough then supplementing and verifying the paths. During transcription a few choices had to be made as to what level of abstraction to use when modeling the quests. It was decided that:

- A character that can be killed will have a plot object relating to their aliveness
- If a quest relies on another quest being complete to be a trigger, a plot object is made to specify if the quest is complete
- Dialog is not modeled but plot points arising from conversations are
- If an item is required for a quest a plot object is created to specify if the user has the item

The transcriptions were done on a per quest basis since iteration is the standard way game content is created. Additionally, each quest was tested individually at multiple points to verify the creation of the Promela file and that the storyline being encoded matched that in the walkthroughs and FAQs. After completing the quests according to the guides, the overall Fallout storyline appears to be a straight line with many side quests that have very little interaction with one another as seen in Figure 6.1.

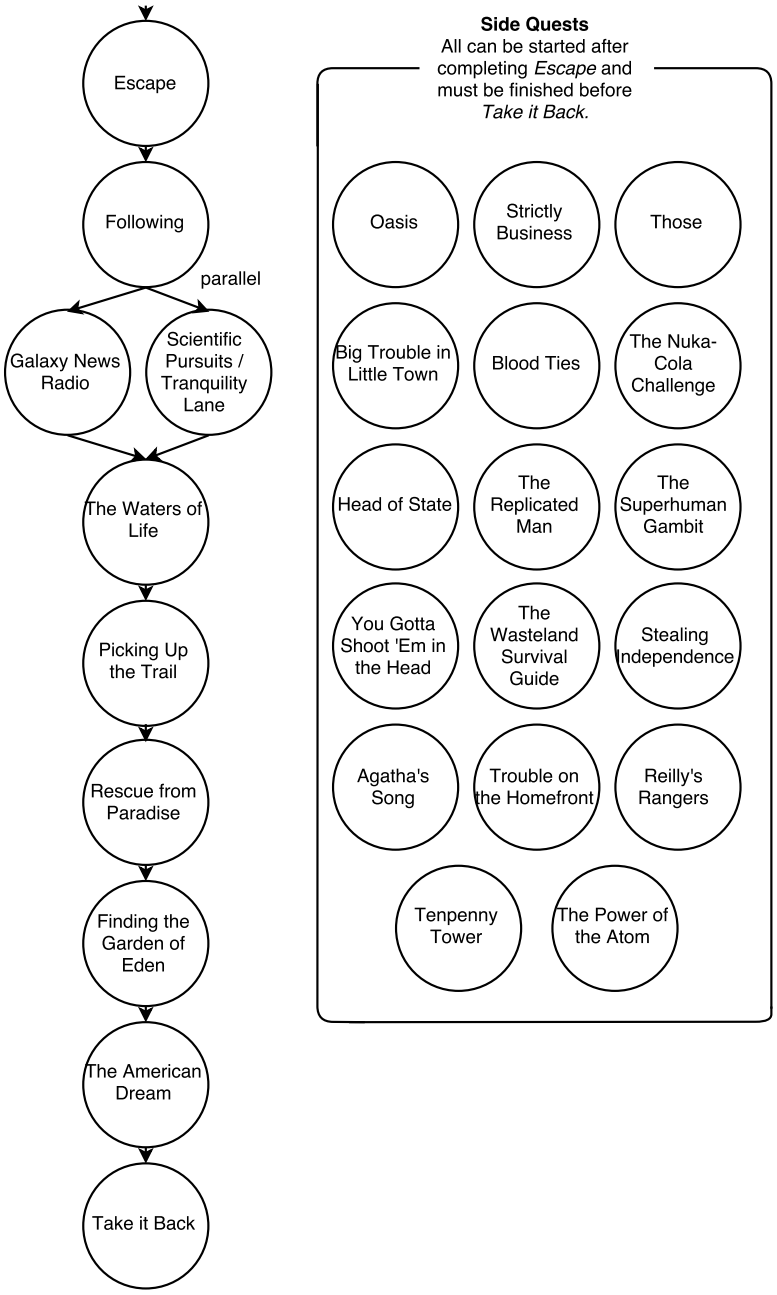

Figure 6.1: Fallout 3 Storyline based on Guides and FAQs

The next section presents in-depth findings about the interactions between the quests during their creation and results of the verification process.

6.1.3 Findings

The key findings of this study can be broken down into three parts:

- SDL file creation process
- discovering inconsistencies between the game guides and the game
- developing a tool to validate SDL files with acceptable performance

SDL File Creation During the SDL creation it was found that there was not much interaction between any quest, be it a side quest or a main story quest. Each quest was insulated from one another with a few exceptions (see Table 6.1). The interaction between Following and Galaxy News Radio is that of a plot object. If the player has killed a person that gives them information at the end of the Galaxy News Radio Quest, the information must be retrieved in a different manner. Galaxy News Radio interacting with the Scientific Pursuits / Tranquility Lane is a different style of interaction but still modeled as a plot object. If the player finishes the Scientific Pursuits / Tranquility Lane quest before a given point in Galaxy News Radio, the player is given a different reward since the information given to the player is how to begin the Scientific Pursuits / Tranquility Lane quest. Next, Finding the Garden of Eden and The American Dream interact on a plot object of a given

Table 6.1: Quests with Interactions

Following	Galaxy News Radio
Galaxy News Radio	Following, Scientific Pursuits / Tranquility Lane
Rescue from Paradise	Strictly Business
Finding The Garden of Eden	The American Dream
Tenpenny Tower	You Gotta Shoot 'em in the Head

character being alive like Following and Galaxy News Radio. However, in this case, having the character alive allows the player to gain them as an ally for the last quest of the storyline. Finally, Tenpenny Tower and You Gotta Shoot 'em in the Head again have the same interaction with a plot point based on a character being alive. If the character is not alive, then a possible quest branch is not allowed.

When the SDL files were first being created, the decision was initially made to model the entire storyline in a single file. However, when validating the storyline after each additional quest was added, it was found that the validation process would not complete when left running for over twenty-four hours once the main quest chain of Escape, Following, Galaxy News Radio, and Scientific Pursuits / Tranquility Lane existed. Two different approaches were taken to alleviate the problem: allow SPIN to run in parallel breadth-first-search mode and using SWARM [93]. Allowing SPIN to run in parallel breadth-first search (BFS) mode did not work because the memory requirements to complete the validation were greater than what the machine had on which it was running (16GB). The second approach was to use SWARM, an

extension of SPIN that can break large verification problems down into many small verification jobs that run in parallel. In this case, the job would complete but the results returned were incomplete in that some jobs reported that they could not verify some paths and others jobs verified the path. As a judgement call at the time, it did not appear appropriate to use SWARM because all jobs did not report completely correct verifications. The decision was made to implement each quest individually and combine quests where interactions between them existed.

Discovering Inconsistencies The validation code found inconsistencies in the FAQs and game guides in the interaction between the Tenpenny Tower and You Gotta Shoot 'em in the Head side quests. A character, Allistair Tenpenny, is central in both of these quests. If Allistair Tenpenny is killed during You Gotta Shoot 'em in the Head, one possible path from the Tenpenny Tower quest becomes unavailable or is automatically ended if it was already started. If Allistair Tenpenny is killed during Tenpenny Tower, a path in You Gotta Shoot 'Em in the Head becomes unavailable. In the official strategy guide, no mention is made of Allistair Tenpenny's involvement between the two quests causing an issue; however, playing the game shows it to be an issue.

