ABSTRACT

ROBERTSON, JOSEPH JUSTUS. Perceptual Experience Management. (Under the direction of R. Michael Young and David Roberts.)

In strong story experience management there is a tension between *player autonomy* and *authorial control*. Autonomy is a player's ability to choose from a full range of actions afforded by the environment and make meaningful change to the story world. Control is an author's ability to impose specific narrative structures in the interactive experience. In automated experience managers, authors impose narrative structures by specifying constraints on the sequences of story events and world states. In these systems, the tension between player autonomy and authorial control arises when a player's action prevents an authorial constraint from holding. When a player acts on an opportunity to break an authorial constraint, there are two options available to an interactive narrative system that preserve both player autonomy and authorial control. The first, called *intervention*, modifies the outcome of the player's action so it no longer reverses the authorial constraint. The second method, called *accommodation*, selects a new story that incorporates the player's action and still conforms to the author's constraints. However, neither of these alternatives are ideal. If the system alters the behavior of a player's action, the player may notice an inconsistency between different similar contexts in which the action is performed. Alternatively, a second story that both incorporates the player's action and adheres to the author's constraints does not always exist.

This document presents an automated strong story experience manager that mitigates drawbacks of both approaches by utilizing a model of player knowledge. First, the experience manager ensures interventions are consistent across game contexts by constraining the space of interventions to exclude those that contradict what the player has previously observed. The experience manager also widens the space of possible accommodations to include alternate possible world histories that are consistent with past player observations. Together, these two components shift the story world between possible worlds consistent with player observations to better maintain author constraints. This process is called *perceptual experience management*. The framework is implemented in a novel hybrid experience manager and game engine called the General Mediation Engine (GME). GME is specially suited for the framework because it merges the declarative methodology of automated strong story experience managers with procedural commercial game engines by utilizing a content generation pipeline to dynamically assemble and revise game content based on the experience manager's underlying symbolic state. Because GME's state representation and update mechanics are embedded in the experience manager's representation, the game world the player interacts with dynamically reconfigures whenever the experience management framework modifies the underlying world state or action transitions.

Finally, this document presents a series of structural and human subject evaluations of the experience manager. It is hypothesized that the experience manager will expand a larger number of branches consistent with author constraints while avoiding branches where players notice modifications of the story world state and mechanics. First, GME is used to evaluate whether the experience manager expands more branches consistent with author constraints by comparing interactive narrative trees produced by perceptual experience management against trees produced by a baseline method. Next, a series of three human subjects experiments are presented that show players notice inconsistencies produced by perceptual experience management when the modifications are not constrained by a model of player knowledge. Together, these evaluations provide evidence that perceptual experience management better mediates the tension between player autonomy and authorial control by expanding the number of branches in interactive narrative trees consistent with author constraints while retaining the player's access to their full range of choice options. This is achieved by maneuvering players between possible world models consistent with their story world observations.

Perceptual Experience Management

by

Joseph Justus Robertson

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2017

APPROVED BY:

DEDICATION

To my parents, who taught me to follow dreams,
and Ebie, who follows dreams with me.

BIOGRAPHY

Justus loves computers, science, stories, and games. He studied all four at North Carolina State.

ACKNOWLEDGEMENTS

I want to first thank my advisor and mentor, Michael Young. Thank you for believing in me and investing your time in my future. I wouldn't be the person or researcher I am today without you. Second, I want to thank my co-advisor David Roberts. You're the only person I know who can turn grading exams into an anticipated end-of-year party (the pizza helps!). I also want to thank my committee members Rob St. Amant, Nick Taylor, and Ian Horswill. Thank you for all your help and support during my time as a graduate student. And thanks to Jon Doyle, Dennis Bahler, and Tiffany Barnes for playing important roles during my early years in the Ph.D. program, Laura Gipson, Mike Davis, and Bill Smith for being early academic influences, Dorianne Laux and The Church of Whitman for nurturing my poetic voice, PiRadical Studios for being my first game development teammates, and James Lester, Jon Rowe, Alok Baikadi, and The IntelliMedia Group for introducing me to academic research.

A big thank you to members of the 2011–2017 Liquid Narrative group. You all have been a family to me for the past six years. Stephen Ware and Brent Harrison, you taught me to excel at graduate school... and *Team Fortress 2*. Matt Fendt, you gave me a Super Mario Bros. film adaptation picture book. Julio Bahamón, Kara Brad Cassell, Zachary Litzinger, and Eric Lang, you're my favorite crime scene investigators. Camille Barot, I still have my *Cold Bagel Expectation* tattoo. David Winer, you have the best NC accent when you drink a Cheerwine. Adam Amos-Binks, I've been working on my Tusken Raider yell for Star Wars trivia. Colin Potts, 0% means there's a chance because Zeno's Arrow. Markus Eger, go WYLDKATZ! And Nate Brown, you're the one true heir to the Ketchup Cache. Thanks also to Jim Thomas, Brandon Thorne, and Jeff Ligon, members of the CIIGAR group including Ignacio Domínguez, Robert Loftin, and Sean Mealin, and to my trivia, ultimate frisbee, and basketball teams over the years, including Michael and Shannon O'Brien, Thomas Price, Kate Corbett, Drew van der Poel, Philip Parker, Ian Coleman, Abhinav Srinivasan, Dominic Zelenak, Aravind Ramachandran, Chris Vaughan, Justin Middleton, Scott Frazee, and Taiga Fukumori.

Finally, thanks to all my family and close friends for their support through my graduate life. To my parents, Angie and Danny Robertson, I got my Dr Pepper license! Thank you for being my first teachers, role models, and mentors. Thanks for always loving me, believing in me, and urging me to follow my dreams. To my sisters, Johanna and Jessa, thank you for being my first classmates, teammates, and friends. You are both industrious, talented, smart, and beautiful women, and I'm proud to be your brother. To Rogelio Cardona-Rivera and Jordan Floyd, thank you for being my teammates and best friends and for always having my back. To Moo Moo, Matt, Willow, Kevin, Marty, Nancy, Laura, Jason, Mere, Nick, and all my grandparents, aunts, uncles, and cousins, I love you all and thank you for being a part of my life. And finally, to my sweet Ebie... *try to untangle this!*

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

This document describes how a model of player knowledge can be used by an *experience manager* to dynamically configure story world content without the player noticing. An experience manager is an artificial agent that generates interactive stories and controls story execution in a virtual environment. A *strong story* experience manager is one that models an interactive story using a central data structure, similar to a branching script. This document shows how a strong story experience manager can dynamically configure content to increase the chance that predefined story constraints are maintained while the player acts freely in the story environment. This method of modifying the story world without the player noticing in order to advance author goals without hindering player autonomy is called *perceptual experience management*, because it shifts the player between perceptually equivalent worlds in order to preserve the experience manager's control of the interactive story environment. In this document I advance the following thesis:

Thesis A model of player knowledge allows strong story experience managers to better mediate between player autonomy and authorial control by allowing the agent to dynamically configure story world properties in order to increase the number of interactive narrative branches compatible with author constraints while avoiding modifications players notice.

Broadly, an experience manager is an automated story telling system which adapts the stories it tells based on feedback from a participant in control of a story character. A strong story interactive narrative agent models all possible story events as a central branching tree data structure where each path from the root to a leaf node models a linear story that conforms to constraints set by an author and each branch represents a choice the player makes during gameplay with unique outcomes. However, participants may have the autonomy to create story branches where authorial constraints break, which moves the story world the control of the experience manager's branching story script. My proposed thesis leads to two research questions:

1. How can story world manipulations be used to increase the number of interactive narrative branches consistent with authorial constraints?

2. How can a model of player knowledge be used to prevent story participants from noticing story world manipulations?

To answer the first question, this document presents an integrated game engine and experience management framework called the General Mediation Engine (GME). GME is a game engine that procedurally configures text or 2D interfaces to represent states in a branching story structure. This feature allows the experience manager to manipulate its interactive narrative world without modeling the game environment, because the interface dynamically configures itself to match the agent's representation. GME tracks story character knowledge according to a microtheory, or extensible model, of what characters come to know from interacting in a story world. GME uses the microtheory of character knowledge to model what the player knows at any point as an interactive narrative plays out. GME utilizes its model of the user's incomplete information of the story world to dynamically configure the environment and increase the probability that the author's story constraints are maintained. GME dynamically configures the interactive narrative tree with two processes: *event revision* and *domain revision.* The framework for a third method, called *superposition manipulation*, that proactively applies event and domain revision is also implemented and laid out by the document. Event and domain revision are evaluated by comparing the branching factor of interactive story trees produced by the two methods against a baseline in the context of GME. These evaluations are performed over three planning domain and problems and validate the mechanics of domain revision and event revision by showing the methods produce less dead ends than accommodation.

To answer the second question, this document presents a series of three experiments that test whether players notice the types of manipulations performed by a perceptual experience manager. All three experiments are performed online with participants from Amazon's Mechanical Turk service. The first is a Choose Your Own Adventure modeled on GME's text interface that relies on player feedback after the story to assess whether the outcomes of their actions were perceived to be believable. Unexpectedly, no significant difference was found between a story that contained no manipulations, a story that contained a manipulation consistent with player observations, and a story that contained a manipulation inconsistent with the player's observations. The second two experiments were designed to as a follow up to the initial experiment that reduce the story size to allow for a greater sample and relies on a direct measure of player behavior instead of self reporting. The two follow up studies contain a series of single-choice Choose Your Own Adventures modeled on experiments in reading comprehension literature where reading times of sentences are directly compared between stories. The follow up experiments provide evidence that players detect inconsistencies introduced by domain and event revision manipulations when the modifications are not constrained by a model of player knowledge.

The rest of this chapter provides a high-level overview of perceptual experience management and provides a roadmap of how the document's content is structured.

1.1 Motivation

This document presents a perceptual experience manager, a process that builds on interactive narrative generation and is implemented in this document as a plan-based interactive narrative agent. This section motivates this document's work in detail in the context of plan-based strong story interactive narrative.

1.1.1 Interactive Narrative

This document present a perceptual experience manager that automatically generates strong story interactive narratives. A narrative is a series of events, performed by story characters, that plays out in a story world. An interactive narrative is a story whose events are influenced by an outside participant who manipulates world states or characters. In the context of this framework, there is a single outside participant, called the player or client, who controls all the actions of a single story character. An automatically generated interactive narrative is an interactive story created by an algorithm. A strong story interactive narrative [46] is a system where story character behavior is centrally controlled according to the goals of an author. In automated strong story systems, the central author is an artificial agent. This artificial agent is called an *experience manager*. When the participant is allowed to control a character in a strong story interactive narrative, the player may act contrary to the script prepared for their character by the author. This tension between player autonomy and authorial control is called *the narrative paradox* [24] or the *the boundary problem* [26]. One way to overcome the boundary problem is to create a branching story structure that accounts for possible action sequences the player could choose to take.

Authoring one series of interesting narrative events is a hard problem that requires time to plan and execute. The amount of material an author must generate to create an interesting story is called the *authorial burden*. One way to model a branching story is by creating a tree of overlapping linear stories that branches for each decision the player can make [51]. Each unique event sequence in the branching story, represented as a path through its tree data structure, increases the branching story's authorial burden. If an interactive narrative tree branches uniformly by two unique choices, then the tree's authorial burden will increase exponentially by a power of two for each choice layer it contains [6]. This exponential rise in content for authoring interactive narrative trees limits the size and scope of interactive stories a human author, or even teams of human authors, can create on their own with this method. One way to mitigate the exponential burden of authoring branching stories is to automate the creation of interactive narrative trees.

1.1.2 Plan-Based Interactive Narrative

A process called *planning* can automatically construct linear narratives with a relatively low authorial burden. The planning process takes as input a planning problem and uses an automated planner to produce a solution plan. A planning problem describes what exists in a world, how the world functions when agents take action, and how the world should be configured at the end of the plan. An automated planner is a problem solver that takes as input a planning problem and performs search for a solution plan. A solution plan is any series of actions taken by agents in the world that begins from the problem's initial configuration and produces a goal state. Since a narrative is a series of actions taken by story characters, it can be modeled with a plan. If a linear narrative's story world is modeled as a planning problem, then the narrative will exist among possible solution plans and can be found by a planner through search. Through planning, the authorial burden of creating stories can be reduced from authoring whole sequences of events to defining a general story world, possible actions characters can take, and a set of constraints in the form of goal conditions. Furthermore, a planning problem defines a set of narratives that conform to the author's world and constraints instead of a single sequence.

Using a planner, a process called *mediation* can transform a linear narrative into a branching interactive narrative tree. Mediation builds its tree structures by iteratively creating and connecting the linear plans found by its planner. Mediation begins with a planning problem and uses its planner to produce an initial solution plan. The solution plan serves as a first full path through the interactive narrative tree. Mediation then analyzes the initial plan and finds every alternate course of action the participant's story character could take that would prevent the planning problem's goal state from being achieved. For each of these alternatives, mediation creates a new planning problem and asks its planner to find a solution. The solution plan serves as a new branch of the tree should the player decide to take the alternate action. This process is recursively invoked on each new branch until no alternate course of action remains. This iterative process, called *accommodation*, expands the branches of the interactive narrative tree. However, one drawback to this solution is that not all accommodative planning problems can be solved, so the participant could follow branches where no valid storyline exists.

One alternative mediation has to expanding the interactive narrative tree through accommodation is called *intervention*. Instead of adding a branch to the tree to account for an exceptional user action, intervention prevents the new branch from being formed and continues to use the current plan. Intervention prevents new branches by temporarily changing how the player's action transforms the story world to no longer produce a state that is contrary to the author's goals. One drawback of intervention is the participant may notice the world responds differently to their actions depending on the situation. This realization may break the user's sensation that they interact in a large world, full of unique possibilities, that is sensitive to the decisions they make. So neither accommodation or intervention is a perfect solution.

1.1.3 Perceptual Experience Management

An open problem in the field of experience management is how authorial control can be balanced with player autonomy. Neither accommodation nor intervention are a complete solution to this problem. Accommodation may favor player autonomy at the loss of authorial control and intervention may favor authorial control at the loss of player autonomy. In this document I utilize a model of player knowledge to perform *perceptual experience management*, a method that balances player autonomy and authorial control in more situations than baseline mediation. Perceptual experience management better balances player autonomy and authorial control by increasing the situations in which authorial control is maintained through accommodation and removing the situations where player autonomy is restricted through intervention. First, perceptual experience management increases the number of branches accommodation produces in an interactive narrative tree by widening the number of solution plans available to the planner. It does this with a process called *event revision*. Perceptual experience management also identifies and prevents interventions that contradict the player's experience. It does this with a process called *domain revision*. Together, event and domain revision allow the experience manager to shift the player between a set of world histories and mechanics that are consistent with what the player has observed during interaction in the story world and invisibly further the author's goals.

Event revision expands the set of solutions for accommodation by allowing the planner to change past events that were not observed by the player. To do this, the system uses a model of player knowledge to identify what past events the player observed. It then allows each of those marked events to be replaced with any alternative action that could have been taken and would not have been observed by the player. This process protects observed actions and ensures any event the player witnessed is retained. In addition to observed events, the process also identifies observed components of the world state and ensures the world as it was observed by the player is not altered by any action exchanges. This process allows accommodation to search the full space of trajectories that are consistent with the player's observations when looking for an alternative solution. Past trajectories consistent with the player's knowledge are the set of possible world histories from the player's perspective. So event revision allows mediation to expand its search for accommodations from the set of trajectories that are consistent with a single, static world history to the set of trajectories that are consistent with the player's knowledge.

Domain revision expands the set of solutions for accommodation by allowing the planner to change world mechanics that have not been observed by the player. The system does this by using a model of player knowledge to identify what author-specified action-outcome pairings the player has observed. It then allows any unobserved pairing to be removed from the domain if it causes an irreversible rift away from author goals. This process allows interventions to take place by removing unwanted effects and also ensures the system never reverts to the original set of mechanics. This process protects observed action-effect pairs and ensures any event outcome observed by the player is retained. In addition to observed event-outcome pairings, the process also identifies observed state literals and ensures the world as it was observed by the player is not altered by any domain removals. This process allows interventions to take place without violating the player's understanding of the world's mechanics. Alternate event-outcome pairings represent different sets of world mechanics. So domain revision allows mediation to shift the player between alternate story worlds that are consistent with the player's knowledge.

In addition to event and domain revision, this document also presents a general framework to proactively shift the player between alternate possible world histories and mechanics without waiting for a player to take an action that violates the current story. This framework, called *superposition manipulation*, models the set of possible world states consistent with the players knowledge at every point during gameplay. The set of possible world states is every world state reached by a history consistent with the player's knowledge model set in a universe with mechanics consistent with the player's knowledge model. In other words, the set of possible world states is the set of all possible states that could be reached by performing event and domain revision. This set of possible states is called the player's *superposition*. The utility of tracking a superposition is that whenever the player observes a state literal that is not consistent across the states in the superposition, the superposition must split. One successor superposition will be the worlds in which the observation is true and the other will be the worlds in which the observation is false. At this point, superposition manipulation may choose which of these possible superpositions it wants to transition the player into in order to maximize the probability that author constraints continue to hold.

1.1.4 General Mediation Engine

In addition to a theory of perceptual experience management, this document describes an implemented plan-based perceptual experience management system, called the General Mediation Engine (GME). GME is a state space mediator that builds turn-based interactive narrative trees based on feedback from a planner. GME is equipped to interface with any narrative or non-narrative planner capable of reading planning problems and returning solution plans. In addition to implementing a basic accommodation-driven mediator, GME also performs event revision, domain revision, and superposition manipulation. GME is capable of building interactive narrative trees using baseline accommodation or any combination of these features. In addition to building interactive narrative trees, GME provides a set of interfaces for exposing its trees to a human participant. It currently has several text interfaces and a visual 2D interface. The pipeline procedurally configures game worlds based on the underlying declarative experience management state and mechanics. This pipeline is useful for perceptual experience management, because no matter what change the experience manager makes to the world state the interface will reconfigure itself to match.