Otherwise, the plot of *Fallout 3* was found to not have any inconsistencies using our tool in this scenario. The reasons for this are that many of the techniques discussed in the abstract, such as immortal characters and quest isolation are heavily used. There are no side quests that interact with the main

quest in any way. Another aspect to missing possible inconsistencies is due to the after-the-fact process upon how this case study was done. Unfortunately, besides the inconsistencies found when creating the storyline description files based on the data available there was no method for us to truly put ourselves into the designers shoes.

Creating the tool The verification tool must be able to verify quests quickly or else it loses much of its utility to game writers. As a guideline, we believed it should be able to verify a quest within ten seconds. As part of the testing, the tool was run 1000 times to ensure the results were consistent and to generate a large enough number of runs to accurately calculate the run time of each quest. The statistics of the runs can be seen in Table 6.2. The longest running quest to test was The Wastelands Survival Guide which took about four seconds on average to verify.

It was found out during the creation of the plot files that the largest plot verification file that can be solved within a reasonable time by the underlying program is approximately 130 plot points with a few branches on a modern machine (2.7GHz Intel i7, 16GB RAM). As mentioned above, the storyline had to be broken down into individual quests and quests that have interactions between them. The longest quest to verify was The Wasteland Survival Guide which was not the largest quest in terms of plot points but it was the most complex having multiple branches that had to be explored. Although not ideal to have to break the storyline down into multiple quests files, it does fall in line with industry practices (pers. comm. Royal McGraw); quests are proofread

Table 6.2: Fallout 3 Quests and Averge Time to Validate

Quest	Time (ms)
Agatha's Song	292.11
Blood Ties	245.33
Big Trouble in Little Town	244.95
Escape!	253.19
Following	253.24
Finding the Garden of Eden	282.98
Galaxy New Radio	252.44
Head of State	244.91
Oasis	241.84
Picking up the Trail	249.82
Rescue from Paradise	246.81
Reilly's Rangers	242.63
Strictly Business	281.86
Stealing Independence	249.98
Scientific Pursuits and Tranquility Lane	266.67
The American Dream	244.92
Those	259.75
Take it Back	246.87
The Nuka-Cola Challenge	241.24
Trouble on the Homefront	260.87
The Power of the Atom	240.94
The Replicated Man	243.85
The Superhuman Gambit	244.62
Tenpenny Tower	245.80
The Waters of Life	248.77
The Wastelands Survival Guide	4154.13
You Gotta Shoot 'Em in the Head	243.35

and logically tested by designers on an individual basis. Some companies do no such proofreading and testing and assume a critical path check will catch the issues. The ability to allow a designer to easily verify that their quest can be completed based on formal verification is very powerful and can provide additional safety around quests that cannot be completed.

6.2 Quality Assurance for Implementation

The second case we explore is the use of the tool as a software developer in test verifying the implementation of the story. In this case, the SDL files will be created from the implementation and then verified using the tool. As in the first case study section, where the data was obtained, how the files were created, and results will all be discussed.

6.2.1 Where the Data was Obtained

The data was obtained from *Fallout 3: Game of the Year Edition* on the PC using G.E.C.K.[3], the world editing tool from Bethesda, the game's creator. G.E.C.K. provides a GUI for creating, modifying, and browsing all assets of the game. Using this tool we were able to examine each quest in detail. This included all in-game scripts and variables used to track the quests completion status and world state. We also used the Fallout Wiki[4] to determine what glitches existed within the PC version of the game.

[3] `http://geck.bethsoft.com/index.php?title=Main_Page`
[4] `http://fallout.wikia.com/wiki/Portal:Fallout_3`

6.2.2 SDL File Creation

The Storyline Description Language files were created by examining the data within the G.E.C.K. toolkit. The first step was modeling the full storyline then creating each quest as need be if it deviated from the already created storyline quests from the first use case.

6.2.3 Findings

The first step was to model the full storyline as implemented to see if it matched the storyline as specified by the writers. We did not assume any bugs or inconsistencies due to game engine bugs and glitches such as warping between zones and going though walls and doors – which is a vast majority of the bugs reported and still found within the game [5]. The quests the player could get to were determined by how the player could get to the quest within the game world. The storyline as implemented can be seen in Figure 6.2, the side quests are the same so they are not repeated in the figure. Obviously, this does not match what the writers created.

From this it can be seen that the user can circumvent all quests between Escape and Tranquility Lane by going directly to Tranquility Lane. In terms of game play, when Escape! is completed, the player is outside the starting vault instead of heading to the closest town, the player can find Vault 112 and sit in a lounger in the vault to begin Tranquility Lane.

[5] `http://fallout.wikia.com/wiki/Fallout_3_quests`

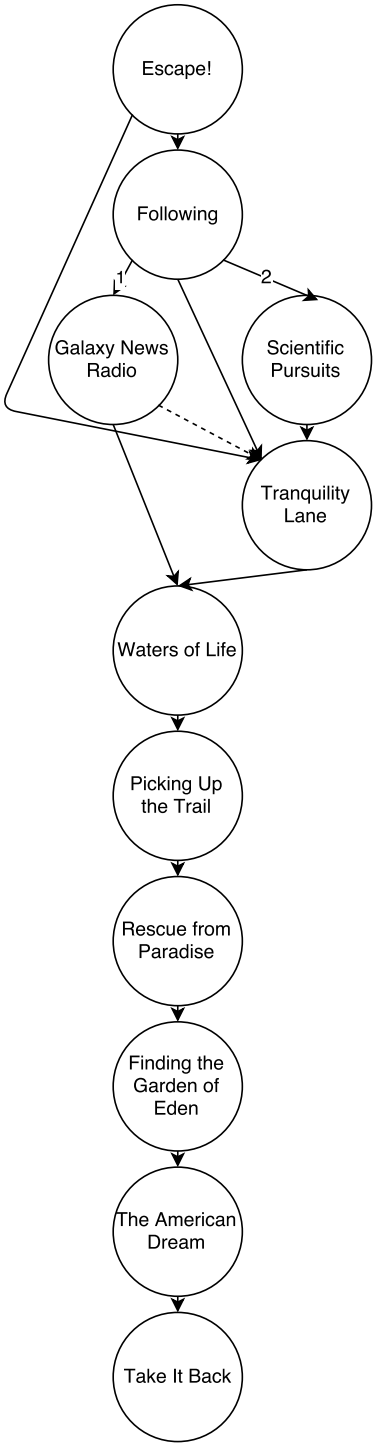

Figure 6.2: Fallout 3 Quests as Implemented

To fix this problem, the use of a plot object and pre- and post-conditions must be used to only allow the starting of a quest after the player has finished the correct quests. Inside the game, the problem can be fixed by modifying (or adding) scripts associated with the quests to not allow the quest to start until the preceding quest has moved into the completed state. We verified that this can be fixed by adding plot objects and condition checks such that when each quest is complete, that check is used to prevent the player from going to the next quest unless the previous one is complete. The next step was to verify the implementation of the quest matched the writers' designs. It was found that the implementation of the quest matched the models that were created in the previous use case.

The next step is to analyze the storylines and quests based on glitches that are known. Assuming the commonly known glitches that can be used within *Fallout 3* easily, the reexamination of the storyline looks like Figure 6.3. Notice that the user can now access up to the Picking Up the Trail quest without completing the other quests and the quest Escape can also not be completed. As with the initial storyline examination, the reason the player can access the quest is due to the implementors not inserting gating functions into the quest's scripts for *Fallout 3*. Again, following the same process, we can verify that storyline is now unable to be circumvented by using glitches within the game engine.

Finally, if we assume the player can access any section on the map with no problems, the issues seen in the first two iterations are seen again and

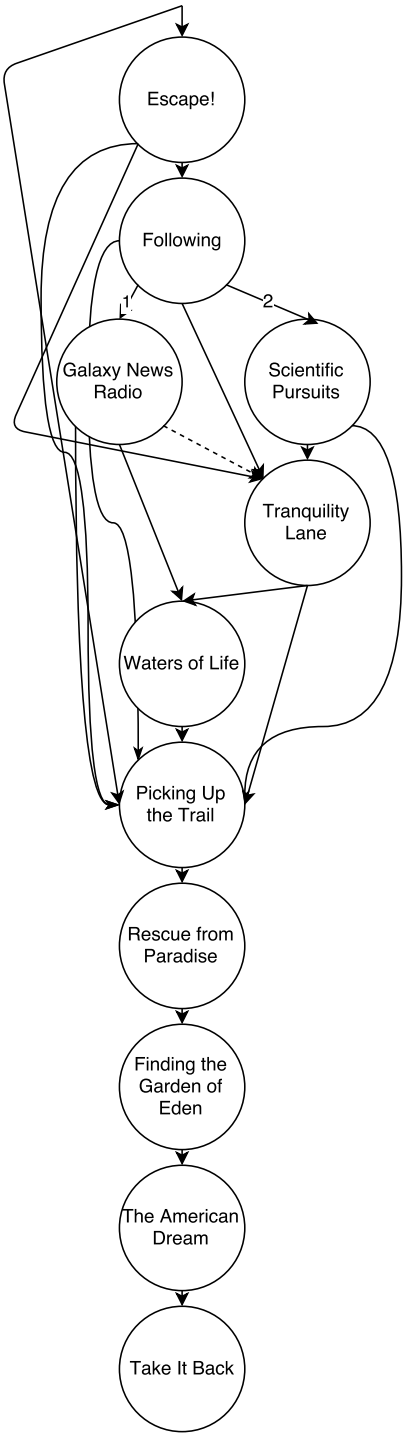

Figure 6.3: Fallout 3 Quests Assumming Common Glitches

the storyline becomes an almost point to point map since there is no way to stop a player from starting the quest (we have omitted this graph since it is hard to make heads or tails of it). As in the first two iterations, we apply the same routine and verify that using a gating function based on previous quests completing will prevent the player from skipping the storyline.

6.3 *Fallout 3* Quest Developer

This case study is an example of where the tool would be most useful, allowing a team of writers and developers or a single developer to create a quest for the game. The tool can be used at each step in the process to verify the quest will not cause a conflict or error within the quest itself or with any quest in the game.

6.3.1 Where Data was Obtained

The data used was obtained from the same sources as the previous two use cases. In this use case, we are also creating an entirely new quest that will integrate into the *Fallout 3* storyline.

6.3.2 SDL File Creation

In this case, we rely upon the SDL files from the previous two use cases to be the basis for the work in this use case. The creation of any new files will be based on the quest which is being created.

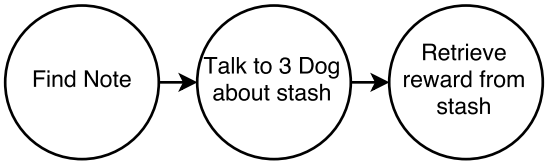

Figure 6.4: Intial Three Dog's Ultimate Stash Quest

6.3.3 Findings

This case starts with the idea for a quest. In this case, a simple side quest that requires finding Three Dog's ultimate stash is designed. It is a simple quest that has the player finding a note in Three Dog's stash, which is found during the Galaxy News Radio quest. This note directs the player to talk to Three Dog about his ultimate stash whereupon discussing it, Three Dog will give the player the location of the stash and the player must go retrieve its contents for a reward. Figure 6.4 shows the quest as a flowchart.

The next step is to convert this quest into a SDL file and verify that it does not cause problems using StoCk. It does have interactions with the quest Galaxy News Radio since it can start during this quest. Running the verification tool results in a failure stating: "Time Taken: 3040ms, File ../src/test/resources/json/fallout3/three-dog-ultimate-stash/tdus.json is invalid" meaning we cannot complete all plot points because our second plot point is talking to Three Dog to find out the location of his ultimate stash. This means we need to provide a route that if Three Dog is dead, the player can still find out about the ultimate stash. The new plot with this change can be seen in figure 6.5. Again, an SDL file is created and tested to prove that no errors

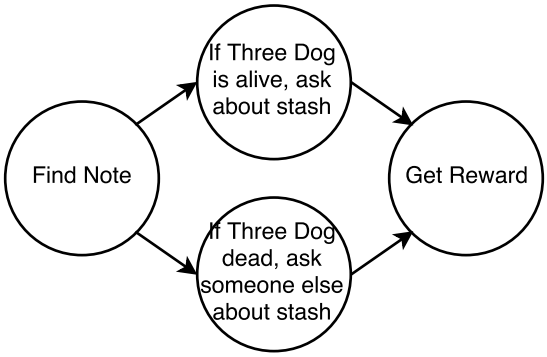

Figure 6.5: Second Attempt of Three Dog's Ultimate Stash Quest

can occur; on this run StoCk returns a passing message "Time Taken: 765ms, File: ../src/test/resources/json/fallout3/three-dog-ultimate-stash/tdus.json is valid", there are no errors meaning we are not breaking any quest that is currently made and that our quest is able to be completed by the player.

Now, the quest can be created using *Fallout 3*'s developer tool, G.E.C.K., that allows developers and users to create and modify elements of *Fallout 3*. The figures following show the various stages of the quest while playing it in *Fallout 3*. Figure 6.6 shows the player finding the note from Three Dog inside of the weapons cache inside of Hamilton's Hideaway. The next image (Figure 6.7) shows the game starting the quest and the next figure (Figure 6.8) is the contents of the note. The next figures: Figure 6.9, Figure 6.10, Figure 6.11, and Figure 6.12 show the conversations that are had with Three Dog if he is alive or with Doctor Li if Three Dog has been killed. The last few figures then show the completion of the quest (Figures 6.13 and 6.14) and the contents of the last note left by Three Dog in Figure 6.15.

Figure 6.6: Player locating note to start quest

Figure 6.7: Player beginning quest

Figure 6.8: Contents of note

Figure 6.9: Asking Three Dog about the weapons stash

Figure 6.10: Three Dog's Response to the question

Figure 6.11: Asking Doctor Li about the weapons stash

Figure 6.12: Doctor Li's Response to the question

Figure 6.13: The player about to pick up Three Dog's Ultimate Stash note

Figure 6.14: The player completing the quest

Figure 6.15: Contents of Three Dog's Ultimate Stash Log

This use case shows how using the plot verification tool can save time and effort through quick feedback loops on the ability to test and verify the quest's ability to be completed without harming the pre-existing storyline. Without this tool, after the author had spent time creating the quest, playing the quest, and deciding that it works correctly enough, they'd receive a complaint or bug report that the quest could not be completed if Three Dog was dead when they received the quest. This would lead to the developer having to replay the quest, verify the bug exists, change the quest, and test again. Even with a simplistic quest, this is a lot of additional work. If a quest existed that had more dependencies that the creator did not know about, this process could be repeated multiple times, each time the developer having to make sure the changes fix the problem and do not regress on the problems that had been fixed.