1.1.5 Evaluation

Finally, this document presents a series of structural and human subjects experiments that evaluate whether perceptual experience management answers the research questions laid out in this chapter. The first evaluations are a series of three structural experiments that generate trees for a given domain and problem using GME's different features. Each of the experiments compares a baseline accommodation tree to one produced by domain revision, event revision, or both working in tandem. The purpose of these experiments is to validate the first research question, whether story world manipulations produced by perceptual experience management can increase the number of interactive narrative tree branches consistent with author constraints. The first experiment is set in a Wild West domain and focuses on domain revision, the second experiment is set in a Batman domain and focuses on event revision, and the final experiment is set in a Spy domain and uses both event and domain revision. The experiments show that perceptual experience management produces trees with more branches consistent with author constraints when compared to baseline accommodation-driven mediation.

The second series of evaluations is three human subject experiments set in Choose Your Own Adventure stories similar to those constructed by GME. The purpose of these experiments is to validate the second research question, whether a model of player knowledge can be used to prevent participants from noticing story world manipulations. The first experiment shows that domain revision performs as well as a baseline, but so does intervention unconditioned on player knowledge. A follow up study and a third experiment use direct measures of player behavior instead of self-reporting and show that domain and event revision manipulations are noticed by human participants when they are unconditioned by a model of player knowledge. Together, these structural and human subjects evaluations answer the research questions posed by this section.

1.2 Roadmap

This document lays out a problem formulation, algorithms, implementation details, and a series of evaluations that answer the research questions laid out at the start of this introduction. Chapter 2 gives a summary of related work that covers planning and narrative generation, interactive narrative generation, perceptual simulations and alibi generation, procedural content generation, and a survey of interactive narrative experience evaluations in the context of Choose Your Own Adventures.

Chapter 3 continues by formally defining the *perceptual strong story interactive narrative problem*. This problem definition lays the groundwork for the system definition by describing how interactive narrative problems are modeled for perceptual experience managers. The model contains definitions for modeling story world objects and their properties, transitions between story world states, trajectories of states and transitions, sets of trajectory transition, state, and trajectory types, tree data structures that combine these sets and enable gameplay, a model for tracking player knowledge, a model for tracking possible story world states and histories, and the formal task of a perceptual experience manager. These definitions lay out the building blocks with which the system definition will create perceptual interactive narrative experiences.

Chapter 4 describes the components of perceptual experience management. Building off the problem description given in Chapter 3, this section describes how the problem components can be operated on to generate a perceptual interactive narrative tree and how the tree can be exposed to a player through a procedurally generated interface. Chapter 5 presents an implementation of the framework described in Chapter 4, called the General Mediation Engine (GME). GME is a game engine that implements the perceptual experience manager and exposes perceptual interactive narrative trees to the player through a procedurally generated interface.

Chapter 6 presents a series of four evaluations that address my two research questions. The first section uses GME to generate interactive narrative trees in order to test whether perceptual experience management modifications can create trees with a greater amount of authorial control over a baseline. This is tested by measuring the number of nodes at each level within the trees that conform to author constraints. If the question has been answered, perceptual interactive narrative trees should have an equal or greater number of nodes per level than baseline interactive narrative trees. The final three sections present a series of three human subject experiments that address the second research question, whether a model of player knowledge can be used to avoid manipulation detection by human participants. Finally, Chapter 7 concludes the document and discusses future work.

This document tells the story of perceptual experience management. Perceptual experience management is a framework for generating and controlling a perceptual simulation of an interactive narrative environment where the manager is capable of manipulating story world events and mechanics to better balance player autonomy and author control while retaining the player's perception of a consistent world.

Chapter 2

Related Work

This related work chapter is divided into several sections: planning and narrative generation, interactive narrative generation, perceptual simulation and alibi generation, knowledge modeling, procedural content generation, and Choose Your Own Adventure evaluations. The perceptual experience management framework presented in this document uses plan-based narrative generation to create branching tree structures that model interactive stories. The experience manager uses a model of player knowledge to transition the player between alternate possible worlds in order to maintain author constraints. To facilitate this process, the framework uses a procedural content generation pipeline to configure and manage an interactive visual interface for the branching tree structures. Finally, the system is evaluated in a series of human subject experiments in the context of Choose Your Own Adventure stories. This chapter situates the framework within a context of related research with a focus on plan-based systems.

2.1 Planning and Narrative Generation

One way to view narrative is as a series of events connected by an author and presented to an audience using words or images. Planning is one way stories can be represented and generated [67]. Planning is a method of finding action sequences that achieve desired world states through search. STRIPS [13] (The STanford Research Institute Problem Solver), an early action language and planner, was invented at Stanford Research Institute as part of the Shakey the Robot project [37]. STRIPS is the name of both a problem solver, or planner, and a logic-based language used to model world states and world update mechanics. A planning problem expressed in STRIPS notation models an initial and goal world state and a set of action operators. STRIPS operators consist of a set of preconditions that must be true for the action to be taken and a set of positive or negative postconditions that become true after the action is performed. The STRIPS planner searches through a space of states by applying operators enabled in a current state that create resulting states. The search concludes when a goal state is reached and the series of actions taken to reach the goal is returned as a plan. This method of constructing plans by searching through a space of states is called state space planning.

Since STRIPS, other planning methods have been created that construct plans without searching a state space. A system called UCPOP [42] is an example of a plan space planner. Every interior node in UCPOP's search space is a partial, or incomplete, plan. Partial plans consist of steps, which correspond to action operators, ordering constraints, which specify a total ordering between two steps in the partial plan, and causal links, which specify that an effect of one step is needed to satisfy a precondition of another, later step. UCPOP begins at a root node that consists of the initial and goal states modeled as steps and expands the space of partial plans by creating and resolving flaws. A partial plan has a flaw whenever there exists a step precondition that is not connected by a causal link to another step's effect or when there is an ordering conflict between two steps. Flaws are resolved by adding steps, causal links, or ordering constraints to partial plans. UCPOP ends its search when it encounters a leaf node with no remaining flaws. The plan associated with the leaf node is returned by UCPOP as the solution to the planning problem. This method is called plan space planning because search is performed across a space of nodes that represent plan data structures.

One problem with the proliferation of planning methods after STRIPS is that many different custom action languages were used for representing planning problems. For example, UCPOP operates with an action language called ADL [41] (Action Description Language), which has more expressive power than STRIPS. In 1998, the International Conference on Automated Planning and Scheduling (ICAPS) ran its very first planner competition [33] to compare different planning methods and search heuristics to see which found solutions most efficiently across a range of planning domains and problems. In order to compare these different planning methods using a range of benchmark planning domain and problems, the competition needed a unified, shared action language for describing worlds and update mechanics. An action language called PDDL [32] (Planning Domain Description Language) was created for the first competition and has become the standard method of modelling planning problems. The framework presented in this document models interactive worlds using PDDL.

The ICAPS planning competition led to advancements in the speed of heuristic-guided state space planners. A heuristic is a rule of thumb or estimate that informs how the search algorithm expands its search graph. Heuristics should lead the search towards optimal solutions by solving easier sub-problems or approximations of the overall problem. Advances in heuristics were made in the early years of the competition that allowed state space planners to outperform other methods in terms of speed. The Fast-Forward planner [20] was the most successful planner in the second planning competition. Fast-Forward computes its heuristic by solving a simplified version of the planning problem from each node, similar to Graphplan [5]. The framework presented in this document primarily uses the Fast Downward [19] planner, a successful state space system first introduced during the fourth planning competition that makes representation-based improvements over Fast-Forward. While Fast Downward is used as the primary planner, the framework described in this document can interface with any algorithm that takes as input planning problems and produces solution plans.

One problem with using planning to generate narrative is that not all sequences of events are interesting stories. One open research question is how to reason about or identify interesting stories in a solution space. One planner built in the UCPOP tradition, called IPOCL [52], is an algorithm that adds narrative reasoning to the planning process. IPOCL expands its PDDL input to include subjective goals for individual characters. These character goals allow the system to ensure all characters in the plans it produces act according to their interests. This additional layer of reasoning allows the algorithm to exclude stories where characters behave erratically or irrationally. Another algorithm, CPOCL [62], is an iteration on IPOCL that allows and identifies conflict between characters in story plans. A planner called Glaive [63] is a heuristic state space planner based on Fast-Forward that produces plans equivalent to those found by IPOCL and CPOCL. In addition to Fast Downward, this framework bundled with and capable of finding linear intentional stories with Glaive.

2.2 Interactive Narrative Generation

There are many approaches to generating interactive narratives. Riedl and Bulitko [46] classify interactive narrative systems along three dimensions: virtual characters, authorial intent, and player modeling. Virtual characters describe how autonomously characters in the story behave, whether they can act on their own or are controlled by a central storyline. Authorial intent describes how strongly the system depends on authored content, whether story content is manually created or generated by an algorithm. Player modeling describes to what extent the storyline adapts to learned preferences of individual players. The rest of this section will use these dimensions to situate this document's framework in the context of other approaches.

One popular form of interactive storytelling is the Choose Your Own Adventure [40] (CYOA) book series. The pages of a CYOA present story events like a normal book, but at the bottom of each page the CYOA prompts the reader with a decision where each choice option corresponds to a page number. Once the reader makes a choice, they turn to the page number that corresponds to their decision, and are presented with new story content followed by new choice options. This story-decision cycle continues until the reader reaches an ending, which is a page with no choice options. Books in the CYOA series are on the strongly authored end of Riedl and Bulitko's scale. CYOAs provide no character autonomy because the behaviors of all characters are pre-scripted and the books are strongly authored by a human with no automation.

On the other side of the spectrum from CYOAs, emergent narrative [2] systems, like FearNot! [3], take a highly automated, decentralized approach to interactive narrative generation. Emergent narratives are characterized as having high character autonomy and high automation. Emergent narratives arise through unscripted interactions between a user and virtual characters in an interactive environment. The authorial burden of creating emergent narratives is not in storyline generation, for there is no central storyline, but through crafting interesting and robust virtual characters with the ability to react and improvise along with the player. Since there is no central story controller, the agents that control character behavior must create interesting experiences for the player through their local behavior.

Between CYOAs and emergent narratives are systems like Façade [29] and Versu [11]. Façade is a hybrid system that balances central plot control with autonomous agents and pre-authored content with automation. In Façade, the player assumes the role of a dinner guest and interacts with a virtual couple, Grace and Trip, as they work through marital problems. The game's character behaviors are written with a reactive planning system, ABL [28], which is an iteration on the Hap [25] agent behavior architecture pioneered by the OZ Project [4]. ABL allows an author to create beats, or general story components that can be arranged at a high level and decompose into executable character behaviors [28]. Each beat progresses the story and beats are chosen by the system depending on what the player does during interaction. Versu is a text-based hybrid system that allows players to provide suggestions within social situations while remixing the different characters that participate within each situation.

The framework presented in this document controls interaction using a central script, like a CYOA, but instead of being hand authored, the stories used to control characters are automatically generated. The framework descends from Mimesis [66], a narrative generation and control system integrated with the Unreal Tournament [14] (UT) engine. Mimesis uses a plan space planner called Longbow [65] to create plans that correspond to actions virtual characters could take in the UT environment. Mimesis uses a process called mediation [47] to adapt its plan to any action the player could take while interacting in the game. The data structures mediation creates to adapt its plan, called mediation trees, are representationally equivalent to a CYOA [51], but consist of stories that are automatically constructed with a planner.

Since Mimesis, several other plan-based interactive narrative systems have been created. ASD [49] is a mediation-based framework that decouples itself from any one game engine by creating a modular API that allows its mediator to sense an external game environment once connected [48]. ASD moves closer to Façade's mixed story and character autonomy system by decoupling character control from the current plan. Proactive Mediation [17] is a successor to Mimesis' mediation system. Instead of waiting for the player to take exceptional actions, it proactively arranges the interactive story world to prevent exceptional behavior from taking place. Another system, The Merchant of Venice [43], allows players to make interventions that change how characters behave, instead of controlling any one character directly. Finally, the PAST [44] system is built on the ASD and PaSSAGE [60] frameworks that uses real-time planning to select between stories on the fly based on a predicted player type of the interactor. The system is tested with a Little Red Riding Hood choose your own adventure game that uses pre-scripted text fragments to convey story events to the player.

2.3 Perceptual Simulation and Alibi Generation

Perceptual experience management increases the branching factor of mediation trees by tracking what the player knows about story world and taking advantage of their incomplete knowledge to create story alibis in a perceptual story world simulation. Alibis and perceptual simulations were first used by Sunshine-Hill [58] to make non-player characters (NPCs) in large, sandbox worlds appear to act intelligently for relatively little computational cost. As the player interacts in the sandbox, NPCs are spawned, take action according to a random distribution, and then are destroyed once they leave the player's area. However, if the player begins to pay attention to any NPC, the system creates an alibi that explains why the character is performing the current action and from that point forward the character behaves according to the alibi. Li [23] applied alibi generation to narrative domains by allowing non-player characters to formulate backstories using crowdsourced plot graphs [22].

Similar to alibi generation, Initial State Revision (ISR) [50] allows narrative planners to dynamically reconfigure the initial state of a planning problem in order to better facilitate story creation. Initial state literals can be set to true or false, like in a normal PDDL problem, but can also be set to undetermined. Undetermined literals can be changed to true or false by the planner at run time. The Most General Initial State [61] later introduced a streamlined version of ISR by allowing domain authors to succinctly specify the range of initial states ISR can reach in a single data structure. Perceptual experience management's event revision process can be thought of as an ongoing initial state revision constrained by a model of player knowledge. An emergent narrative system, called the Virtual Storyteller [59], uses a concept called late commitment that is similar to initial state revision. Late commitment is a feature where story characters can improvisationally change the state and rules of the story world simulation.

Perceptual experience management takes advantage of a player's incomplete knowledge of their environment. In order to do this, the framework needs a theory that predicts what story characters observe and what they know about their world. However, creating a robust theory of player knowledge is outside the scope of this dissertation. To make the problem tractable, this framework uses a microtheory [16] to create and maintain its model of player knowledge. A microtheory is a modular and extensible way to specify a model of some domain that can later be expanded or substituted for a more complete theory. This framework uses a simple microtheory of character knowledge that relies on co-location, but this can be substituted at a later time for a more complex model.

2.4 Plan-Based Procedural Content Generation

The final component of the framework is a procedural content generation (PCG) pipeline that creates a visual interface for the interactive narrative plan trees created by mediation. A few other systems have used plans as a basis of procedural content generation. Hartsook [18] procedurally generates game world layouts that support a given narrative plan and Zook [70] uses planning models to procedurally generate game mechanics. This system uses a PCG pipeline to automatically create, configure, and maintain virtual worlds based on an underlying PDDL representation in order to support story alibis in the game world.

2.5 Choose Your Own Adventure Evaluations

As part of its evaluation, this document presents a series of studies in the context of Choose Your Own Adventure stories that aligns with a growing body of work in the area. Mawhorter et al. [30] outlines a theoretical framework for modes of engagement, choice idioms, and player experiences in Choose Your Own Adventures. These interactive CYOA stories are modeled as a series of choice framings, or situations in which a choice is being made, choice options, actions a player can take in a given situation, and choice outcomes, the effects of the player's choice on the story world. Mawhorter presents a general framework of how players engage with CYOAs, what kinds of choices players can make in a CYOA, and how these choices can impact player experience. A growing body of research is testing this theoretical framework with a series of human subjects evaluations.

One of the most important feelings that an interactive story can provide to its participants is *agency*. Agency is the satisfying feeling of taking action and effecting meaningful change in the story world. Several studies have been performed to measure feelings of agency in the context of Choose Your Own Adventure stories. Fendt et al. [12] presented one of the earliest CYOA human subjects studies. The document provides evidence that players who were given unique local feedback to their choices felt agency at a similar level to those who played a truly branching interactive story with both unique local and global feedback. A follow up study [8] provides evidence that players feel heightened agency when choosing between choice options that lead to world states that players foresee as being meaningfully different. Other work has examined the link between choice options, player behavior, and story enjoyment [68, 69] and a validation of a generative theory of certain pairings of situations and choice options, like dilemmas [31]. This document presents an evaluation of domain and event revision by comparing reading times in the context of short Choose Your Own Adventures, one choice in length. This study builds off work done in the area of reading comprehension [39, 1] that measures when readers encounter story inconsistencies that break story coherence.

2.6 Summary

This chapter provides a context of related work that situates the perceptual experience management system presented by this document. It began by explaining the history of plan-based narrative generation, explained how past systems have generated interactive experiences using plan-based systems, covered past systems that have performed some version of alibi generation, presented other plan-based approaches to procedural content generation, and situated the human subject evaluations presented in this document in the context of a growing body of work on measuring player experiences in Choose Your Own Adventures. The next chapter describes the problem of perceptual experience management.

Chapter 3

Problem Description

This chapter presents a formal definition of the perceptual strong story interactive narrative problem. The problem definition allows for multiple initial and goal states, multiple world mechanics in the form of operator libraries, and partial knowledge of the story world by each character. This problem description lays a foundation and builds a formal language for the perceptual experience management framework presented in Chapter 4.

3.1 World Representation

A strong story interactive narrative is a branching story with one or more human participants where the non-player characters are controlled by an automated system according to some central formal narrative model. Interactive stories take place in an environment called a story world. In this model, story worlds consist of locations, characters, and objects. These three types of story world entities are represented in the formal model by constants.

Definition 1 (Constant). A *constant* is an entity that exists in the story world being modeled. In this model, constants can be characters, locations, or physical objects.

This chapter uses an example called the Key Domain to illustrate its concepts. The Key Domain is a simple example planning problem and domain with a single actor controlled by a human participant. The objective of the Key Domain is to reach a room through a door that must be unlocked. Full PDDL for the Key Domain example is available in Appendix A.3. There are ten constants in the Key Domain that represent story world objects: a Player, a Key, four Doors, and four Rooms. The Key Domain constants are pictured in Figure 3.1.

Characters are special story entities, called story agents, that can change the story world configuration by taking action.