Chapter 7

Conclusion and Future Work

The recent developments in gaming has spawned larger complex stories and worlds involving multiple paths and choices to complete the story. However, these storylines are often so complex that game designers with their current tools could not verify all possible story lines. The application of formal verification to video game storylines is an unexplored research topic that can have an impact on the video game market by unburdening the designers, writers, and developers from constraining their designs based on in-game interactions becoming too difficult to test in a timely manner.

In this dissertation, we addressed the following research questions: how can complex storylines be modeled using formal verification methods, and what techniques are needed to map story lines to implement these methods into a usable and practical tool for game developers. Our work resulted in the following three tools:

1. SChar which provides a method to categorize the storyline model of a game base on simple questions

2. Our Storyline Description Language (SDL) which provides an implementation agnostic way for designers, writers, and developers to describe a

game storyline

3. StoCk which provides a tool to verify the correctness of a storyline

We then presented three case studies using StoCk to examine all of the storylines in *Fallout 3*. We examined the usefulness of our research from three different perspectives: the writers and designers, quality assurance, and developers. These case studies showed that a quick feedback loop can be achieved during all the use cases and integrated into an existing process. The work presented in this dissertation thus provides a solid base for storyline verification, and we look forward to seeing what can come from the work.

7.1 Future Work

Although inspired by many people in the industry and their input has been invaluable in shaping the work done. Our use cases were determined by speaking with them and the design and features of both the SDL and StoCk were also influenced by discussions with them. This groundwork provides a solid base but there are areas that must be addressed: user studies, user friendliness, and implementation refinement.

StoCk and the SDL has not yet been tested as a part of the game development process in the gaming industry. First, studies with companies in the gaming industry would provide valuable information on how to best integrate our tools. For instance, this could mean developing StoCk as a

plugin for Maven, Gradle, Visual Studio, or even a game development toolkit such as GECK.

Secondly, user friendliness was not a factor in our design, as it stands our toolset works for developers who are willing to put the time in to make it work within their process. To be even more useful, however, there are many tasks to undertake. should provide better correlation between the model checker output and the storyline description language input. For instance, the ability to present the results of the model checker in a visual tool or report that pin-points the exact problem would be very helpful. Additionally, the ability to generate the quests visually or from an implementation or other tooling used by developers could increase the interest and usage. Lastly is the refinement of the implementation.

Lastly, in this dissertation we focused on one game from both the designer or writer perspective and from the developer perspective. Additional testing with other real world games can provide insight that could drive the adoption of another model checker or formal verification method. Another aspect to test is other model checking programs to find the best match for the goal. This could mean a change from SPIN to another tool or, even, a model checker written specifically for our game testing use case.

Appendices

Appendix A

Exploratory Survey

This contains the survey and tabulated results from the running of the survey.

1. Preliminaries

   (a) What sex are you?

Male	Female
532	68

   (b) How old are you?

18-20	21-30	31-40	41-50	51-60	60 and older
202	340	46	9	3	2

   (c) What is the highest level of education you have attained?

High school	some college	bachelors	masters	phd
39	282	157	102	23

2. Now, lets talk about video games

   (a) Do you play video games? Which type of video games?

Type	Number
Action	483
Adventure	424
Puzzle	309
Role Playing	419
Simulation	199
Sports	197
Strategy	406
I don't play games	20

(b) How important is story to you within a game?

0	1	2	3	4	5
18	15	43	96	242	181

(c) Would you play a game that has a dynamic storyline?

Yes	No
575	24

(d) Would you prefer a static or dynamic game?

Static	Dynamic
91	506

(e) Given your answer from above, why did you choose static or dynamic?

Selected Answers
It simulates more of a realistic realm.
It allows the storyline to be played multiple times.
More realism, more immersion.
I feel like it's easy to miss content in games with dynamic stories.

Appendix B

History of Storytelling

The background work presented in the dissertation is directly related to the work presented within the document. However, there is more work that also relates to the history of video games and academic dramatic storytelling.

B.1 Live Action Storytelling

Live action storytelling can be used to convey information, tell an entertaining anecdote, pass along tradition, or entertain an audience or participants. It can be as simple as telling a story to a single child or a group; it can involve a cast of many, each fulfilling a different role with pre-determined motivations. Storytelling is diverse and has many different aspects, but four main styles of storytelling stand out: traditional storytelling, dinner mysteries and improvisational theater, tabletop role-playing, and live-action role-playing.

B.1.1 Traditional Storytelling

Traditional storytelling, according to the National Storytelling Network, is four things: interactive, uses words and gestures, presents a story and encourages the imagination of its listeners [94]. Storytelling is interactive

because it involves two-way interaction between the teller and the listeners. It can tightly connect the storyteller and the audience and help with the impact of the story. Using words and gestures differentiates storytelling from other forms of interaction such as dance, miming, or reading text in a computer game. Storytelling always presents a narrative; this again differentiates it from stand-up comedy or poetry readings in which narratives are not necessary to understand the underlying messages. Finally, it encourages the imagination of its listeners. The narrative and some details may be supplied by the storyteller, but the world in which the story is taking place is fully fleshed out by the listeners.

B.1.2 Improvisational Theater

Improvisational Theater is another form of storytelling which can encourage user feedback to drive the action. In its most pure form it is a method in which the actors play a dramatic scene with minimal or no predetermined activity [95]. In another form improvisational theater is often comedic and takes cues from the audience to create impromptu scenes. This form of improvisational theater can be seen on shows such as "Whose Line is it Anyway?" or in person at various theaters.

B.1.3 Tabletop Role-Playing

Tabletop role-playing games involve a group of people with one person acting as the game master and the others playing characters within the game.

The game master defines the world, its inhabitants, outcomes of player actions with the inhabitants and guides (hopefully discreetly) the players through the narrative. The players are allowed to improvise within the world and with their actions inside of the world, helping to shape the full narrative of the story. An example of a tabletop role-playing game would be Dungeons & Dragons, which is still the most dominant table-top role playing game on the market [96].

B.1.4 Live-Action Role-Playing

Live-action role-playing (LARPing) can be thought of as a combination of improvisational theater and tabletop role-playing. Players in a live-action role-playing setting act out their character's actions physically and undertake their character's goals in a fictional setting in the real world while interacting with one another [97]. As with tabletop role-playing, LARPing requires a game master (or game masters depending on the size of the group) to define rules and settle disputes between players during play. The players in live-action role-playing are much like the actors in an improvisational play since they must act out goals given to them by an audience (the GM) while remaining in character.

B.2 Computational Story Creation

Computational story creation, although not interactive, can be considered the impetus for much of the research today into interactive storytelling. TALESPIN, MINSTREL and BRUTUS are classic examples of this research.

All computational story creation systems create a story based upon a set of initial conditions and logic statements and attempt to solve the logic statements in order to create a story.

B.2.1 TALESPIN

TALESPIN [98] attempted to derive stories based upon the goals of simulated characters. The content of the story was represented as character goals and operators to achieve the goals. The storylines in TALESPIN were driven entirely by the characters' motivations. This neglected any storylines that involved cooperation or portrayals of characters for dramatic effect[1].

B.2.2 MINSTREL

Turner's MINSTREL [99] is a step forward from TALESPIN in that it augmented TALESPIN's story planning with meta-level goals and plans that represent what the author is trying to achieve. It views authoring a story as a problem solving exercise and uses a case-based reasoner to write the stories. The process used by MINSTREL is: first, identify a problem to solve; secondly, recall a past solution similar to the current problem; next, adapt the past solution; finally, apply the adapted solution to the current problem.