Definition 1.1 (Story Agent). A *story agent* is a story character, represented by a constant, capable of taking action and effecting change in the story world.

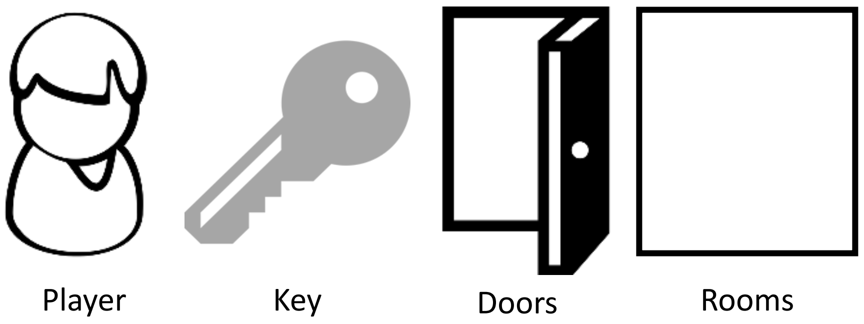


Figure 3.1: World objects in the Key Domain, represented as constants. There are ten constants: a Player, a Key, four Doors (A-B, A-C, B-D, C-D), and four Rooms (A, B, C, D). The Player is the only story agent in the Key Domain.

Story world configurations are formed by mapping constants to logical predicate variables. Different assignments of truth values to ground atomic formulae represent different states of the story world.

Definition 2 (State). A story world *state* is a truth assignment to possible atomic formulae of the world model.

An example Key Domain state graphical representation is given in Figure 3.2. In the example state, the Player is at Room A which has an open door leading to Room B and Room C. The Key is at Room C. Room B and Room C both have a locked door leading to Room D.

A story world changes when characters take action. The next section describes how states in this model are updated by character actions.

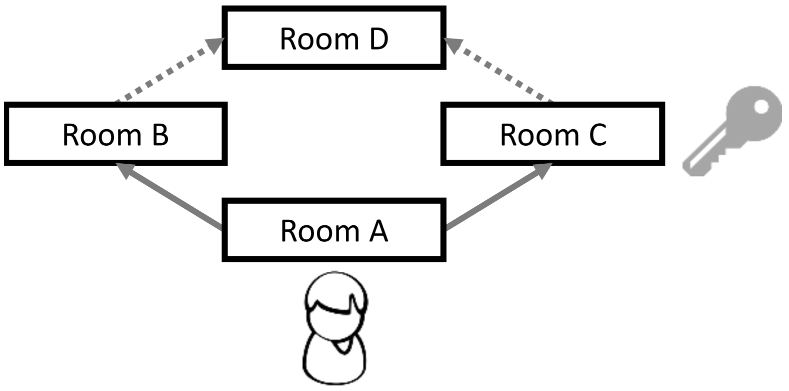


Figure 3.2: A Key Domain world state. The arrows between rooms represent doors. Solid arrows represent open doors. Dotted arrows represent locked doors. The Player is at Room A and the Key is at Room C.

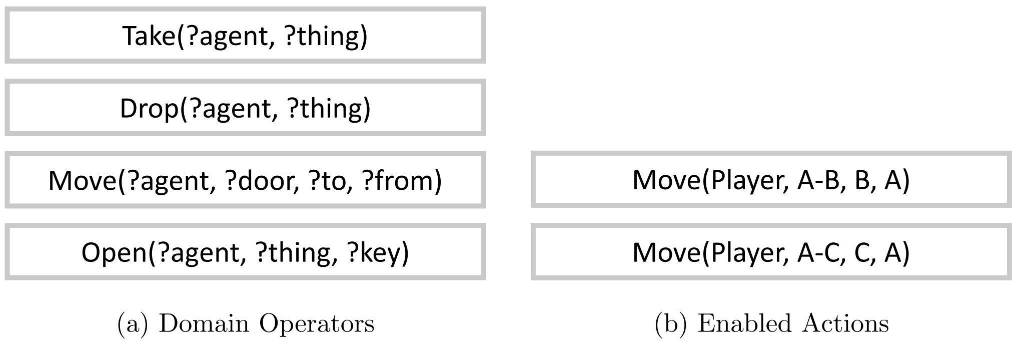


Figure 3.3: Figures that represent the set of operator templates in the Key Domain and the set of actions available to the Player from the state pictured in Figure 3.2. Figure 3.3a shows the set of operators in the Key Domain. The word before the open parenthesis is the title of the operator and each word beginning with a ? inside the parentheses is an unbound variable. Figure 3.3b shows the set of fully ground actions enabled by the state pictured in Figure 3.2.

3.2 World Dynamics

Character actions are represented in the model by instantiated, fully ground operator templates. An operator template describes a type of action story agents can take, in what states the action can be performed, and how the action affects the world state.

Definition 3 (**Operator**). An *operator* is a tuple $\langle p, e \rangle$ of precondition formulae p and effect formulae e. For some operator o, o's preconditions are referred to as p_o and its effects e_o.

The Key Domain's set of operators is given in Figure 3.3a. In the Key Domain, characters can take something they are colocated with, drop something they are carrying at their current location, move from one location to another through an open door, and open a locked door they are colocated with using a key.

When variables in an operator template's precondition and effect formulae are fully ground with world constants, the operator is called an action.

Definition 4 (**Action**). An *action* is a tuple $\langle o, b \rangle$ of some operator o and set of bindings b where b contains a mapping for every free variable in p_o and e_o to some world constant. For some action a, a's fully bound preconditions are p_a, its bound effects are e_a, and its bindings are b_a.

An action is called an instantiation of its abstract operator.

Definition 4.1 (**Instantiated Operator**). Actions are instantiations of their operator templates. An action a is an *instantiation* of an operator o if after setting $b_a \leftarrow \emptyset$, $p_a = p_o$ and $e_a = e_o$.

If the preconditions of an action all hold in a given state, the action is enabled in the state.

Definition 4.2 (Enabled Action). An action a is *enabled* in state s if $p_a \in s$.

An example of enabled actions is given in Figure 3.3b, which pictures actions that are enabled in the state from Figure 3.2. From the state, the player can either move from Room A to Room B or move from Room A to Room C.

All actions belong to some agent, called the actor.

Definition 4.3 (Actor). All actions are carried out by an agent, which is a character in the story. When action a is carried out by an agent g, g *performs* a and a's *actor* is g.

Together, a story world state and an operator library describe a transition system of states connected by enabled actions.

Definition 5 (State Transition System). A *state transition system* is a labeled directed graph whose vertices are states and edges are enabled actions. A state transition system can be represented with the tuple $\langle s, \omega \rangle$, where s is a state and ω is a set of operators.

Applying an enabled action to a state produces a successor state.

Definition 5.1 (Successor State). Any enabled action performed by an agent on a state in a transition system produces a *successor state*. When an enabled action a is performed by an agent g in state s it produces successor state s', where each ground atomic formula in s' matches s unless it is updated by an effect in e_a.

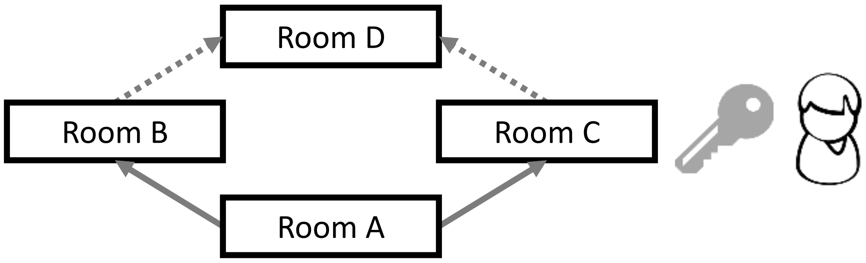


Figure 3.4: The successor state produced by the player performing *Move(Player,A-C,C,A)* from the Key Domain state shown in Figure 3.2. The result is that the Player is no longer at Room A and is now at Room C.

An example successor state is given in Figure 3.4. It shows the state that results when the *Move(Player,A-C,C,A)* action is applied to move the Player from Room A to Room C from the state in Figure 3.2.

A sequence of enabled actions and their corresponding successors in a transition system from an initial state is called a trajectory.

Definition 6 (Trajectory). A *trajectory* $t = (s_0, a_1, s_1, a_2, s_2, ..., a_n, s_n)$ is a series of alternating states and actions that describes a series of enabled actions performed in a state transition system from some initial state s_0 along with each resulting successor state. The sequence of actions in a trajectory t is referred to as α_t and the sequence of states in a trajectory σ_t.

An example trajectory with three states and two connecting actions is given in Figure 3.5. It begins with the state from Figure 3.2 and progresses to the state from Figure 3.4 through the *Move(Player,A-C,C,A)* action. It then progresses to a state where the Player is holding the Key through the *Take(Player,Key)* action.

A trajectory reaches the final state in its sequence. The final state in a trajectory is the state reached by performing its series of enabled actions from the first state in the trajectory.

Definition 6.1 (Reached State). A state s is *reached* by trajectory $t = (s_0, ..., s_n)$ if $s = s_n$.

Planning is the process of searching for trajectories that reach a transition system state where given goal conditions are enabled. The next section describes sets of solution plan trajectories.

3.3 Planning

A planning problem describes a set of trajectories through a state transition system where goal conditions are enabled. It is the task of a planner to find trajectories that belong to the set specified by the planning problem. Planning problems are comprised of a state transition system, which is an initial state and set of operators, and a goal state.

Definition 7 (Planning Problem). A *planning problem* is an initial state, or complete starting truth assignment, a goal state, a truth assignment over a subset of world state formulae, and a set of operators. Planning problems are represented $\langle i, \gamma, \omega \rangle$ where i is the initial state, γ is the goal state, and ω is the set of operators.

A plan is a solution to a planning problem. It is a trajectory from the planning problem's initial state that reaches a goal state.

Definition 8 (Plan). A *plan* for planning problem $\langle i, \gamma, \omega \rangle$ is a trajectory $t = (s_0, ..., s_n)$ through the state transition system $\langle i, \omega \rangle$ where $s_0 = i$ and $\gamma \in s_n$.

A solvable trajectory is any trajectory from which a goal state can be reached by adding a further sequence of enabled actions.

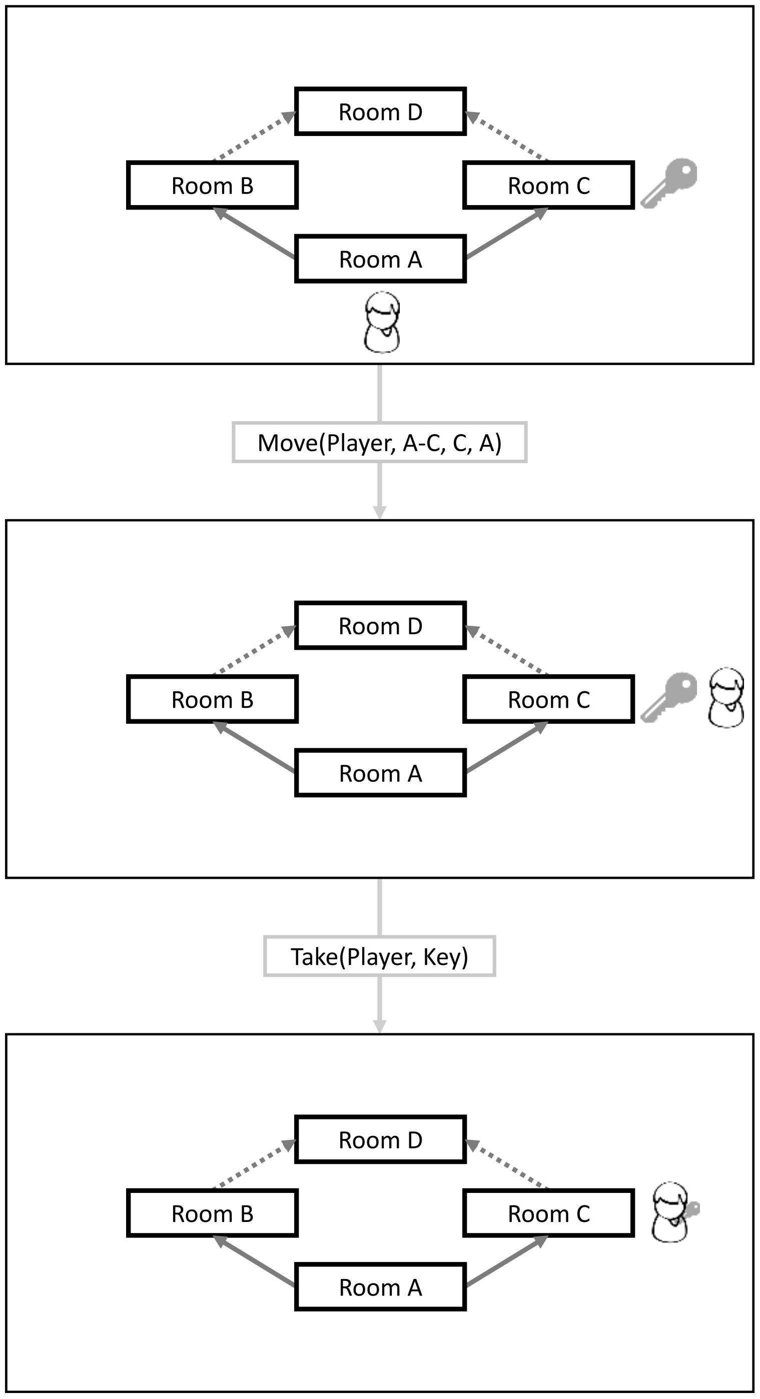


Figure 3.5: A trajectory through states in the Key Domain. The topmost box pictures the initial world state where the Key is at Room C and the Player is at Room A. The middle box shows the world after the Player moves to Room C. The bottom box shows the world after the Player takes the Key at Room C.

Definition 8.1 (**Solvable Trajectory**). A trajectory t is *solvable* for planning problem $P = \langle i, \gamma, \omega \rangle$ if there is some plan p for P where t is a substring of p.

It is the experience manager's task to keep the story world on a solvable trajectory while navigating its state transition system.

A story can be modeled as a plan that achieves goal conditions specified by a domain author and story generation can be automated as planning. However, interactive narrative systems must account for alternate courses of action that could be taken by a human participant. The next section describes how these alternate actions can be used to create a branching story structure.

3.4 Interactive Narrative Planning

In interactive narrative domains, one or more of the story agents acting in the transition system is a client outside the planner's control. These clients are called players.

Definition 9 (**Player**). A *player* is an interactive narrative client playing the role of a story agent by controlling that agent's actions in an interactive story environment.

Given freedom to act, players may disrupt the system's plan by taking enabled actions that contradict the story trajectory. The goal of interactive narrative generation is to plan contingencies for any deviation a player may perform.

This interactive narrative problem definition allows for multiple initial states, similar to Initial State Revision [61], multiple goal states, similar to ASD's fourth tier of replanning [49], and multiple independent sets of operators, which represent different sets of possible world mechanics.

Definition 10 (**Interactive Narrative Problem**). An *interactive narrative problem* is a set of initial states, a set of goal states, and a set of operator libraries. Interactive narrative problems are represented $\langle I, G, O \rangle$ where $I = \{i_0, i_1, ..., i_n\}$ is a set of initial states, $G = \{g_0, g_1, ..., g_n\}$ is a set of goal states, and $O = \{\omega_0, \omega_1, ..., \omega_n\}$ is a set of sets of operators.

An example IN Problem is shown in Figure 3.6. The problem has three initial states: one where the Key begins at Room A, one where it begins at Room B, and one where it begins at Room C. The problem has a single goal state literal, that the Player is at Room D.

A set of actions, trajectories, solvable trajectories, states, and plans can be derived from each interactive narrative problem. These sets bound what events, states, and solutions can occur during interaction. The set of actions describe every possible instantiation of operators in the set of operator libraries with constants from each initial state.

Definition 10.1 (**Action Set**). An interactive narrative's *action set*, A, consists of every valid binding of each operator $o \in \omega \in O$ using every combination of constants in each $i \in I$.