[1] `http://grandtextauto.org/2007/10/30/scott-turner-on-minstrel/`

B.2.3 BRUTUS

BRUTUS [100] generates stories based upon rigorous logical definitions of betrayal and heartbreak. BRUTUS' goal is to create stories that are sufficiently distant from an initial knowledge representation of the logical definitions and vary independently across dimensions such as characters, setting, and themes.

B.3 From Computation Story Creation to Interactive Drama

While much academic research has been done with computational storytelling, it was not until the mid-1980s that academic research started into interactive storytelling.

B.3.1 Interactive Storytelling System

Laurel's work [76] was the design of an interactive storytelling system. Her work defined a playwright, now called a drama manager, along with a list of thirteen functions required to make the system complete. Laurel's work offered no ideas as to how these functions would be implemented, and, in Crawford's [13] words, "it's a wish list and not a plan" however, it set the stage for all academic interactive fiction research.

Laurel's work is the genesis for the current academic research such as search-based drama management [77], Facade [80], OPIATE [82], and PaSSAGE [84] among many others.

Appendix C

Video Game Ontology

Video games have a few characteristics upon which they are normally described. This section describes the common attributes used to describe most games.

C.1 Gameplay Points-of-View

We need to categorize the point-of-view of the player. There are three main points-of-view: text-based, third person and first person.

Text-based points-of-view were very popular before computers and video game systems had the power to display graphics. The *Zork* series of games is considered to be one of the greatest adventure games of all time, and it is text-based. As a text-based game, the world, characters, and objects are described as blocks of text on the screen and the player provides input via the keyboard.

Third-person is a superset of different perspectives. These perspectives are: top-down, side, tile-based isometric, and 3D isometric. The top-down perspective is used in shooter-style games with the player's character centered

on the screen. Some examples of the top-down view are: *The Legend of Zelda*[1], *Asteriods*[2], and *Galaga*[3]. Side, or profile, view displays the player's character from a profile perspective and is often used in beat 'em up and early action games. *Metroid*[4], *Streets of Rage*[5], *Super Mario Brothers*[6] and *Mega Man*[7] are all examples of games that use the side perspective. Finally, tile-based and 3D isometric views often display the player's character or characters from a 3/4 view. In tile-based isometric views, the camera never changes its position on the world since the world is comprised of sprites (images) and does not use polygons to create the world. In 3D isometric views, the player has the ability to spin, zoom in or out, and position the camera however they choose since the world is made of polygons and is a true 3D representation. *Civilization IV*[8], *Ultima 6*[9] and *Crusader: No Remorse*[10] are examples of tile-based isometric viewpoints. *Dead Space*[11] and *Mass Effect*[12] are examples of 3D isometric views.

The first-person viewpoint displays the world as it is seen through the

[1]Nintendo, *The Legend of Zelda*, Console, 1987.
[2]Atari, *Asteroids*, Arcade, 1979.
[3]Namco, *Galaga*, Arcade, 1981.
[4]Nintendo, *Metroid*, Console, 1986.
[5]SEGA, *Streets of Rage*, Console, 1991.
[6]Nintendo Entertainment, *Super Mario Brothers*, Console, 1985.
[7]Capcom, *Mega Man*, Console, 1987.
[8]*Civilization 4*, Firaxis Games, 2005.
[9]Origin Systems, *Ultima VI: The False Prophet*, CD-ROM, 1990.
[10]Origin Systems, *Crusader: No Remorse*, CD-ROM, 1995.
[11]Visceral Games, *Dead Space*, Console, 2008.
[12]BioWare, *Mass Effect*, Console, 2007.

eyes of the character the player is controlling. *Wolfenstein 3D*[13], *Doom*, *Half-Life*[14], *Quake* and *Unreal Tournament*[15] are all examples of the first-person viewpoint.

C.2 Number of Players

Another aspect of all games is the number of players allowed to play the game at one time. The four categories are: single player, two player, multiplayer, and massively multiplayer (MM). Single player games are designed to allow only one player to play the game. *Donkey Kong*[16], *Metroid*, and *Dead Space* are all examples of single player games. Two player games are designed for two players - usually at the same time although there are some exceptions such as *Super Mario Brothers*, where the players alternate playing the game. Beat 'em ups like *Streets of Rage* and fighting games such as *Street Fighter II*[17] are quintessential examples of two player games where the players play at the same time. Multiplayer games are games where two to 64 or more (depending on the game) players can play at the same time. The multiplayer option in the game *Halo*[18] is a great example of this - it pits multiple teams against one another. Massively multiplayer games allow thousands of players in a single persistent virtual world. They are different than multiplayer games

[13]id Software, *Wolfenstein 3D*, Disk, 1992.
[14]Valve Software, *Half-Life*, CD-ROM, 1998.
[15]Epic Games, *Unreal Tournament*, CD-ROM, 1999.
[16]Nintendo, *Donkey Kong*, Arcade, 1981.
[17]Capcom, *Street Fighter II*, Arcade, 1991.
[18]Bungie, *Halo*, Console, 2001.

due to the massive number of players that can play at the same time. Another distinguishing feature of massively multiplayer games is the persistent world - even as players log in and out of the game, the world continues without them. This is in stark contrast to other multiplayer games where, when the players leave, the world is destroyed. The most popular massively multiplayer game is *World of Warcraft*[19].

C.3 Game Types

Video and computer games can be divided by type into a few major categories: action, adventure, puzzle, role playing, simulation, sports, strategy, and hybrids. Below we will look at each of the types briefly and give examples of each type.

C.3.1 Action Games

Action games are games that challenge a player's speed, dexterity and reaction times when responding to on-screen stimulus. These place a premium on exciting actions such as shooting, fighting, and dodging. Under the umbrella of action games are the many sub-genres such as fighting, beat 'em ups, platformers, and shooters. Fighting games normally pit the user in on-screen hand-to-hand combat against one or more enemies at a time within an arena. Some popular fighting games are: *Street Fighter II*, *Soul Calibur IV*[20], and

[19]Blizzard, *Wold of Warcraft*, CD-ROM, 2004.
[20]Namco, *Soul Calibur 4*, Console, 2008.

Dead or Alive 4[21]. Beat 'em ups focus again on hand-to-hand fighting, but in this case, it is one (or two) versus many across multiple levels. The players fight through hordes of enemies to arrive at the end of the level where a fight with a much harder enemy (a boss) occurs. When the players defeat the boss they are allowed to move to the next level. Examples of beat 'em up games include *Streets of Rage* and *Double Dragon*. Platformers are similar to beat 'em ups but place more emphasis on gymnastic feats such as jumping. *Super Mario Brothers*, *Metroid*, and *Mega Man* are examples of platformer games. Finally, there are shooters; these games are like beat 'em ups but instead of hand-to-hand combat you are shooting enemies with various weapons. *Doom*, *R-Type*, and *Half-Life* are all shooters.

C.3.2 Adventure Games

Adventure games focus on exploration, puzzle solving, and interaction with characters inside the story world to solve the puzzles. The player navigates through the story and puzzles are used to continue the story. Popular adventure games include the *Zork* series, *King's Quest*[22], *Quest for Glory*[23], *Under a Killing Moon*[24], and *Sam & Max Hit the Road*[25]. The adventure game as described today is no longer the juggernaut it was in the '90s; it is now a niche genre with action-adventure games taking their place. The

[21] Tecmo, *Dead or Alive 4*, Console, 2005.
[22] Sierra On-line, *King's Quest*, Disk, 1984.
[23] Sierra On-line, *Quest for Glory*, Disk, 1989.
[24] Access Software, *Under a Killing Moon*, CD-ROM, 1994.
[25] LucasArts, *Sam and Max Hit the Road*, CD-ROM, 1993.

action-adventure games will be discussed in the hybrids section.

C.3.3 Puzzle Games

Puzzle games emphasize the player solving a puzzle. For instance, *Tetris*[26] requires the player to create solid horizontal lines out of falling pieces that are different shapes, and the goal is to continue making these lines and clearing the board for as long as possible. Other game types like action, adventure, and role playing might have puzzle elements inside of them but puzzle solving is not the goal of the game. Some other popular puzzle games are: *Columns*[27], *Lumines*[28], *Dr. Mario*[29], and *The Incredible Machine*[30].

C.3.4 Role Playing Games

Role playing games share much in common with the tabletop role playing games like Dungeons and Dragons. The player takes control of a single person or group and controls them in a story. An emphasis is placed on the statistical improvement of the characters while playing the game. However, computer and console role playing games do not have a human guiding the story. The story's plot is most often linear or has very few branches so the player only guides his group through quests and interactions defined by the game. The battles are done in a turn-based style with the user allowed to pick

[26]Nintendo, *Tetris*, Console, 1989.
[27]SEGA, *Columns*, Console, 1990.
[28]Bandai, *Lumines*, Console, 2004.
[29]Nintendo, *Dr. mario*, Console, 1990.
[30]Dynamix, *The Incredible Machine*, Disk, 1993.

an action per character per turn. Examples of this style of game are *Final Fantasy*, *Neverwinter Nights*[31], and *Lost Odyssey*.

C.3.5 Simulation Games

Simulation games try to model some aspect of reality so that the user can interact within this programmed reality. An example of a simulation game would be *Sim City*. In *Sim City*, the program simulates the workings of a city and its population, and the user can decide to strive to head a successful city or become a slumlord. There are multiple other simulation subtypes; some of the most popular are: vehicle simulations like *Gran Turismo*[32] and *Forza Motorsports*[33], life simulations such as *The Sims* and *Nintendogs*[34], and management simulations such as *Sim City* and *NFL Head Coach*[35].

C.3.6 Sports Games

Sports games attempt to simulate playing a typical real-world sport such as football, basketball, or soccer. These games would not be considered simulation games since they focus on playing the sport as opposed to handling the management aspects of the sport. However, since many of the sports games are beginning to merge many of the back office dealings into the games, this would not change the classification of the game since the point of the game is

[31]BioWare, *Neverwinter Nights*, CD-ROM, 2002.
[32]Sony Computer Entertainment, *Gran Turismo*, Console, 1998.
[33]Microsoft Game Studios, *Forza Motorsport*, Console, 2005.
[34]Nintendo, *Nintendogs*, Console, 2005.
[35]Electronic Arts, *NFL Head Coach*, Console, 2006.

to play the sport against another opponent.

C.3.7 Strategy

Strategy games are characterized by the need for the user to think and plan in order to achieve victory. The game itself can be either turn-based or real-time. In turn-based strategy, the player and all opponents take turns moving and manipulating their forces, whereas in real-time strategy all moves and manipulations are done by all players at the same time. Some examples of strategy games are *Civilization*, *StarCraft*[36], *Galactic Civilizations*[37], and *X-COM*[38].

C.3.8 Hybrids

Many games have now begun to exhibit qualities found in multiple game types. For instance, action-adventure games have over adventure games in terms of popularity. Action-adventure games often combine the puzzles found in adventure games with the speed and dexterity needed to defeat enemies within the game. Some popular action-adventure games are *Resident Evil*[39], *Metroid*, and *Castlevania*[40]. Another hybrid genre is the strategy-role playing game. In this style of game, the normal battle sequence is replaced with a tiled field populated with enemies and the player's group where the player

[36]Blizzard, *StarCraft*, CD-ROM, 1998.
[37]StarDock, *Galactic Civilizations*, CD-ROM, 2003.
[38]MicroProse, *X-COM: UFO Defense*, Disk, 1993.
[39]Capcom, *Resident Evil*, Console, 1996.
[40]Konami, *Castlevania*, Console, 1986.

must command their troops to victory versus the enemies. The other aspects of the role-playing game are kept intact such as the emphasis on statistically increasing the player's party. A hybrid game is one that exhibits traits found within multiple game types.

C.4 Examples of Number-of-players, Game types and plots

Now that number-of-players, game types and plot representations have been examined we can explore how they interact with one another in current games. The three factors (number-of-players, game type and plot type) usually interact with one another and affect the type of game that is being created. The primary player-type-plot combinations seen today are: single and multi player action linear plot, single player RPG linear plot, single player Hybrid action-RPG sandbox plot and massively multiplayer RPG branching and foldback plot.

C.4.1 Single and multi-player action linear plot

The single player action game with a linear plot is a fairly standard choice for many games. Classic arcade games such as *Donkey Kong* and *Pac-Man* are of this type. Many side scrolling shooter games such as *R-Type* and *Gradius* are as well. In these games, the plot is an excuse to transport the player from world to world, conquering the enemies that appear on the screen. The interactions with the world are normally minimal, such as opening a door,

pressing a switch, or collecting an item from a box.

C.4.2 Single player RPG linear plot

RPGs such as *Final Fantasy*, *Lost Odyssey*, and *Chrono Trigger*[41] exemplify single player RPG games. The player is given control of a group of characters who are the main characters within a highly constrained narrative. The stories often begin with a simple goal that soon expands into a story about saving the world or a kingdom by defeating another character and his powerful minions. The player is given no choices on how the story is going to be completed; they merely level up their characters in order to fight the next boss character and finally, defeat the last boss and see the ending. Interactions in this type of RPG expand from what is allowed in the typical action game to include simple dialog with NPCs and the ability to buy and sell items at a set price from defined NPCs.

C.4.3 Single player Hybrid (RPG/Action) sandbox plot

The *Fallout* series and *The Elder Scrolls* games are prime examples of the single player hybrid sandbox plot game. In both of these games, the player is given a straightforward first quest to get them familiar with the game and establish the backstory for the rest of the game. After the first quest is completed, the player is then allowed to enter the larger world and continue with the game as they see fit. As quests are completed, other quests are

[41]Square, *Chrono Trigger*, Console, 1995.

revealed that can be completely unrelated to the main storyline. Interactions within this style of game are similar to the single player RPG linear plot but, sometimes, speaking with NPCs can uncover additional quests unrelated to the main quest that the player can participate in.

C.4.4 Massively Multiplayer RPG branching and foldback plot

The massively multiplayer RPG branching foldback plot style games are often called Massively Multiplayer Online RPGs (MMORPGs) and allow a player to interact with many other players in the same virtual world as themselves. The most popular game of this type is World of Warcraft, in which the player is allowed to create a character who is then placed into the world in a starting town and given simple tasks to become acclimated with the world. As the quests are completed, more zones and quests are opened to the player. The player can, at any time, party with other players and complete quests together. The interactions in an MMORPG are the same as the single player RPG linear plot, but the human controlled players can talk and sell items to one another in addition to speaking with the NPCs that inhabit the world.