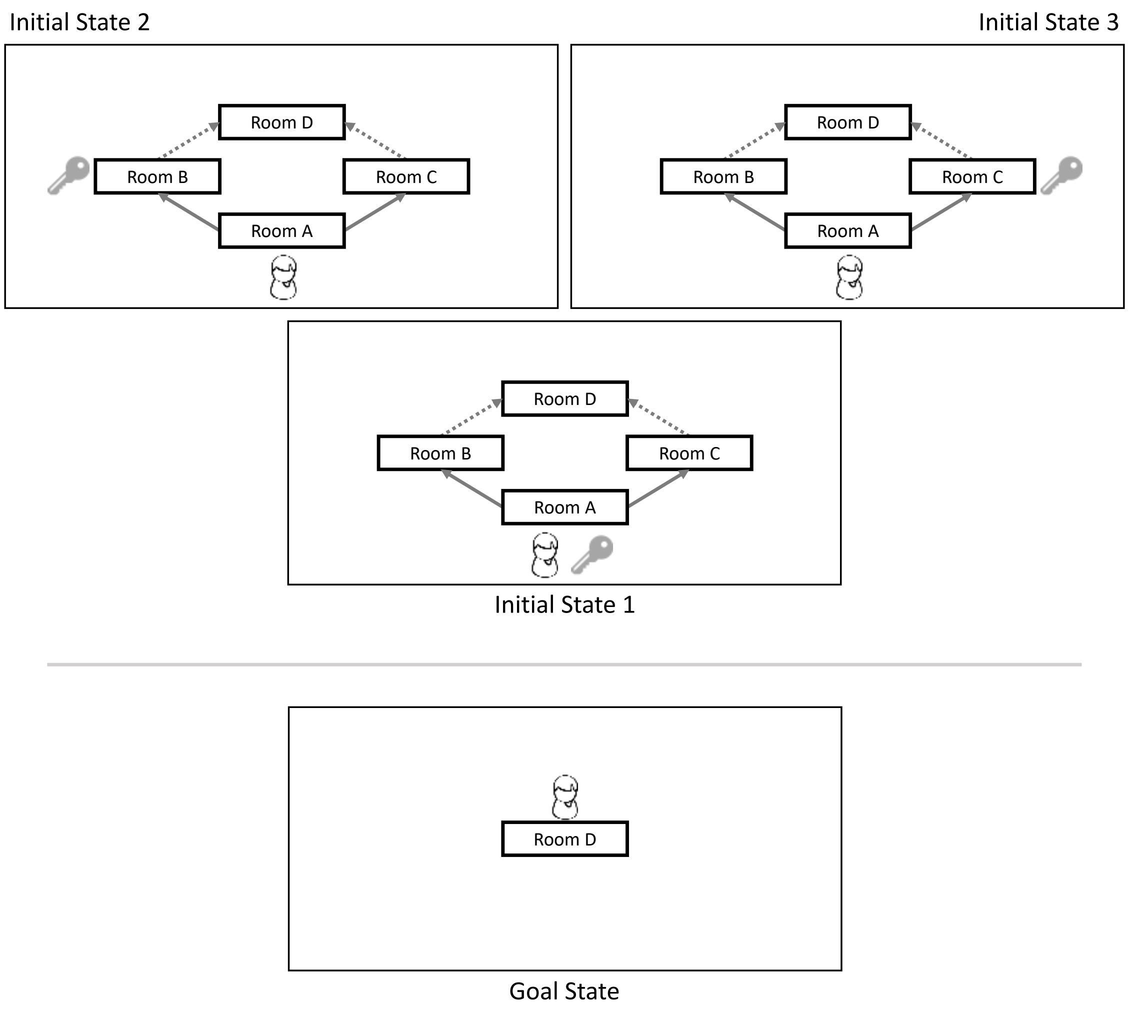
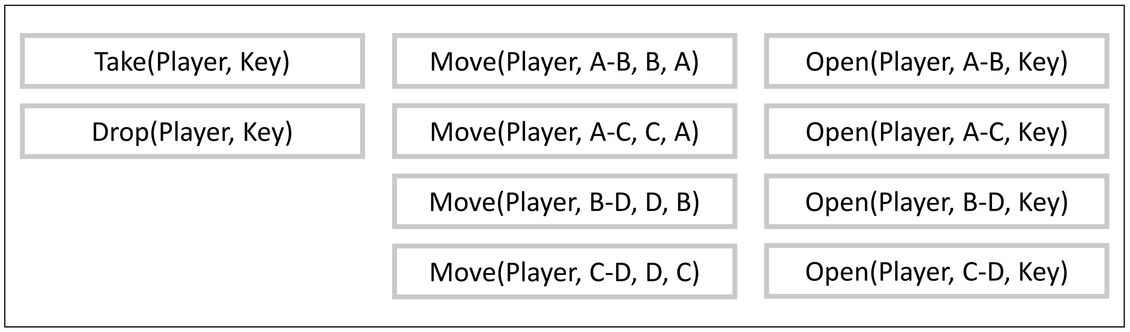


Figure 3.6: A Key Domain interactive narrative problem with three initial and one goal state. The initial states differ in Key placement and the goal is for the Player to be at Room D.



Take(Player, Key)
Drop(Player, Key)
Move(Player, A-B, B, A)
Move(Player, A-C, C, A)
Move(Player, B-D, D, B)
Move(Player, C-D, D, C)
Open(Player, A-B, Key)
Open(Player, A-C, Key)
Open(Player, B-D, Key)
Open(Player, C-D, Key)

Figure 3.7: The full action set of the Key Domain interactive narrative problem given in Figure 3.6. These are all actions that could possibly be performed by the Player in the world of the Key Domain from the problem definition given in Figure 3.6.

An example action set is given in Figure 3.7. This represents all possible actions that could occur given the world described by the interactive narrative problem in Figure 3.6. The Player can possibly take the Key, drop the Key, Move from A to B, A to C, B to D, C to D, or open A-B, A-C, B-D, or C-D with the Key.

The set of trajectories describe every possible way the state transition systems bound by the problem can evolve over time.

Definition 10.2 (Trajectory Set). An interactive narrative's *trajectory set*, T, consists of all possible trajectories from each $i \in I$ using each $\omega \in O$.

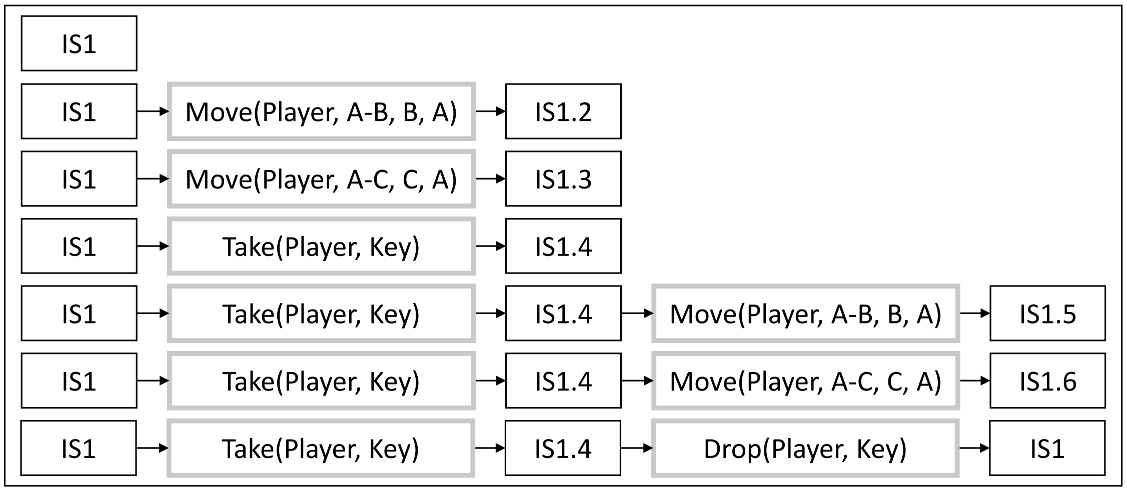

IS1
IS1 → Move(Player, A-B, B, A) → IS1.2
IS1 → Move(Player, A-C, C, A) → IS1.3
IS1 → Take(Player, Key) → IS1.4
IS1 → Take(Player, Key) → IS1.4 → Move(Player, A-B, B, A) → IS1.5
IS1 → Take(Player, Key) → IS1.4 → Move(Player, A-C, C, A) → IS1.6
IS1 → Take(Player, Key) → IS1.4 → Drop(Player, Key) → IS1

Figure 3.8: A portion of the trajectory set of the Key Domain interactive narrative problem given in Figure 3.6 for trajectories starting from Initial State 1 with up to two actions.

An example trajectory set for the Key Domain problem is given in Figure 3.8. It shows all possible trajectories of length 1, 2, and 3 states from Initial State 1 in the interactive narrative problem shown in Figure 3.6.

The set of solvable trajectories describe every possible sequence of actions from which a goal state can be reached.

Definition 10.3 (Solvable Trajectory Set). An interactive narrative's *solvable trajectory set*, V, consists of all solvable trajectories from each $i \in I$ using each $\omega \in O$.

The set of states describe every possible state that can be reached by some trajectory.

Definition 10.4 (State Set). An interactive narrative's *state set*, S, consists of all possible states reached by some trajectory $t \in T$.

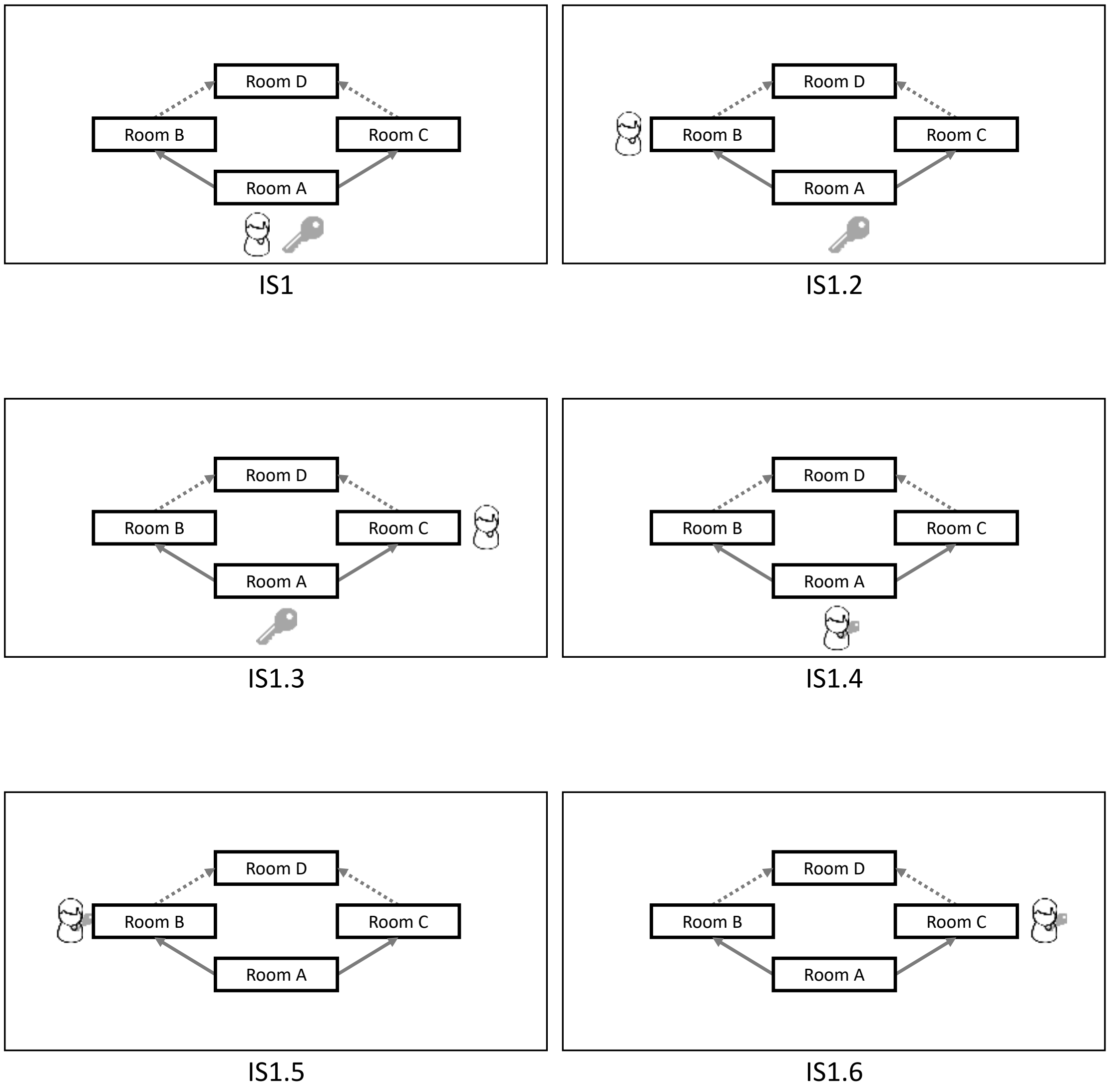


Figure 3.9: A subset of the state set of the Key Domain interactive narrative problem given in Figure 3.6 reached by trajectories in Figure 3.8.

An example state set for the Key Domain problem is given in Figure 3.9. It shows six possible states reachable from Initial State 1 in the interactive narrative problem presented in Figure 3.6 reached by the seven trajectories in Figure 3.8. IS1 is reached by both the single state trajectory and three state trajectory where the player picks up then drops the Key at Room A.

Finally, the set of plans describe every trajectory that reaches a goal state.

Definition 10.5 (Plan Set). An interactive narrative's *plan set*, P, consists of all possible trajectories from each $i \in I$ to every state s such that $\exists g \in G$ where $g \subset s$.

Interactive narratives do not exist in isolation, they are meant to be performed with players in an interactive environment. An interactive narrative environment can be defined as an interface that exposes the state transition system defined by an interactive narrative problem to players. The next section describes how a branching story graph can be used to control interaction during gameplay.

3.5 Interactive Narrative Play

One way to conceptualize the possible ways a state transition system can progress over the course of an interactive narrative is to enumerate its trajectories as a tree.

Definition 11 (World Tree). An interactive narrative's *world tree*, w, from some initial state i over some set of operators ω is the tree formed from the set of trajectories starting at i using only actions in ω.

An example world tree is given in Figure 3.10. The world tree is constructed from the seven trajectories in Figure 3.8 and arranged according to the different choices the player can make as events play out from the initial state by taking one outgoing edge or another to reach child states.

There is a world tree for every combination of initial states and operator libraries in the interactive narrative problem. A collection of world trees is called a world forest.

Definition 12 (World Forest). An interactive narrative's *world forest*, W, is the forest of all possible world trees given a set of initial states I and a set of operator sets O.

An example world forest is given in Figure 3.11. Each world forest shows pictures all possible trajectories up to three states in length from Initial States 1, 2, and 3 in Figure 3.6.

Gameplay is the act of iteratively progressing through a world tree as players consume an interactive narrative. During gameplay, the *world history* is a series of events that has played out in the game world. The world history corresponds to a single trajectory in T. The world history

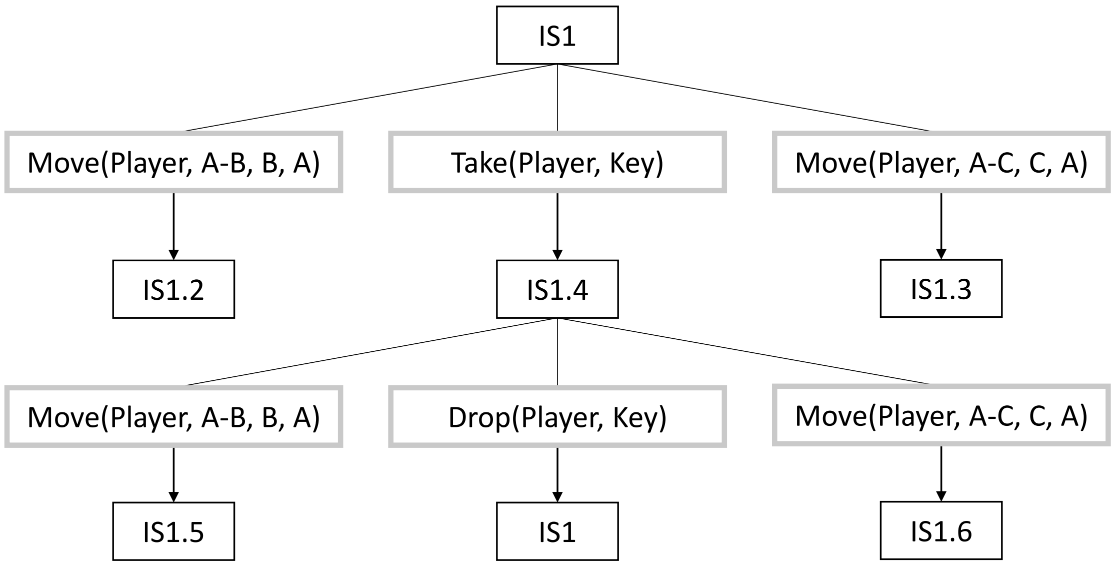


Figure 3.10: The world tree that represents the trajectory subset given in Figure 3.8 of up to two action trajectories from Initial State 1.

leads to the *current state*, the current state of the story world. Interactive narratives require participation from players. The system must offer an interface to expose state configurations and allow players to take enabled actions in order to provide gameplay. A *game world* is an environment that provides an interface for clients to act as story agents and perform gameplay on a world tree. The game world exposes the current state to players and allows them to take enabled actions that correspond to outgoing edges from the current state in the world tree.

Players may not have complete knowledge of the story world during gameplay. Incomplete player knowledge can be used to further the system's goals. The final component of the system definition is a model of what characters observe as they interact in the game environment. The next section describes an extensible model of character knowledge, called a microtheory.

3.6 Agent Knowledge

Story agent knowledge can be tracked using a microtheory. A microtheory is an extensible model of some domain that can be changed or replaced without affecting the overall system. Since building a robust model of how story characters come to know the world around them is outside the scope of this system, this definition uses a collection of axioms to predict character knowledge. The theory's rules can be extended or replaced without affecting how the system uses the model.