Bibliography

[1] M. S. O. Almeida and F. S. C. da Silva, "A systematic review of game design methods and tools," in *Entertainment Computing–ICEC 2013*, Springer, 2013, pp. 17–29.

[2] S. Kriglstein, R. Brown, and G. Wallner, "Workflow patterns as a means to model task succession in games: a preliminary case study," in *Entertainment Computing–ICEC 2014*, Springer, 2014, pp. 36–41.

[3] K. Neil, D. de Vries, and S. Natkin, "A tool for evaluating, adapting and extending game progression planning for diverse game genres," in *Entertainment Computing–ICEC 2014*, Springer, 2014, pp. 60–65.

[4] J. Lebowitz and C. Klug, *Interactive Storytelling for Video Games: A Player-centered Approach to Creating Memorable Characters and Stories*. Focal Press, 2011, ISBN: 9780240817170. [Online]. Available: `https://books.google.com/books?id=QUrarEcvaO8C`.

[5] T. Kropf, *Introduction to Formal Hardware Verification*. Springer Berlin Heidelberg, 2010, ISBN: 9783642084775. [Online]. Available: `https://books.google.com/books?id=usAHkgAACAAJ`.

[6] C. Baier and J.-P. Katoen, *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008, ISBN: 026202649X, 9780262026499.

[7] M. Jaring and J. Bosch, "Representing variability in software product lines: a case study," in *Software Product Lines*, Springer, 2002, pp. 15–36.

[8] S. Kent, *The Ultimate History of Video Games: from Pong to Pokemon and beyond... the story behind the craze that touched our li ves and changed the world*. Three Rivers Press, 2010.

[9] V Sarinho and A Apolinário, "A feature model proposal for computer games design," in *VII Brazilian Symposium on COmputer games and Digital entertainment, Belo horizonte*, 2008, pp. 54–63.

[10] J. Blow, "Game development: harder than you think," *Queue*, vol. 1, no. 10, pp. 28–37, Feb. 2004, ISSN: 1542-7730. DOI: `10.1145/971564.971590`. [Online]. Available: `http://doi.acm.org.ezproxy.lib.utexas.edu/10.1145/971564.971590`.

[13] C. Crawford, *Chris Crawford on Interactive Storytelling*, ser. New Riders Games. Pearson Education, 2005, ISBN: 9780132582254. [Online]. Available: https://books.google.com/books?id=_SnhNhcNGr4C.

[14] L. Beyak and J. Carette, "Saga: a dsl for story management," *arXiv preprint arXiv:1109.0776*, 2011.

[15] I. G. D. Association, *Scriptwriting for games: part 1: foundations for interactive storytelling*. [Online]. Available: http://aii.lgrace.com/documents/IDGA_Foundations_of_Interactive_Storytelling.pdf.

[16] ——, *Scriptwriting for games: part 2: advanced plot story structures*.

[39] G. Sutcliffe and C. Suttner, "Evaluating general purpose automated theorem proving systems," *Artificial intelligence*, vol. 131, no. 1, pp. 39–54, 2001.

[40] J. A. Robinson, "A machine-oriented logic based on the resolution principle," *Journal of the ACM (JACM)*, vol. 12, no. 1, pp. 23–41, 1965.

[41] M. Sipser, *Introduction to the Theory of Computation*. Cengage Learning, 2012.

[42] V. Vychodil, "On generating of proofs," in *SCIS & ISIS*, Japan Society for Fuzzy Theory and Intelligent Informatics, vol. 2006, 2006, pp. 1071–1078.

[43] D. W. Loveland, "Mechanical theorem-proving by model elimination," *Journal of the ACM (JACM)*, vol. 15, no. 2, pp. 236–251, 1968.

[44] D. W. Loveland, "A simplified format for the model elimination theorem-proving procedure," *J. ACM*, vol. 16, no. 3, pp. 349–363, Jul. 1969, ISSN: 0004-5411. DOI: 10.1145/321526.321527. [Online]. Available: http://doi.acm.org/10.1145/321526.321527.

[45] R. Kowalski and D. Kuehner, "Linear resolution with selection function," *Artificial Intelligence*, vol. 2, no. 3, pp. 227–260, 1972.

[46] J. Otten, "Leancop 2.0 and ileancop 1.2: high performance lean theorem proving in classical and intuitionistic logic (system descriptions)," in *Proceedings of the 4th international joint conference on Automated Reasoning*, Springer-Verlag, 2008, pp. 283–291.

[47] E. W. Beth, "Semantic entailment and formal derivability," in *Mededelingen van de Koninklijke Nederlandse Akademie van Wetenschappen*, 1955, pp. 309–342.

[48] R. M. Smullyan, *First-order logic*. Courier Corporation, 1995.

[49] S. Schulz, "System Description: E 1.8," in *Proc. of the 19th LPAR, Stellenbosch*, K. McMillan, A. Middeldorp, and A. Voronkov, Eds., ser. LNCS, vol. 8312, Springer, 2013.

[50] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski, "Spass version 3.5," in *Automated Deduction–CADE-22*, Springer, 2009, pp. 140–145.

[51] L. Kovács and A. Voronkov, "First-order theorem proving and vampire," in *Computer Aided Verification*, Springer, 2013, pp. 1–35.

[52] T. Hillenbrand, "Superposition and decision procedures – back and forth," PhD thesis, Universität des Saarlandes, 2008.

[53] D Knuth and P Bendix, *Simple word problems in universal algebra. computational problems in abstract algebra, conference held at oxford in 1967*, 1970.

[54] L. Bachmair and H. Ganzinger, "Rewrite-based equational theorem proving with selection and simplification," *Journal of Logic and Computation*, vol. 4, no. 3, pp. 217–247, 1994.

[55] R. Nieuwenhuis and A. Rubio, "Paramodulation-based theorem proving.," *Handbook of automated reasoning*, vol. 1, pp. 371–443, 2001.

[56] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Communications of the ACM*, vol. 5, no. 7, pp. 394–397, 1962.

[57] M. Davis and H. Putnam, "A computing procedure for quantification theory," *Journal of the ACM (JACM)*, vol. 7, no. 3, pp. 201–215, 1960.

[58] A. Biere, M. Heule, and H. van Maaren, *Handbook of satisfiability*. ios press, 2009, vol. 185.

[59] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient sat solver," in *Proceedings of the 38th annual Design Automation Conference*, ACM, 2001, pp. 530–535.

[60] N. Eén and N. Sörensson, "An extensible sat-solver," in *Theory and applications of satisfiability testing*, Springer, 2004, pp. 502–518.

[61] K. Pipatsrisawat and A. Darwiche, "Rsat 2.0: sat solver description," Automated Reasoning Group, Computer Science Department, UCLA, Tech. Rep. D–153, 2007.

[62] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching-time temporal logic," in *Logic of Programs, Workshop*, London, UK, UK: Springer-Verlag, 1982, pp. 52–

71, ISBN: 3-540-11212-X. [Online]. Available: http://dl.acm.org/citation.cfm?id=648063.747438.

[63] E. A. Emerson and E. M. Clarke, "Characterizing correctness properties of parallel programs using fixpoints," in *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, London, UK, UK: Springer-Verlag, 1980, pp. 169–181, ISBN: 3-540-10003-2. [Online]. Available: http://dl.acm.org/citation.cfm?id=646234.682526.