Definition 13 (Knowledge Theory). A *knowledge theory* is a collection of axioms Mt that,

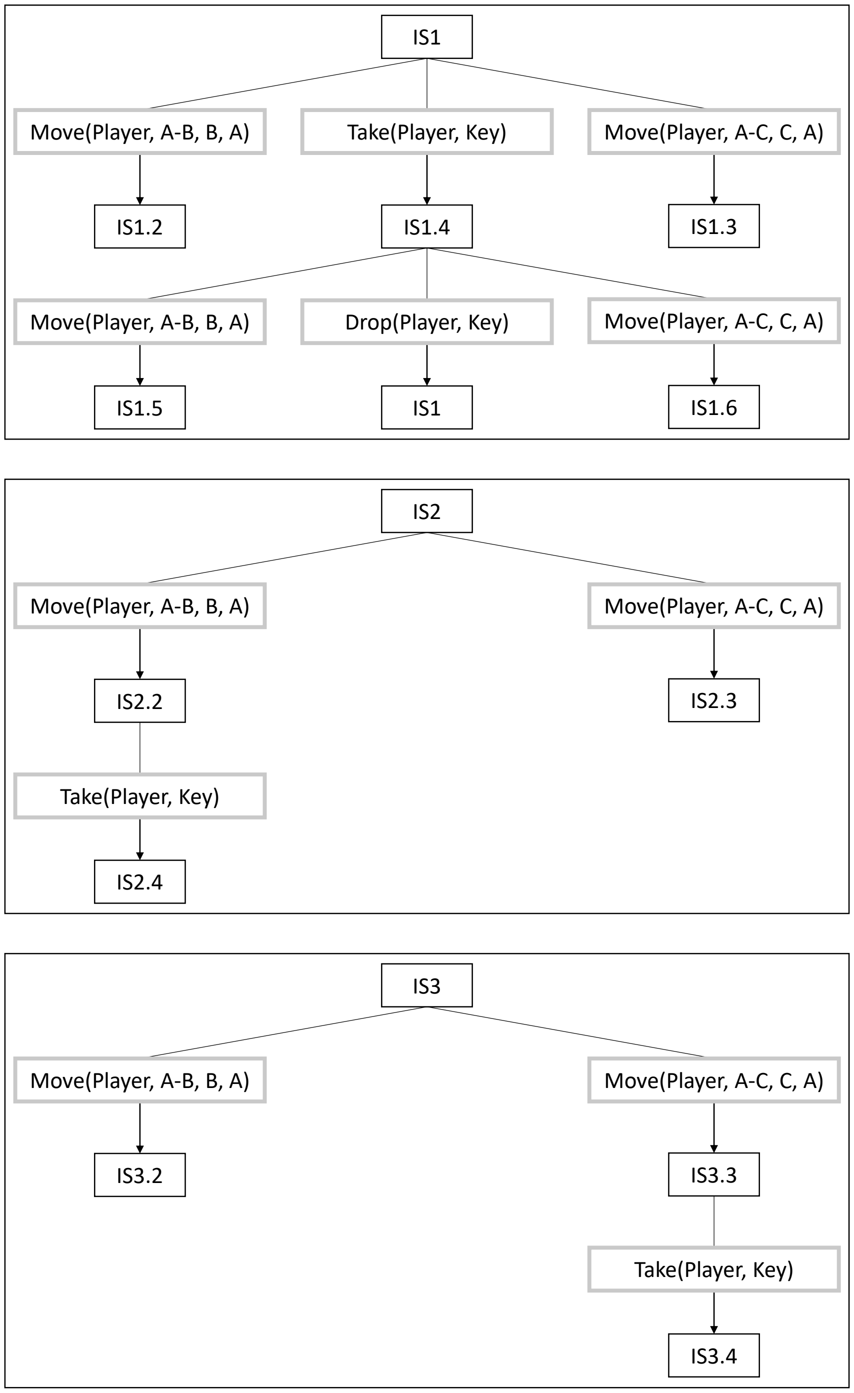


Figure 3.11: The forest of world trees for the interactive narrative problem given in Figure 3.6 on the subset of trajectories of length two for Initial States 1, 2, and 3.

given a history trajectory h and an agent g, describe a subset of actions in α_h and a subset of atomic formula in each $s \in \sigma_h$ that g has observed.

The particular axioms of Mt defines a trajectory for each agent participating in the interactive narrative.

Definition 14 (**Agent Knowledge**)**.** During gameplay, an agent g's *knowledge* k_g describes what g knows about the world history h given a knowledge theory Mt. For each action and each state literal in each state in the world history h, k_g indicates whether agent g observes or does not observe the state literal or action.

An agent's knowledge of the world history describes a superposition of possible trajectories that may have occurred.

Definition 15 (**World History Superposition**)**.** During gameplay, an agent g's *world history superposition* is a set $H_g = \{t_0, t_1, ..., t_n\}$ of trajectories that could possibly be the world history from an agent g's perspective according to Mt. A trajectory $t \in T$ belongs to H_g if the truth value of every observed state literal in k_g matches the truth value of the corresponding state literal in t and each observed action in k_g appears in t. There could be trajectories in H_g that correspond to paths through multiple trees in W if the agent has limited knowledge of h's initial state or operator library.

Two example world history superpositions of length one are given in Figure 3.12. This superposition corresponds to the initial states from the problem presented in Figure 3.6. Initial States 2 and 3 are initially superposed because the Player won't know if the Key is at Room B or Room C from Room A.

Any trajectory that belongs to an agent's superposition is consistent with what the character knows according to the microtheory.

Definition 15.1 (**Consistent Trajectory**)**.** A trajectory t is *consistent* with an agent g's knowledge if $t \in H_g$.

The trajectories in H_g describe a set of possible world states the client g could exist in. Since the history trajectories consist of a single state, Figure 3.12 is also an example world state superposition.

Definition 15.2 (**World State Superposition**)**.** During gameplay, an agent g's *world state superposition* is a set $S_g = \{s_0, s_1, ..., s_n\}$ of states that could possibly be the current world state from an agent g's perspective according to Mt. A state $s \in S$ belongs to S_g if it is reached by a trajectory $t \in H_g$.

Given the world history superposition of every client participating in the interactive narrative, the system's task of maintaining a solvable trajectory can be defined. The next section formally defines the problem of perceptual experience management.

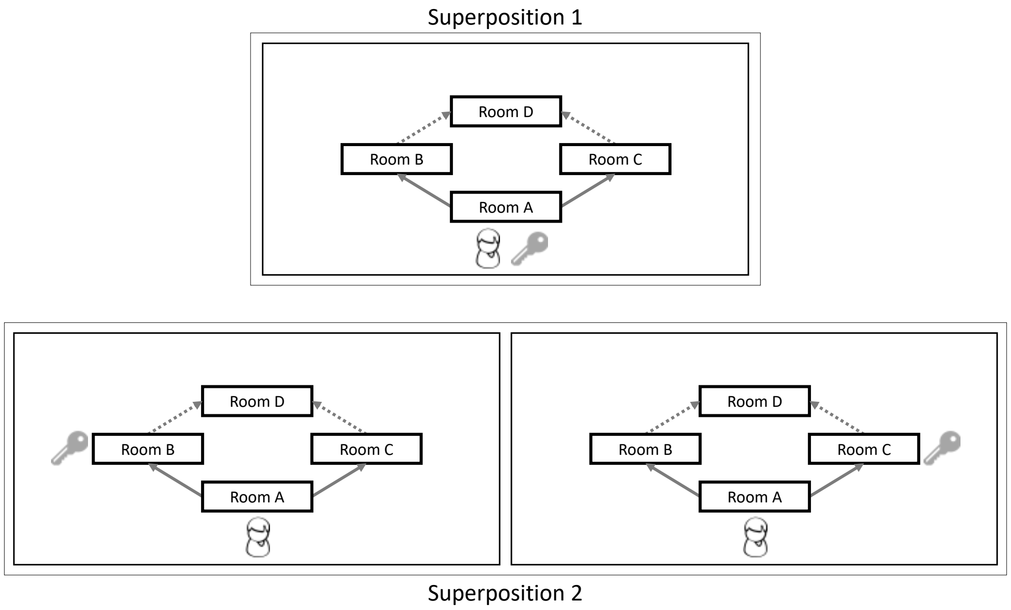


Figure 3.12: The two superpositions from the initial state of the interactive narrative problem shown in Figure 3.6, given the microtheory that a character observes everything they are colocated with. If the player starts in Superposition 2, they could exist in Initial State 2 or 3.

3.7 Interactive Narrative Task

A trajectory is both solvable and consistent if a goal state can be reached from the trajectory and the trajectory belongs in the superposition of an agent.

Definition 16 (Consistent, Solvable Trajectory). During gameplay, a consistent and solvable trajectory for agent g is any trajectory t where $t \in V, H_g$.

During gameplay, every character has a set of trajectories that are consistent and solvable given their superposition.

Definition 16.1 (Consistent, Solvable Trajectory Set). The set of all consistent and solvable trajectories for an agent g is referred to as VH_g.

The task of an automated, strong story, interactive narrative system is to ensure that each client has at least one solvable trajectory consistent with their observations.

Definition 17 (The Interactive Narrative Task). During gameplay, for every *player* ensure $VH_{player} \neq \emptyset$.

3.8 Summary

An automated, strong story, interactive storyteller's objective is to allow players to take action and effect change in a story world while also controlling events according to a set of pre-authored constraints. This section casts that task as ensuring at all times during gameplay ensure for all players $VH_{player} \neq \emptyset$. In other words, for each client ensure there is a possible world history consistent with the client's observations that enables a future series of events which reaches a goal state. Chapter 4 presents a plan-based experience management framework that takes advantage of this problem description by tracking player knowledge in order to dynamically configure the story world to achieve author goals while ensuring the player doesn't notice. This process of tracking and manipulating the story world according to a model of player knowledge is called *perceptual experience management*.

Chapter 4

Perceptual Experience Management

This chapter presents a description of a perceptual experience management framework. The framework dynamically builds relaxed world trees from an interactive narrative problem for a single player, exposes world tree data structures to the player through a procedurally generated interface, and performs revisions to the story world state and update mechanics in order to better maintain author constraints while providing full player autonomy. Modifications to the story world state are performed by a process called *event revision* and modifications to the update mechanics are performed by a process called *domain revision.* This section also outlines a process for proactively performing event and domain revisions, called *superposition manipulation.* Superposition manipulation transitions the player between world state superpositions to maximize the probability that the story world will reach a goal state, given a model of superposition utility and expected player action. Together, these components allow an experience manager to perform *perceptual experience management.* A perceptual experience manager generates a *perceptual interactive narrative*, or an interactive narrative tree that appears as a consistent simulation to the player but can be dynamically modified by the experience manager.

4.1 Introduction

Chapter 3 defines the task of automated, strong story interactive narrative systems as ensuring at all times during gameplay, for all players, $VH_{player} \neq \emptyset$. The chapter defines this task in terms of abstract sets. In practice, these sets are large and cannot be exhaustively generated. The first hurdle to overcome when solving the interactive narrative task for a particular problem is to find trajectories that belong in VH_{player} without expanding the full set of possibilities. This chapter presents a plan-based approach to finding members of VH_{player} through search. This framework makes two improvements over existing strong story experience managers that allow it to ensure that $VH_{player} \neq \emptyset$ holds in more situations. These two improvements, event revision and domain revision, are the basis of perceptual experience management. These components work by creating a *perceptual simulation* of the story world.

4.1.1 Perceptual Simulation

A perceptual simulation is some virtual world that appears to be a full, consistent simulation to an outside observer but is actually a façade that simulates just enough to appear consistent [58]. Without perceptual simulation, experience managers are limited to selecting a single *simulation trajectory* in VH_{player}. A simulation trajectory is the default behavior of a story world.

Definition 18 (Simulation Trajectory). A *simulation trajectory*, referred to as t_{sim}, is the default history that occurs in a story world if events take place without perceptual experience management.

Creating a perceptual simulation allows experience managers to search the full space of VH_{player} by finding two new types of consistent trajectories that belong to H_{player} in addition to the simulation trajectory. The first type of trajectory differs from the simulation in terms of its events.

Definition 19 (Event Difference). An *event difference* between two trajectories consistent with an agent g's knowledge, $t_1, t_2 \in H_{player}$, is when t_1 and t_2 differ in their series of actions $\alpha_{t_1} \neq \alpha_{t_2}$ and all actions in t_1 and t_2 are instantiations of operators from a single operator library ω.

If two trajectories have an event difference, they are alternate histories within the same universe. Both trajectories share the same operator mechanics, but their event sequences differ. The second type of trajectory differs from the simulation in terms of its domain.

Definition 20 (Domain Difference). A *domain difference* between two trajectories consistent with an agent g's knowledge, $t_1, t_2 \in H_{player}$, is when t_1 and t_2 differ in their series of states $\sigma_{t_1} \neq \sigma_{t_2}$ due to actions in t_1 and t_2 being instantiations of operators from two different operator libraries, ω_1 and ω_2.

If two trajectories have a domain difference, they are the same story world history set within different universes. Both trajectories share the same course of events, but they differ in how the actions have affected the world. The two components of perceptual experience management, event revision and domain revision, enumerate the full set of trajectories in the set of possible world histories H_{player}. From the simulation trajectory, event revision generates the subset of H_{player} that are event differences from t_{sim} and domain revision generates the subset of H_{player} that are domain differences from t_{sim}. In addition to the differences appearing separately, any two trajectories can have concurrent event and domain differences. Used together, event and domain revision can generate the subset of H_{player} that have both event and domain differences.

4.1.2 Event Revision

The first component of perceptual experience management is event revision. Previous strong story experience managers do not utilize a model of client knowledge when building branching story data structures using *accommodation*. Accommodation is the process of reacting to diverging player actions in order to build a branching story where all paths are consistent with author constraints. Not incorporating a model of player knowledge into accommodation artificially restricts H_{player}, the world history superposition, to a single trajectory. This single trajectory is the one that corresponds to the full simulation series of events that have played out in the story world. This restriction holds if the player has complete knowledge of all story events. However, in most cases the player will only know a portion of what has happened in the story world. In these cases when the player has incomplete knowledge of the story world, the full simulation model limits VH_{player}, the set of consistent and solvable trajectories, only to those that begin with the full simulation member of H_{player}. Accessing the full set of H_{player} by considering all consistent trajectories according to a model of character knowledge strictly increases the trajectories that belong to VH_{player} and ensure that $VH_{player} \neq \emptyset$ in more gameplay situations. Section 4.3 presents the event revision algorithm. Event revision searches the full range of trajectories in VH_{player} when performing accommodation given a microtheory of character knowledge, a client character, and a world history.

4.1.3 Domain Revision

The second component of perceptual experience management is domain revision. Previous strong story experience managers do not utilize a model of client knowledge when executing *interventions*. One type of intervention is a shift between two alternate models of world mechanics to prevent an unwanted outcome of a client's action. In plan-based systems, world mechanics are modeled with operator libraries. So interventions in plan-based systems can be represented as shifts between alternate sets of operators that represent similar actions but have different sets of preconditions and effects. If the experience manager is allowed to jump back and forth between operator libraries at will, it is commonly thought that the client will notice inconsistencies in the way the world operates. As a result, plan-based systems like ASD [49] remove the feature entirely. However, if the experience manager only shifts between operator libraries when both are consistent with what the client has experienced, then the client should not notice an inconsistency. Allowing intervention to cancel out effects of player actions that contradict authorial goals is a second way to ensure that $VH_{player} \neq \emptyset$ in more gameplay situations. Section 4.4 presents the domain revision algorithm. Domain revision restricts the set of interventions to those that are consistent with the player's experience.

4.1.4 Superposition Manipulation

The two framework components expand the number of possible trajectories in VH_{player} that an experience manager can search. The final piece of the theoretical framework will actively make decisions to increase the probability that at least one trajectory exists in VH_{player} at a later point in time. This is achieved by maintaining a data structure that represents the superposed set of states S_{player} the player exists in at the current point in time. Every time this superposition is collapsed due to the client making an observation, the component uses a heuristic to determine the collapsed superposition. A good heuristic allows the system to maximize the probability that the story world will reach a goal state. Section 4.5.1 presents a method for modeling and updating state superpositions. Section 4.5.4 presents a framework for evaluating the utility of superposed states given a utility heuristic and oracle of player choice.

4.1.5 Roadmap

This chapter lays out the theoretical description of perceptual experience management. Section 4.2 defines the formal components of plan-based, state-centric, accommodative mediation, which forms the baseline for this perceptual experience management framework. Sections 4.3, 4.4, and 4.5 describe the two components of perceptual experience management and a framework for proactively using the two to shift the player between possible worlds. Section 4.6 gives a running example of each method and the proactive framework in the context of the Spy Domain PDDL given in Appendix A.4. Chapter 5 continues the technical discussion by describing how perceptual experience management components have been implemented in the context of the General Mediation Engine (GME). GME is a game engine that consists of a plan-based experience manager and a procedural content generation pipeline that utilizes the underlying experience management data structures to create and configure a game interface with which the player can interact.

4.2 Mediation

This perceptual experience management framework is realized as a plan-based agent, called a mediator. This section describes a baseline non-perceptual mediation agent.

Definition 21 (**Mediation**)**.** A *mediator* is a strong-story interactive narrative agent that builds branching story structures by first generating a linear narrative and then creating contingencies for possible ways a player-controlled character could disrupt the planned story.

The framework is driven by a planner, which is an algorithm that solves planning problems.

Definition 22 (**Planner**)**.** A *planner* is an algorithm that takes as input a planning problem, $\langle i, \gamma, \omega \rangle$, and produces a plan, $t = (s_0, ..., s_n)$.

Given a planning problem and a planner, the framework generates a mediation game tree. A mediation game tree is a relaxed world tree that incorporates a plan at every vertex for controlling non-player characters.

Definition 23 (Mediation Game Tree). A *mediation game tree* is a world tree where there is exactly one player-controlled character, each character g is given a sequential turn order for taking action $(g_0, g_1, ..., g_{n-1})$, and a plan is associated with every vertex. The root of the tree is the initial state i. The outgoing edges of any vertex s at tree level j is the set of actions $\alpha \in A$ that are enabled in s and performed by character g_k at sequence position k where $k = j \mod n$ and n is the number of characters in the problem. Actions that non-player characters take are dictated by the plan associated with the outgoing vertex. The plan associated with the root node is a solution to the planning problem given by the planner.

As new levels of the mediation game tree are generated, the original plan must be updated to reflect the updated world state at each newly expanded successor node. One way to accomplish this task is to query the planner at each new state vertex to find a plan that will reach a goal. However, planning is not an easy task and this may become time consuming.

Instead of generating a new plan at every vertex, the framework utilizes a plan management strategy that asks the planner for a new solution only when necessary. The system accomplishes this management task by first drawing dependencies between actions in the current plan.

Definition 23.1 (Dependency). A *dependency* is an interval in a trajectory t that begins at a state s, ends at an action a_j, and is associated with an atomic formula p where p is a precondition of a_j and s is the successor of the last action a_i that is ordered before a_j in t and has p as an effect. If no such action exists, s becomes the initial state s_0.

An example set of dependencies is given in Figure 4.1. Each action has a one or more dependency that is drawn from a preceding state. For example, the initial state supports the action *Take(Player,Key)* with the literals the Player at Room A and the Key at Room A. It also supports the action *Move(Player,A-B,B,A)* with the literal the Player at Room A. Each following action has at least one precondition fulfilled by a state later than the initial state in the trajectory.

For every precondition of every action in the current plan, the system draws its corresponding dependency. From the plan and set of dependencies, the system can determine how it should respond to the player's action by placing it in one of three categories. The system classifies each player action as either constituent, consistent, or exceptional. The first type of action the player can perform is a constituent action.

Definition 23.2 (Constituent Action). A *constituent* action is one that is prescribed by the current plan. All non-client actions are constituent since they are taken from the current plan.