[64] J.-P. Queille and J. Sifakis, "Specification and verification of concurrent systems in cesar," in *International Symposium on Programming*, Springer, 1982, pp. 337–351.

[65] R. Jhala and R. Majumdar, "Software model checking," *ACM Computing Surveys (CSUR)*, vol. 41, no. 4, p. 21, 2009.

[66] M. Ouimet and K. Lundqvist, "Formal software verification: model checking and theorem proving," *Embedded Systems Laboratory, MIT*, 2007.

[67] G. J. Holzmann, "An analysis of bitstate hashing," *Formal methods in system design*, vol. 13, no. 3, pp. 289–307, 1998.

[68] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *Computers, IEEE Transactions on*, vol. 100, no. 8, pp. 677–691, 1986.

[69] ——, "Symbolic boolean manipulation with ordered binary-decision diagrams," *ACM Computing Surveys (CSUR)*, vol. 24, no. 3, pp. 293–318, 1992.

[70] C Courcoubetis, M Vardi, P Wolper, and M Yannakakis, "Memory-efficient algorithms for the verification of temporal properties," *Formal Methods in System Design*, vol. 1, no. 2-3, pp. 275–288, 1992.

[71] J. Staunstrup, H. R. Andersen, H. Hulgaard, J. Lind-Nielsen, K. G. Larsen, G. Behrmann, K. Kristoffersen, A. Skou, H. Leerberg, and N. B. Theilgaard, "Practical verification of embedded software," *Computer*, no. 5, pp. 68–75, 2000.

[72] J. Lind-Nielsen, H. R. Andersen, G. Behrmann, H. Hulgaard, K. Kristoifersen, and K. G. Larsen, "Verification of large state/event systems using compositionality and dependency analysis," in *Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 1998, pp. 201–216.

[73] G. J. Holzmann and D. Peled, "An improvement in formal verification.," in *FORTE*, vol. 6, 1994, pp. 197–211.

[74] A. Valmari, "A stubborn attack on state explosion," in *Computer-Aided Verification*, Springer, 1991, pp. 156–165.

[75] D. Peled, "Combining partial order reductions with on-the-fly model-checking," in *Computer aided verification*, Springer, 1994, pp. 377–390.

[76] B. K. Laurel, "Toward the design of a computer based interactive fantasy system," PhD thesis, Ohio State University, 1986. [Online]. Available: `https://etd.ohiolink.edu/rws\textunderscoreetd/document/get/osu1240408469/inline`.

[77] P. Weyhrauch, "Guiding intreactive drama," PhD thesis, Carnegie Mellon University (CMU), 1997. [Online]. Available: `gel.msu.edu/classes/TC848/papers/Weyhrauch.GuidingInteractiveDrama.pdf`.

[78] M. J. Nelson and M. Mateas, "Search-based drama management in the interactive fiction anchorhead.," in *AIIDE*, 2005, pp. 99–104.

[79] M. Sharma, S. Ontanón, C. R. Strong, M. Mehta, and A. Ram, "Towards player preference modeling for drama management in interactive stories.," in *FLAIRS Conference*, 2007, pp. 571–576.

[80] M. Mateas, "Interactive drama, art and artificial intelligence," 2002.

[81] B. S. Magerko, "Player modeling in the interactive drama architecture," PhD thesis, University of Michigan, 2006.

[82] C. R. Fairclough, "Story games and the opiate system: using case-based planning for structuring plots with an expert story director agent and enacting them in a socially simulated game world," PhD thesis, University of Dublin, Trinity College, 2004. [Online]. Available: `https://www.cs.tcd.ie/publications/tech-reports/reports.05/TCD-CS-2005-59.pdf`.

[83] V. Propp, *Morphology of the Folktale*. University of Texas Press, 2010, vol. 9.

[84] D. Thue, "Player-informed interactive storytelling," Master's thesis, University of Alberta, 2007. [Online]. Available: `https://sites.google.com/a/ualberta.ca/ircl/projects/passage/thue07-thesis.pdf`.

[85] C. A. Petri, "Kommunikation mit automaten," 1962.

[86] T. Murata, "Petri nets: properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[87] J. P. Morrison, D. A. Power, and J. J. Kennedy, "A condensed graphs engine to drive metacomputing.," in *PDPTA*, 1999, pp. 902–908.

[88] S. Pakdel and A. C. Elster, "Enhancing the performance of condensed graph's computation in distributed systems by using numerical libraries as super nodes," in *Proceeding of the International Conference on Computer Science, Computer Engineering, and Social Media (CSCESM 2014)*, London, UK, UK: SDIWC Publications, 2014, pp. 86–93, ISBN: ISBN: 978-1-941968-04-8.

[89] D. Harel, "Statecharts: a visual formalism for complex systems," *Science of computer programming*, vol. 8, no. 3, pp. 231–274, 1987.

[90] O. OMG, "Unified modeling language (omg uml)," *Superstructure*, 2013.

[91] G. Holzmann, *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2011, ISBN: 9780321773715. [Online]. Available: https://books.google.com/books?id=1F1HYgEACAAJ.

[92] F. Press, *Fallout 3*, ser. The official strategy guide. Future Press Verlag und Marketing GmbH, 2008, ISBN: 9783940643223. [Online]. Available: https://books.google.com/books?id=carlOgAACAAJ.

[93] G. J. Holzmann, R. Joshi, and A. Groce, "Swarm verification techniques," *Software Engineering, IEEE Transactions on*, vol. 37, no. 6, pp. 845–857, 2011.

[94] E. Ellis, *From Plot to Narrative: A Step-by-step Process of Story Creation and Enhancement*. Parkhurst Brothers Publishers Incorporated, 2012, ISBN: 9781935166818. [Online]. Available: https://books.google.com/books?id=8bmepwAACAAJ.

[95] V. Spolin, *Improvisation for the Theater: A Handbook of Teaching and Directing Techniques*, ser. Drama and Performance Studies. Northwestern University Press, 1999, ISBN: 9780810140080. [Online]. Available: https://books.google.com/books?id=W24B26mGvQkC.

[96] W. R. Team, *Player's Handbook*, ser. D&D Core Rulebook Series. Wizards of the Coast, 2014, ISBN: 9780786965601. [Online]. Available: https://books.google.com/books?id=ZjzjoAEACAAJ.

[97] D. Mackay, *The Fantasy Role-Playing Game: A New Performing Art*. McFarland & Company, 2001, ISBN: 9780786450473. [Online]. Available: https://books.google.com/books?id=s8YRVbDknyUC.

[98] J. Meehan, "The metanovel: writing stories by computer," PhD thesis, Yale University, 1976.

[99] S. R. Turner, "Minstrel: a computer model of creativity and storytelling," 1993.

[100] S. Bringsjord and D. Ferrucci, *Artificial intelligence and literary creativity: Inside the mind of brutus, a storytelling machine*. Psychology Press, 1999.

Index

Vita

Lane Thomas Holloway was born July 21, 1978 to James Lovis Holloway and Patricia Wilkinson Holloway. He grew up loving computers, video games, and basketball. He received a B.S.E.E. from The University of Texas at Austin in December of 2001 and a M.S.E.E. from The University of Texas at San Antonio in September 2003. He has worked as a software developer and software architect for IBM, Sotera Defense Solutions, and HomeAway. He is currently Principal Portfolio Architect in the Office of the CTO at Rapid7 researching topics related to computer security and vulnerabilities, and scalable architectures.

Permanent address: 1101 Brown Drive
Pflugerville, Texas 78660

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.