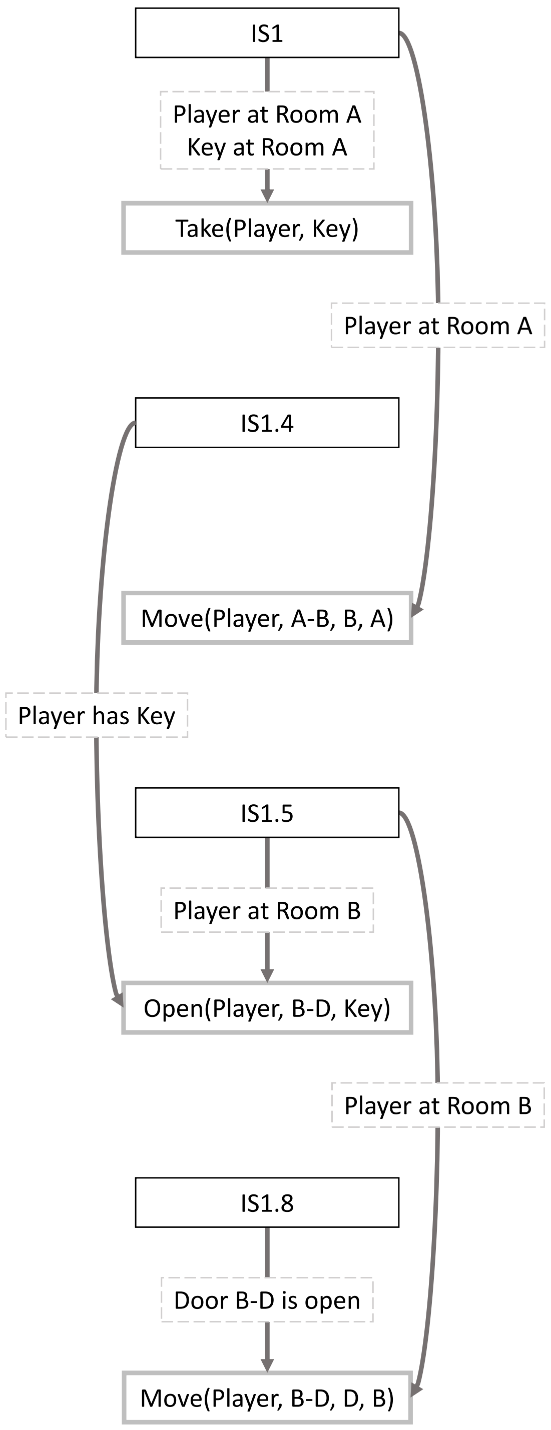


Figure 4.1: A plan for the Key Domain problem given in Figure 3.6 with its dependencies drawn as arrows and labeled with the formula it represents.

Constituent actions are those the system has planned for the user to take. To update the plan after a constituent action, the system removes the taken action from the trajectory and updates its dependencies. This process is identical to the system update process after an NPC has taken its turn. The second type of action is a consistent action.

Definition 23.3 (**Consistent Action**). A *consistent* action is one that is not in the plan and whose effects do not reverse any atomic formula attached to a dependency in the initial state.

Consistent actions aren't planned by the system, but they don't disrupt the current plan by reversing a dependency needed to execute a future action. When a consistent action is performed by the player, nothing about the current plan has to be updated. The final type of action is an exceptional action.

Definition 23.4 (**Exceptional Action**). An *exceptional* action is one that is not in the plan and at least one of whose effects reverses an atomic formula attached to a dependency in the initial state.

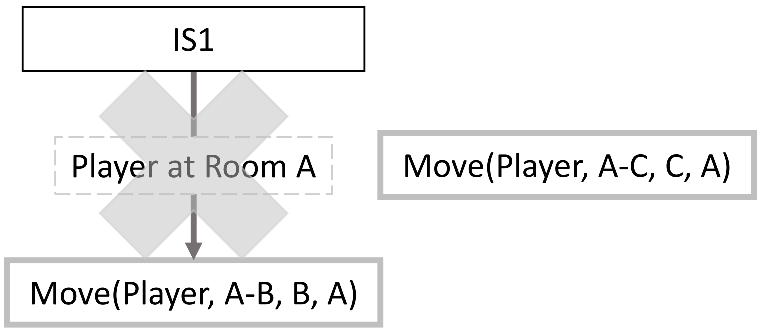


Figure 4.2: An exceptional action over the dependency between IS1 and *Move(Player,A-B,B,A)* in the plan given in Figure 4.1.

An exceptional action is one that breaks the current plan by reversing a literal that satisfies a planned future action's precondition. When an exceptional action is taken, a new plan must be found to control execution. The system creates a new planning problem where the vertex's state becomes the new initial state and queries its planner for a solution to the new problem. The solution returned by the planner becomes the new plan at the vertex. If the planner does not return a solution, the game tree has entered the set of non-solvable trajectories from which there is no return in a non-perceptual simulation.

An example exceptional action is given in Figure 4.2. It shows the Player at Room A dependency from the trajectory in Figure 4.1 being broken by the *Move(Player,A-C,C,A)* action.

If the Player chooses to move to Room C they will no longer be at Room A and can no longer move from Room A to Room B as the original plan prescribes. This broken dependency makes *Move(Player,A-C,C,A)* an exceptional action that requires replanning to repair.

As described in this section, the mediation system is over-constrained to consider only plans that match the simulation path through the game tree, t_{sim}, when the client may exist in a superposition of possible consistent trajectories from the initial state through different domains given their partial knowledge of the game world. Two methods allow the system to perform perceptual experience management by transitioning the client between all trajectories in the set of possible world histories H_{player}: event revision and domain revision. The first method, event revision, searches through world histories containing different events than t_{sim} but still compatible with what the player has observed during gameplay.

4.3 Event Revision

Event revision [53, 54] explores a larger portion of the client's world history superposition when planning during gameplay. Event revision is capable of finding trajectories with alternate histories in H_{player} instead of defaulting to the path taken through the mediation game tree, which is the single trajectory t_{sim}. The process proposed in this section is platform independent. It is presented modularly and is meant to be integrated into a state space planner's action selection process. There are two phases of event revision: removal, where the system creates a subjective history by removing actions and state formulae unobserved by the client's character from the current trajectory, and planning, where the system searches for a new plan that is compatible with the remaining client observations.

Algorithm 1 The *Remove* method creates and returns a trajectory that contains all world events and world state literals an agent has observed during gameplay according to a microtheory of knowledge.

```
REMOVE (World History Trajectory $h$, Knowledge Microtheory $Mt$, Story Agent $g$)
    trajectory $k_g \leftarrow h$
    for each action $a \in \alpha_{k_g}$
        if $g$ does not observe $a$ according to $Mt$
            $a \leftarrow \emptyset$
    for each state $s \in \sigma_{k_g}$
        for each atomic formula $f \in s$
            if $g$ does not observe $f$ according to $Mt$
                $s \leftarrow s - f$
    return $k_g$
```

The first phase of event revision, removal, takes place before the system searches for a plan. During this phase, the player's knowledge trajectory, k_{player}, is created where all unobserved actions and atomic state formulae in the world history are removed. Pseudocode for the action removal phase is given in Algorithm 1. Note that the world history trajectory h's set of states σ must contain all true and false literals for each world state bounded by the planning problem's object and predicate lists, not just the positive literals stored under a closed world assumption. A list of positive literals would provide only the player's observations of positive state literals, but they also observe negative literals as well. The trajectory returned by Algorithm 1 is the client's knowledge of world history events, k_{player}. This trajectory represents all things the player knows happened in the story world according to the microtheory.

Algorithm 2 The *Constrain* method returns a set of enabled actions for a given state in a modified planning problem that allows past events to be replanned if they are consistent with player observations.

```
CONSTRAIN (Current Trajectory $t$, Agent Knowledge $k_{player}$, Knowledge Microtheory $Mt$,
    Story Agent $g$, Current Actor According to MGT Turn Ordering $c_a$)
  $s \leftarrow$ state reached by $t$
  $\alpha \leftarrow$ set of actions enabled in $s$
  if $|\alpha_t| > |\alpha_{k_{player}}|$ return $\alpha$
  for each action $a \in \alpha$
      $s_a \leftarrow$ successor state created by applying $a$ to $s$
      $t_a \leftarrow$ trajectory created by appending $(a, s_a)$ to $t$
      if Consistent(Remove($t, Mt, g$), $k_{player}$) is false or $a$'s actor is not $c_a$
          $\alpha \leftarrow \alpha - a$
  return $\alpha$
```

The second phase of event revision is planning. During this phase, the system uses a modified planning algorithm to search for a solution plan from the initial state of the planning problem at the root of the mediation game tree. The planning algorithm used is modified to constrain enabled actions prior to the current state of gameplay to those that are consistent with the client's observations and turn ordering. This method would be used during a planning algorithm's expansion process when it generates enabled actions for a given state. Instead of the full set of actions, the set would be filtered through the *Constrain* method. Pseudocode for generating the constrained enabled actions is given in Algorithm 2 and pseudocode for a subprocess that determines whether a world history trajectory is consistent with player knowledge is given in Algorithm 3.

Together, these algorithms enable the system to explore the subset of H_{player} that corresponds to the initial world tree when replanning so that all possible world histories consistent

Algorithm 3 The *Consistent* method determines if a world trajectory is perceptually consistent with the player's knowledge of world events.

```
CONSISTENT (Current Trajectory $t$, Agent Knowledge $k_{player}$)
  for each action $a_t \in \alpha_t$ at position $i$
    if action $a_{k_{player}} \in \alpha_{k_{player}}$ at position $i$ is not $\emptyset$ and not equal to $a_t$  return false
  for each state $s_t \in \sigma_t$ at position $i$
    if $s_t$ is not a superset of $s_{k_{player}} \in \sigma_{k_{player}}$ at position $i$  return false
  return true
```

with the client's observations using the initial operator library can be found. It can, with a small extension, also be used to move between trees in the problem's world forest to paths that start with alternate initial states. Figure 4.3 shows event revision used to move between initial states based on a player's decisions in order to reach a state where the player and key are co-located. If the Player moves to Room B, the system will choose the initial state where the Key is at Room B but if the Player moves to Room C, the system will choose the initial state where the Key is at Room C. Both of these outcomes are consistent with the player's knowledge of the Key Domain. However, with event revision alone the system is still over constrained since it can only use the single domain model the framework begins with. The second component, domain revision, allows the framework to invisibly transition between different domains.

4.4 Domain Revision

Algorithm 4 The *Alternate* method creates and returns a trajectory that contains the same exact world events as the current world history, but uses an alternate operator library.

```
ALTERNATE (World History Trajectory $h$, Operator Library $\omega$)
  trajectory $h' \leftarrow h$
  for each sequential action $a \in \alpha_h$ at position $i$ beginning at $i = 1$
    if there is an operator $o \in \omega$ such that $a$ has the same name and variables as $o$
      $\alpha_{h'_i} \leftarrow o$ instatiated as action $a'$ using bindings from $a$
      $\sigma_{h'_i} \leftarrow$ successor state $s'$ of $\alpha_{h'_i}$ applied to $\sigma_{h'_{i-1}}$
    else
      $\alpha_{h'_i} \leftarrow \emptyset$
      $\sigma_{h'_i} \leftarrow \sigma_{h'_{i-1}}$
  return $h'$
```

Domain revision is a method of transitioning between alternate trees in the interactive narrative problem's world forest W that correspond to possible world histories with alternate,

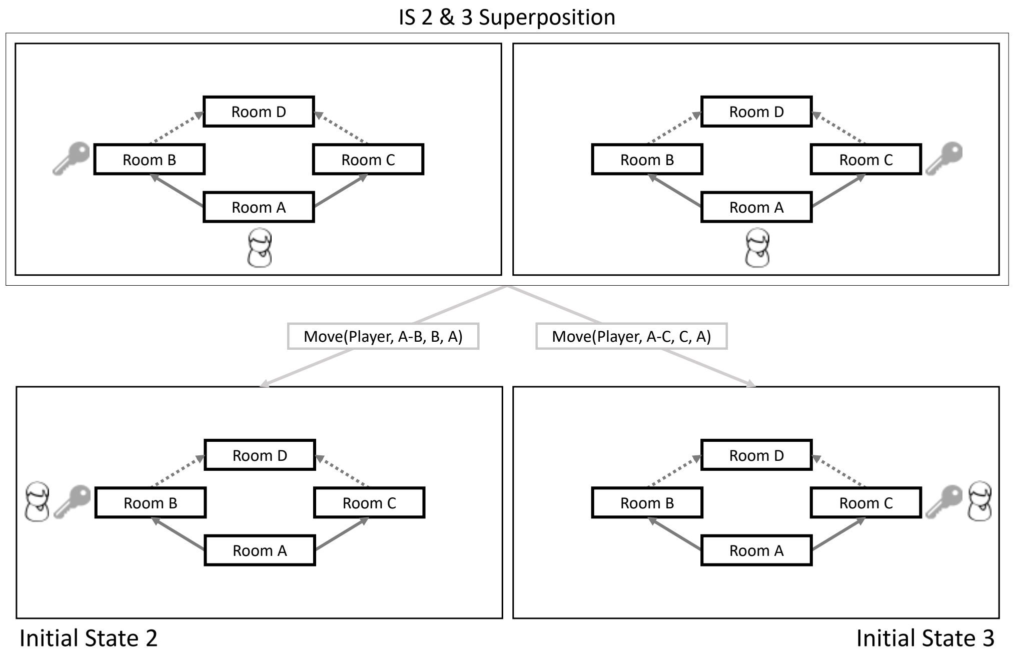


Figure 4.3: Event revision can find a plan for the player to get to Room D no matter what door they choose from Initial State 2 or 3 in the Key Domain problem presented in Figure 3.6.

compatible operator libraries given in the problem's set of operator libraries O. Two operators libraries are compatible, given a trajectory of player observations, if both operator libraries support the same series of observed actions and precondition-effect pairs of observed actions. When combined with event revision and forward planning, domain revision allows the framework to search the full space of VH_{player}, given an interactive narrative problem, world history, player character, and knowledge microtheory. This section details the process of domain revision, gives an additional pseudocode method to perform the operation, and gives an example in the Key Domain.

General domain revision is performed with the same three components that enable event revision, Algorithms 1, 2, and 3, along with an additional component shown in Algorithm 4. Domain revision is a meta-planning method that moves to other domain models in O when the current one fails. When a new domain is selected, planning with event revision fails if the domain is incompatible with the client's observations. To determine a new operator library's compatibility with the client's observations, the *Alternate* method is used which is presented as pseudocode in Algorithm 4. Given a world history trajectory and an alternate operator library, the *Alternate* method constructs a world history with the same events but takes place in a universe with a different set of world mechanics. Once a solution trajectory is generated by *Alternate* it is given to *Consistent* to check whether the alternate history is consistent with

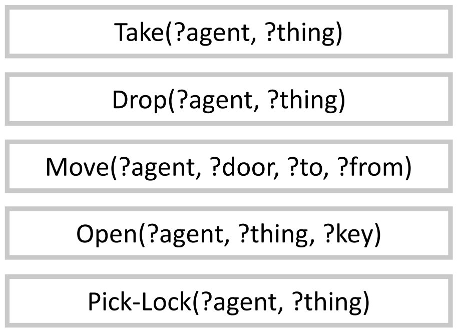

Take(?agent, ?thing)
Drop(?agent, ?thing)
Move(?agent, ?door, ?to, ?from)
Open(?agent, ?thing, ?key)
Pick-Lock(?agent, ?thing)

Figure 4.4: An alternate operator library for the Key Domain that includes the operator *Pick-Lock(?agent,?thing)*. The pick lock operator allows players to open locked doors without a key.

player observations. If so, it is then passed to *Remove* and used as the start of the event revision process which either returns a plan or results in no plan being found.

When a domain is not compatible with past player observations or fails to produce a plan, it can no longer be used by the system and is removed from the system's working version of O. The process exhaustively searches the space of domain models in O until a compatible plan is found or the set of libraries is exhausted. To illustrate this process, Figure 4.4 shows an alternate set of operators for the Key Domain that includes the operator *Pick-Lock* that allows the player to open locked doors without a key and Figure 4.5 shows an example of domain revision being used when the player moves to a new room without the Key object. If the Player chooses to move to Room B and leave the Key behind at Room A, the system will shift the Player to the alternate set of domain mechanics where they can perform the *Pick-Lock* action to open Door B-D and reach Room D. Since transitioning the player to the new domain model where the player can pick locked doors does not contradict what the player has observed in the world, the system can perform domain revision and find a new plan from the player's current state.

Between event and domain revision, the full space of VH_{player} can be explored by mediation when the player takes an exceptional action that cannot be solved by accommodation. However, using the mediation model of waiting until the player takes an exceptional action to search VH_{player} may lead to dead end branches that could have been avoided if the experience manager acted proactively. The earliest opportunity the experience manager has to act when performing perceptual experience management is whenever the player makes an observation that differentiates between two possible world histories that could be reached by event or domain revision. The next section presents superposition manipulation, a method of modeling the set of possible worlds the player exists in S_{player} and proactively choosing between possible worlds whenever an observation is made that collapses the superposition in order to maximize

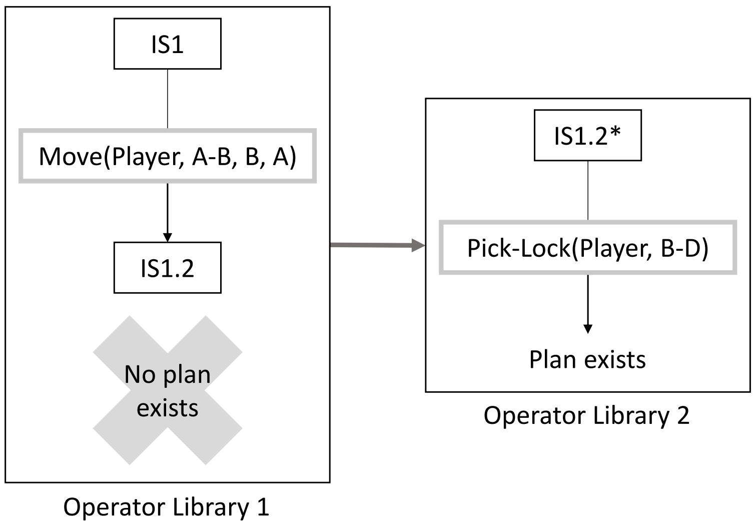


Figure 4.5: An example of general domain revision using the operator libraries presented in Figure 3.3a as Operator Library 1 and Figure 4.4 as Operator Library 2.

the probability that author constraints will be preserved and a goal state will be reached by the system.

4.5 Superposition Manipulation

Event and domain revision search the full space of a client's VH_{player} set when responding to an un-accommodatable exceptional action taken by the client. This is an improvement over previous experience managers that searched only a sub-set of the space that corresponds with the default t_{sim} simulation trajectory. However, by only responding to exceptional actions mediation systems miss out on an opportunity to maximize the probability that VH_{player} is never empty across all gameplay. This opportunity is missed every time the player makes a new observation that differentiates one possible world history from another. A client's set of possible world histories, H_{player}, forms the basis for how many and what kind of trajectories will exist in VH_{player}. The set of states reached by trajectories in H_{player} forms a superposition of possible world states the player may currently exist in. Any trajectory from a state in this superposition to a possible goal state will form part of a trajectory in VH_{player}. Whenever the player learns something new by observing the world that differentiates between the set of states reached by trajectories in H_{player}, the possible trajectories in H_{player} collapse, or shrink, in the direction of a true or false observation and so does the set of superposed current states. In order to maximize the probability that the story world reaches a goal state, the system should track and evaluate each collapsed superposition when the player makes an observation that differentiates between superposition states and decide which is more likely to lead to a goal state based on the state space and the player's potential actions. This process of choosing between possible world states based on a utility function that represents the probability of preserving author control by reaching a goal state when an observation is made by the player that collapses their set of possible worlds is called *superposition manipulation*.

This section defines a framework for performing superposition manipulation. The framework tracks superpositions of possible world states during gameplay. These world states correspond to possible current states that could be reached by performing event and domain revision on past events. This section presents a model of superposed possible states and describes how the model can be grown during gameplay when non-player characters have the opportunity to take unobserved actions. Superposition manipulation explicitly tracks superposed states so the system can evaluate them and make an informed decision between them whenever the superposition is collapsed by a user observation. This section also describes what a differentiating observation is and when it occurs. When an observation occurs, the system must choose one possible world set to make actual and show it to the player. When making this choice, the system should choose the set of possible worlds that maximizes the probability that a goal state will be reached. This determination relies on two components: the space under each possible state and how the player will behave in that space. However, these are both hard problems. This section concludes by presenting an extensible framework that allows different models that characterize state space and player choice to be inserted into superposition manipulation in order to guide its decision making.

4.5.1 Tracking Superpositions

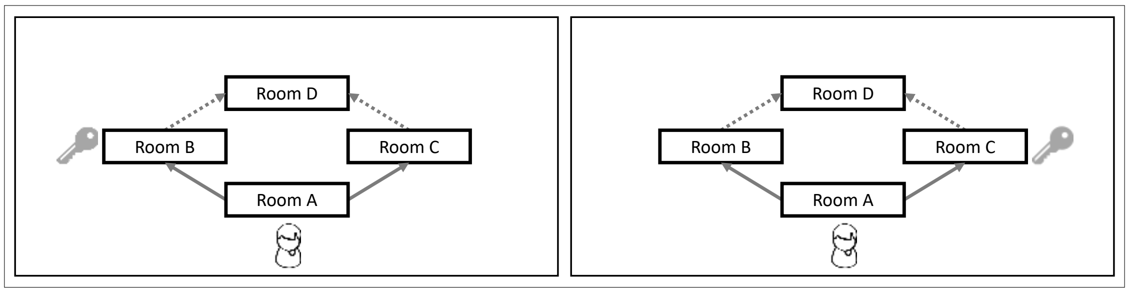
In order to evaluate the benefit of one superposition relative to another the system needs to track the possible states that could exist according to what the player has observed. Instead of tracking a single state at every node in the tree the framework discussed so far, a superposition manipulator tracks a set of unique states. This set is the collection of states, or superposition, at the current depth of the original experience manager tree that are perceptually equivalent to the player. The superposition representation is similar to initial state revision [50] (ISR) in the way it partitions state literals. ISR partitions the initial state into three sets, true, false, and unknown. The set of true literals are always true, the set of false literals are always false, but the set of undetermined literals can be decided by the system at runtime whether they will be true or false. This set of unknown facts is syntactic sugar for simultaneously specifying a state where the fact is true and another state where the fact is false. Superposition manipulation uses this same representation for specifying state literals that could be true or false according to what the player knows about the story world. Figure 4.6 shows a superposition of two states

True Set

Room A	Player
Room B	Key
Room C	Player is at Room A
Room D	A-B is open
Door A-B	A-C is open
Door A-C	B-D is locked
Door B-D	C-D is locked
Door C-D	

Undetermined Set

Key is at Room B
Key is at Room C



Initial States 2 & 3 Superposition

Figure 4.6: The two sets of ground atomic formulae that describe the superposition of Initial State 2 and 3 presented in Figure 3.12.

in the Key Domain with a corresponding true and undetermined sets. Everything about the two states are the same except for the placement of the Key which could either be at Room B or Room C, which is modeled with the undetermined set.

4.5.2 Growing Superpositions

In addition to modeling multiple states as a superposition, the system must also add new states to successor superpositions when non-player characters take unobserved actions. The first step is to find the set of unique actions enabled for the current character from each state in the set of states associated with the current experience manager tree node. Once the set of enabled actions is collected, it is divided into two subsets: observed and unobserved actions from the player's perspective according to the microtheory. Each of the observed actions is given its own outgoing edge from the current node. Each unobserved action is grouped together in a single outgoing edge. Observed actions function as they did in the original experience management tree. If the experience manager decides to traverse an edge that corresponds to one or more unobserved actions, each state in the parent superposition is updated by each enabled unobserved action to create a unique state in the successor superposition.

4.5.3 Collapsing Superpositions

The next component is a method of removing states from, or collapsing, the successor superposition when an observed action is taken or a new literal is observed. First, superpositions must be collapsed both before and after an observed action takes place. Before the action, every state in the parent superposition where the action's preconditions are not satisfied is removed. In terms of the true, false, unknown representation, any unknown literal that is required to be true or false in an observed action's precondition list must be moved from the unknown literal list to the true or false list. Once the initial collapse is finished, the successor superposition is created by generating the successor of each individual state in the set by applying the observed action. This is accomplished in terms of our representation by updating the true, false, and unknown lists according to the ground effects of the observed action.

The experience manager must also ensure that the player's observations are consistent across the set of states every time the player makes a new observation. If not, the manager must choose a perceptually equivalent subset to transition the player into. In terms of our representation, the player makes a new observation any time the knowledge microtheory predicts they will observe a literal in the unknown category. When this happens it is up to the experience manager whether to move that literal to the true or the false category. This choice of whether to collapse the player into the true or false subset of worlds when they observe an unknown literal is the central problem of superposition manipulation. To make an optimal decision the system requires a characterization of the state space under each subset as well as a model of what the player is likely to do in the future. The next section describes how superposition manipulation can choose states given these two components.

4.5.4 Evaluating Superpositions

When superposition sets are collapsed, a direction must be chosen, true or false, for the unobserved literals to collapse. This section describes a framework for calculating the utility of superposed states in order to choose between true or false successor superpositions when new observations are made by the player. Even if two nodes are part of an equal length path to a winning leaf node in the experience management tree, they may present different likelihoods of success for reaching a goal state leaf. Superposition manipulation views the utility of any node in the experience management tree as the probability that a winning leaf node will be reached during play from the current node. This probability can be calculated with two components: a probability distribution over outgoing edges that correspond to player choices and the fully expanded experience manager tree under each superposed state. Both of these components represent solutions to hard problems that are outside the scope of this document. However, superposition manipulation allows these components to be extensible, baseline methods can later be replaced with more robust solutions.

Predicting Player Choice

The more accurately player behavior can be predicted, the more accurately superposition management can predict what path will be taken under a particular experience management tree node. The current model is simply a uniformly random distribution over player choices, but this can be substituted for a more robust model informed by something like choice preference [68], goal recognition [7], and/or role assignment [10]. A more robust model would update the distribution over player choice edges to more accurately represent what behavior the player will exhibit at runtime without altering the structure of the rest of the superposition manipulation framework.

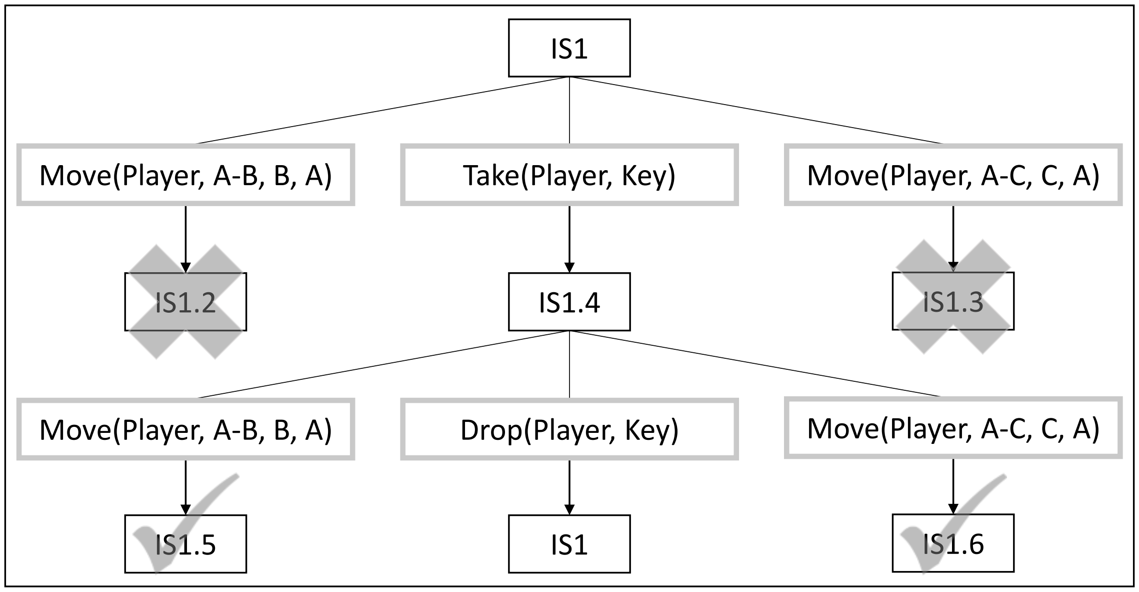
Expanding State Space

The second component of the utility calculation is the structure of the state space beneath each state in the superposition. A fully explored state space will result in a collection of branches with leaf nodes that are either a winning goal state or a losing dead end state. The number of wins and losses below each state along with the probability of reaching wins and losses given player behavior allows superposition manipulation to calculate the utility of choosing a particular state in terms of the probability of reaching a goal state.

Superposition Utility

Given a node, its fully expanded subtree, and a function that gives a likelihood distribution over each set of player choice options in the subtree, the utility of the node can be calculated as the probability that a winning node will be reached given optimal choices performed by the experience manager in the subtree. For example, Figure 4.7 pictures two superpositions in the Key Domain that correspond to the states pictured in Figure 3.12. The first tree is from a single state where the player has three options: two lead to dead ends and the third leads to a second choice. From the successor state that doesn't lead to a dead end the player has another three choice options: two lead to future goal states and one leads back to the original state. Together, this tree represents a 50% win rate assuming the player makes uniformly random decisions. The second tree is from two superposed states where no matter what the player chooses there is a path to a goal state. This tree represents a 100% win rate and should be chosen by the superposition manipulator.

Superposition manipulation is the final component of perceptual experience management. The next section uses a problem set in the Spy Domain, which is more complex than the Key



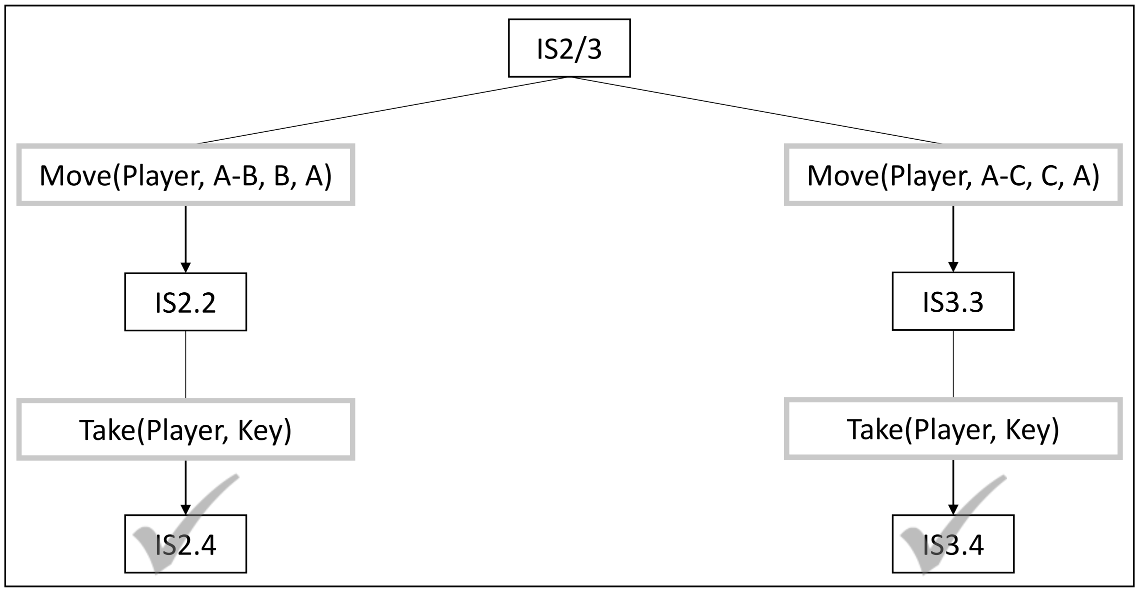


Figure 4.7: The two world trees produced by event revision on the Key Domain interactive narrative problem presented in Figure 3.6 starting from the superpositions presented in Figure 3.12. A goal will be reached in 50% of the branches in the Superposition 1 tree and 100% of the branches in the Superposition 2/3 tree.

Domain and contains system controlled characters, to illustrate the mechanics of event revision, domain revision, and superposition manipulation.

4.6 Spy Domain Example

This chapter has described perceptual experience management with a running example set in the Key Domain. However, the Key Domain has only a single agent and a limited state space. This section gives a larger example for event revision, domain revision, and superposition manipulation using the Spy Domain. The Spy Domain's PDDL domain and problem is given in Appendix A.4. This section begins with a description of the domain and problem, then describes a situation during gameplay in the domain in which event revision, domain revision, and superposition manipulation can be performed by a perceptual experience manager.

4.6.1 The Spy Domain

The Spy Domain is meant to represent a sneaking game and is modeled as a cross of *GoldenEye 007* [45] and *Metal Gear* [21]. The *Spy* game is an example world where the player, as a spy named Snake, must foil the final attempt of the computer-controlled antagonist, the Boss, to bring a weaponized satellite online. The confrontation takes place on a satellite dish antenna cradle with five discrete locations where the Snake and Boss can interact: the Elevator Room, Gear Room, Left and Right Walkways, and the Platform. The locations are connected by doors that can only be traversed in one direction. The initial world layout is pictured in Figure 4.9 using icons for world objects pictured in Figure 4.8. In the initial state figure, locations are labeled rectangles and doors are arrows. The doors can only be traversed in the direction arrows are facing. Snake begins the game in the Elevator Room. Her job is to disable the satellite dish's alignment mechanism in the Gear Room and eliminate the Boss. The Boss is trying to send instructions from his phone to the satellite by linking the phone to one of four computer terminals on the cradle. The domain author wants the Boss to build and set a trap to be disabled by the player before a final confrontation between the two on the Platform. These authorial constraints are coded as conditions in the PDDL goal state.

Snake starts off in the Elevator Room and the Boss begins in the Gear Room. Snake has a pistol (PP7) an explosive (C4) and a detonator for the explosive. The Boss has a laser rifle and a trip-wire for building a trap, and a phone that can be linked to the satellite through an activated computer terminal. There is a computer terminal on the Platform, Left and Right walkways, and in the Gear Room. Any of these four computer terminals can be activated by the Boss and used to send information to the satellite from his phone. This initial world configuration is shown in Figure 4.9. The game progresses by alternating between allowing the Boss and Snake to take an action that updates state information. The Boss is controlled by plots

(a) The Boss

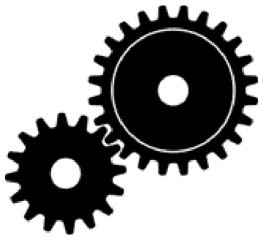
(b) Snake

(c) PC Terminal

(d) Activated PC

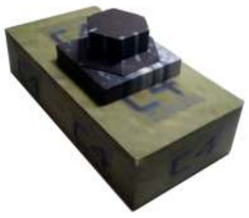
(e) Phone

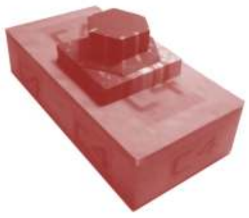
(f) Linked Phone

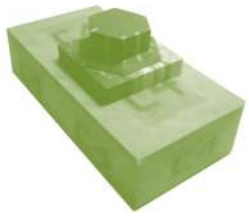

(g) Gears

(h) Trap

(i) PP7


(j) C4


(k) Red C4


(l) Green C4

(m) Detonator

(n) Laser Rifle

(o) Trip Wire

Figure 4.8: A key of images that depict PDDL objects in the Spy domain.

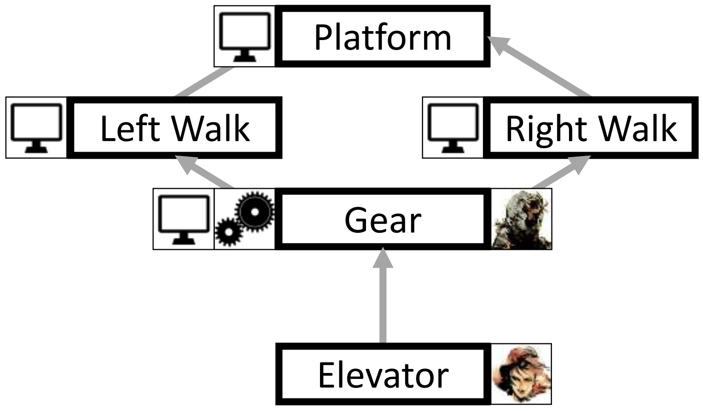


Figure 4.9: A diagram of the initial configuration of the *Spy* world given as the initial state in Appendix A.4.2. Rectangles are rooms, grey arrows are one-way connections between rooms, and squares are world objects in each room.

generated by the system's planner and connected through the experience manager's mediator. Snake is controlled by a player. The game continues until a goal state is reached or the author's constraints are broken.

4.6.2 Event Revision

In the Spy Domain, the NPC Boss character gets to take the first turn. The Boss is controlled by the branching plans created by mediation. According to the mediator's plan the Boss begins gameplay by moving to the Right Walkway. The player as Snake decides to move to the Gear Room. The Boss makes a trap for Snake by combining his laser rifle with a trip wire. Snake places her C4 on the Gears object in the Gear Room. The Boss places his trap at the Right Walkway. Snake uses the detonator to destroy the Gears with the C4. Finally, Boss moves from the Right Walkway to the Platform. At this point, the player must decide whether she will move to the Left Walkway or the Right Walkway. If the player moves to the Right Walkway, the action is constituent. Snake can disable the trap set by the Boss, move to the Platform, and the final showdown between Snake and Boss can commence.

However, if the player decides to move to the Left Walkway, the action is exceptional. No matter what actions are taken from that point on, the goal state cannot be accomplished without a perceptual simulation. If the experience manager uses event revision it can replace past events not observed by the player. Event revision identifies all of the actions the Boss has taken in the story world as unobserved by the player and available to replace. It will be able to find an alternative course of events where the Boss moved to the Left Walkway, built, and set his trap there instead of the Right Walkway. Using this alternate set of events, event revision

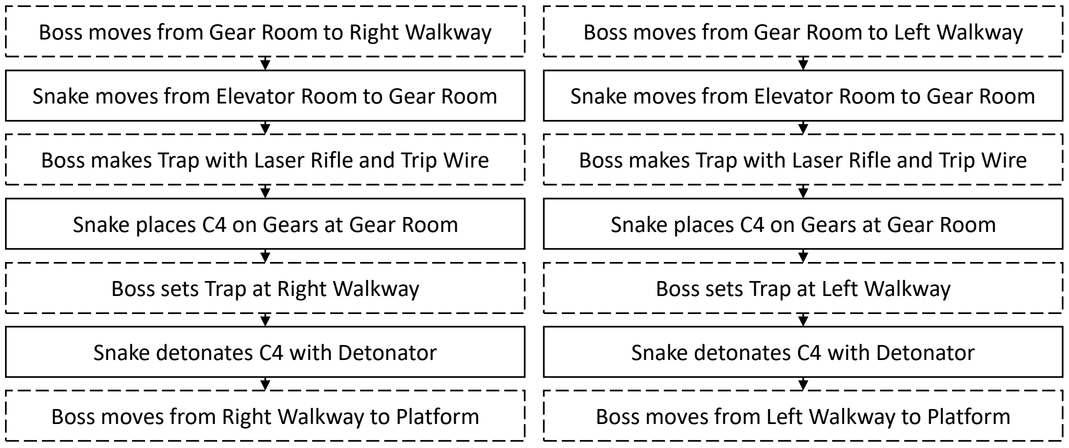

Boss moves from Gear Room to Right Walkway
Snake moves from Elevator Room to Gear Room
Boss makes Trap with Laser Rifle and Trip Wire
Snake places C4 on Gears at Gear Room
Boss sets Trap at Right Walkway
Snake detonates C4 with Detonator
Boss moves from Right Walkway to Platform
(a)
Boss moves from Gear Room to Left Walkway
Snake moves from Elevator Room to Gear Room
Boss makes Trap with Laser Rifle and Trip Wire
Snake places C4 on Gears at Gear Room
Boss sets Trap at Left Walkway
Snake detonates C4 with Detonator
Boss moves from Left Walkway to Platform
(b)

Figure 4.10: Two perceptually equivalent action trajectories in the *Spy* domain. The player observes any action performed in the room where they are located. Actions unobserved by the player have a dotted border. From the player's perspective, either of these sequences of events could have taken place given player actions of moving to the Gear Room, placing C4 on the gears, and detonating the C4.

allows the Spy Domain to continue on to a goal state when it otherwise would reach a dead end. Figure 4.10 shows these two action trajectories in the Spy Domain's story world, both consistent with the player's knowledge of the story world. Event revision works by searching alternate trajectories consistent with player knowledge by replacing unobserved actions with perceptually equivalent alternatives.

4.6.3 Domain Revision

There are two operator libraries for the Spy Domain that have a single difference: the effects of the *detonate-explosive* operator. In the Spy Domain, the C4 object that Snake uses to destroy the gears in the Gear Room has a light that can be toggled between red and green. The *toggle-red* and *toggle-green* actions is used to change the light back and forth between the two colors. The *detonate-explosive* destroys the C4 and anything the C4 is connected to if the C4 is armed. In one domain library the C4 is armed when its light is red. In the other domain library the C4 is armed when its light is green. These two operator libraries can be navigated between by domain revision if player actions produce a dead end in order to give the player a second shot at reaching the goal state.

One of the goals of the Spy problem is for Snake to destroy the antennae gears so it cannot communicate with a satellite in orbit. However, Snake only has the single C4 charge and it can be placed on any object in the game world. If Snake moves to the Gear Room, places the C4 on the computer terminal, and detonates the C4 with her watch, the detonation action will be flagged as an exceptional action but no alternate plan will be found by accommodation because there is no alternative way for the gears to be destroyed. If domain revision is enabled, it can shift to the domain library where a red light equals an armed C4 device that can be detonated. In this alternate universe, the C4 will only detonate if the light is green instead of the original red. This transition between alternate domain mechanics delays the detonation of the C4 and gives the player a second opportunity to destroy the Gears by taking the C4, toggling it to green, placing it on the Gears, and detonating the C4.

4.6.4 Superposition Manipulation

This final example shows how superpositions grow and collapse based on player observations during gameplay. As NPCs take unobserved actions, the set of current states the player could exist in grows. As the player observes whether literals are true or false, the superposition collapses. This section gives an example of these mechanics in terms of the Spy Domain.

Growing Superpositions

Every time an NPC has the opportunity to take two or more unobserved actions, the set of possible worlds the player could exist in grows by one for each unobserved action the NPC could take. For example, from the initial state the Boss has four enabled actions: Move to Left Walkway, Move to Right Walkway, Make Trap, and Use Terminal1. Each of these actions are unobserved by the player according to the microtheory of player knowledge. If the player moves to the Gear Room the next turn and finds the Boss is not there, it means the Boss must either have moved to the Right Walkway or Left Walkway. From each of these two positions the Boss has mirrored action possibilities that are all unobserved as the player destroys the gears in the Gear Room. The Boss can make a trap, place it at his current location, and move to the Platform. Alternatively, the Boss can make a trap, activate the computer terminal, and link his phone to the terminal. Finally, the Boss can make a trap, activate the computer terminal, and place the trap at his current location. These three sequences of unobserved actions can be taken at each of the Right or Left Walkways available in the initial superposition. This will result in the six state superposition given in Figure 4.11.

Collapsing Superpositions

Whenever the player makes a new observation of a previously unknown, superposed state literal superposition manipulation has the opportunity to decide whether the literal will be observed as true or false. If the player moves from the Gear Room to the Right Walkway from the superposition shown in Figure 4.11 one unknown state literal the player will observe is whether

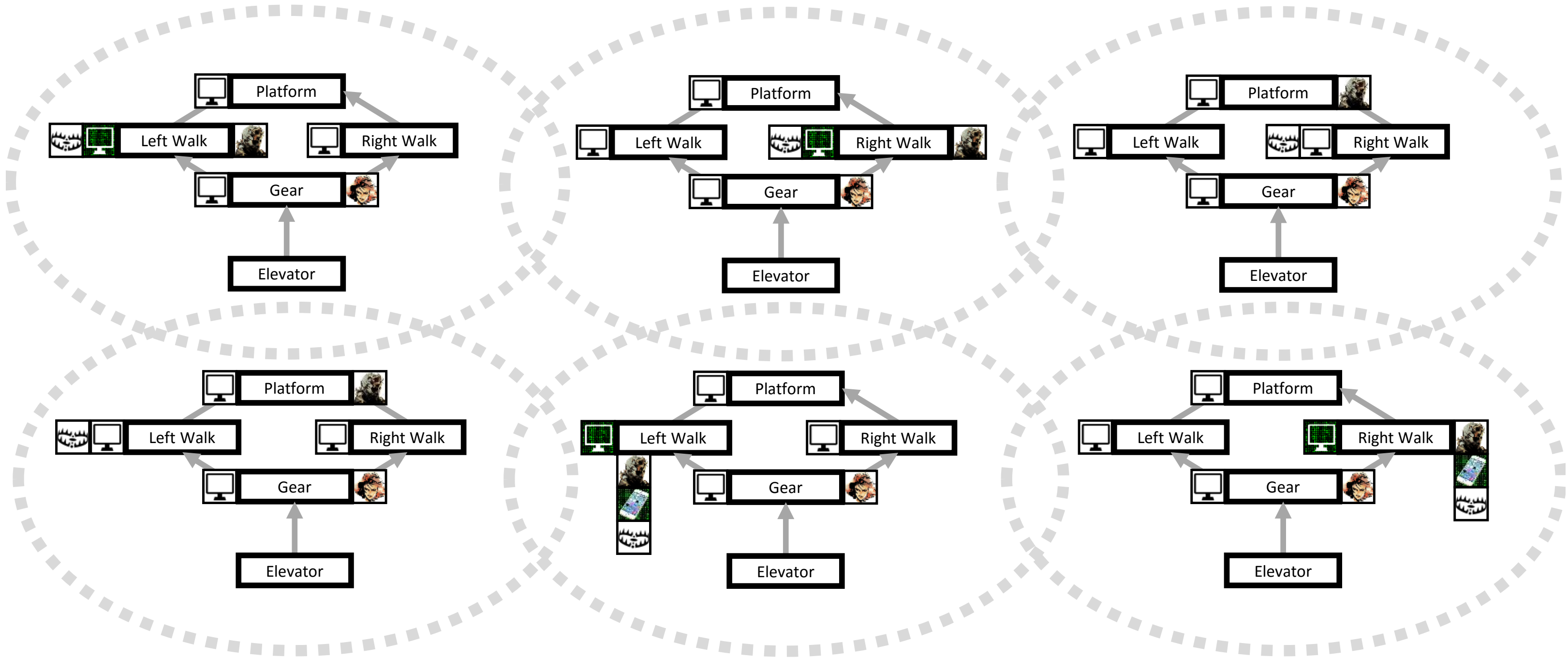
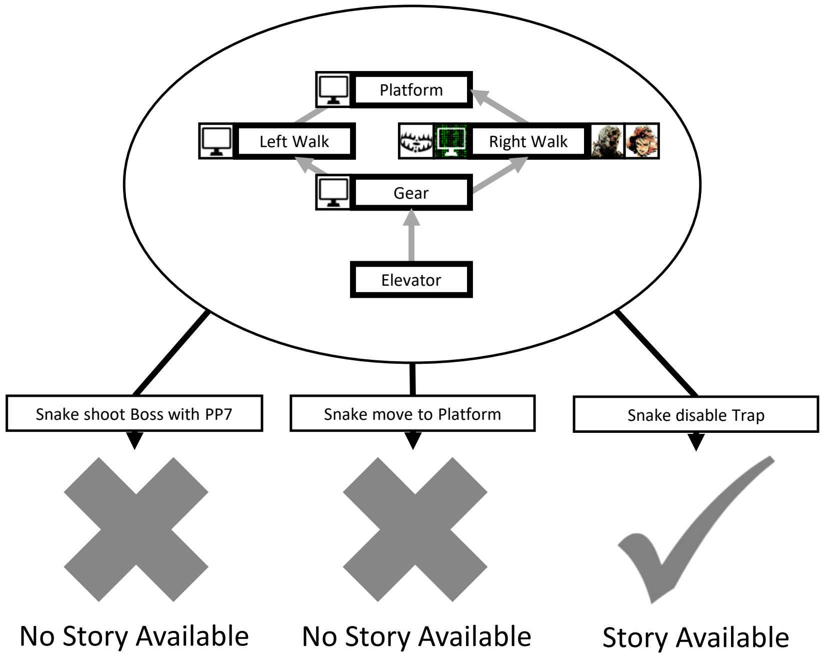


Figure 4.11: A set of six superposed states, each consistent with what the player knows about the world in the *Spy* domain after they perform the actions: move from Elevator to Gear Room, place C4 on Gears, detonate C4.

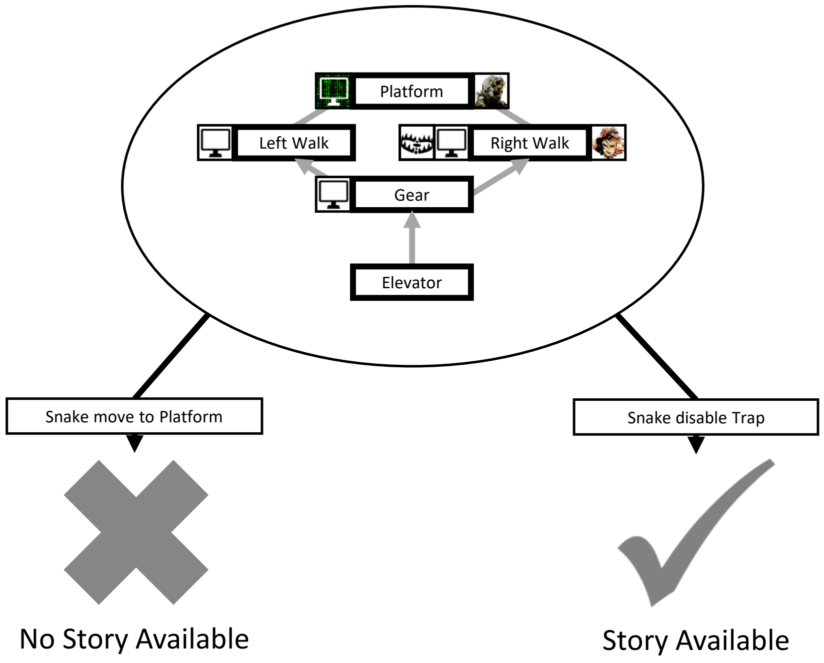
or not the Boss is currently at the Right Walkway. The experience manager will need to choose whether to show the player the collection of states where the Boss is at the Right Walkway or the collection of states where the Boss is not at the Right Walkway. In order to make this decision, the experience manager should determine which collection of states offers the highest probability the story world will reach a goal state. Figure 4.12 shows two states from the Figure 4.11 superposition along with possible outcomes of enabled player actions. Two of the three possible player actions from the state in Figure 4.12a lead to dead ends and a single action leads to a goal state while one of the two possible player actions from the state in Figure 4.12b leads to a dead end and a single action still leads to a goal state. The experience manager should choose the second of the two states because it offers a 50% as opposed to a 33.3% chance at reaching a goal state.

4.7 Summary

This chapter describes a framework for perceptual experience management in a plan-based mediation system. The framework shifts the player between alternate possible world histories and mechanics they could exist in. The chapter began by introducing mediation, then describing event revision, domain revision, and superposition manipulation in the context of a plan-based mediator. The chapter ends by illustrating perceptual experience management's mechanics with an example set in the Spy Domain. The next chapter describes an implementation of a perceptual experience manager called the General Mediation Engine.



(a) A state with a 1:2 ratio of good branches to bad branches.



(b) A state with a 1:1 ratio of good branches to bad branches.

Figure 4.12: Two states and their outgoing player actions in the *Spy* domain. Assuming the player makes uniformly random decisions, the state in Figure 4.10a has a 1/3 chance that authorial constraints hold and Figure 4.10b has a 1/2 chance.

Chapter 5

The General Mediation Engine

This chapter presents structural and implementation details of the General Mediation Engine (GME). GME is an implementation of the perceptual experience management framework presented in Chapter 4. GME is a plan-based experience manager that communicates with an external planner through PDDL and automatically configures a gameplay interface for players based on its underlying world state. Section 5.1 provides an overview of the GME backend architecture and its two interfaces, one text and one visual 2D. Section 5.2 describes in detail the implementation of GME's perceptual experience management backend, including its mediation, event revision, domain revision, and superposition manipulation components. Section 5.3 describes the implementation of each GME interface pipeline. Finally, Section 5.4 concludes the chapter and discusses future work for the system.

5.1 GME Overview

The General Mediation Engine is a perceptual experience manager and interface generator. The main body of GME is implemented in C# [55]. A diagram of GME's architecture is given in Figure 5.1. The GME framework is planner independent, it communicates with planning systems through modifying PDDL files, querying an external planner with command line calls, and reading planner output from file. The current system supports versions of the Fast Downward [19] and Glaive [63] planners, but other planners can easily be attached. The system is capable of performing perceptual experience management operations while treating the external planning operation as a black box. The system uses a procedural content generation (PCG) pipeline to display its interactive narrative trees to clients. GME currently has four PCG interface variations across two types of media: text and visual 2D. GME has three text interfaces, a local console application, a web-based console application, and a web-based Choose Your Own Adventure application. The final interface is a 2D visualizer built with the Unity game

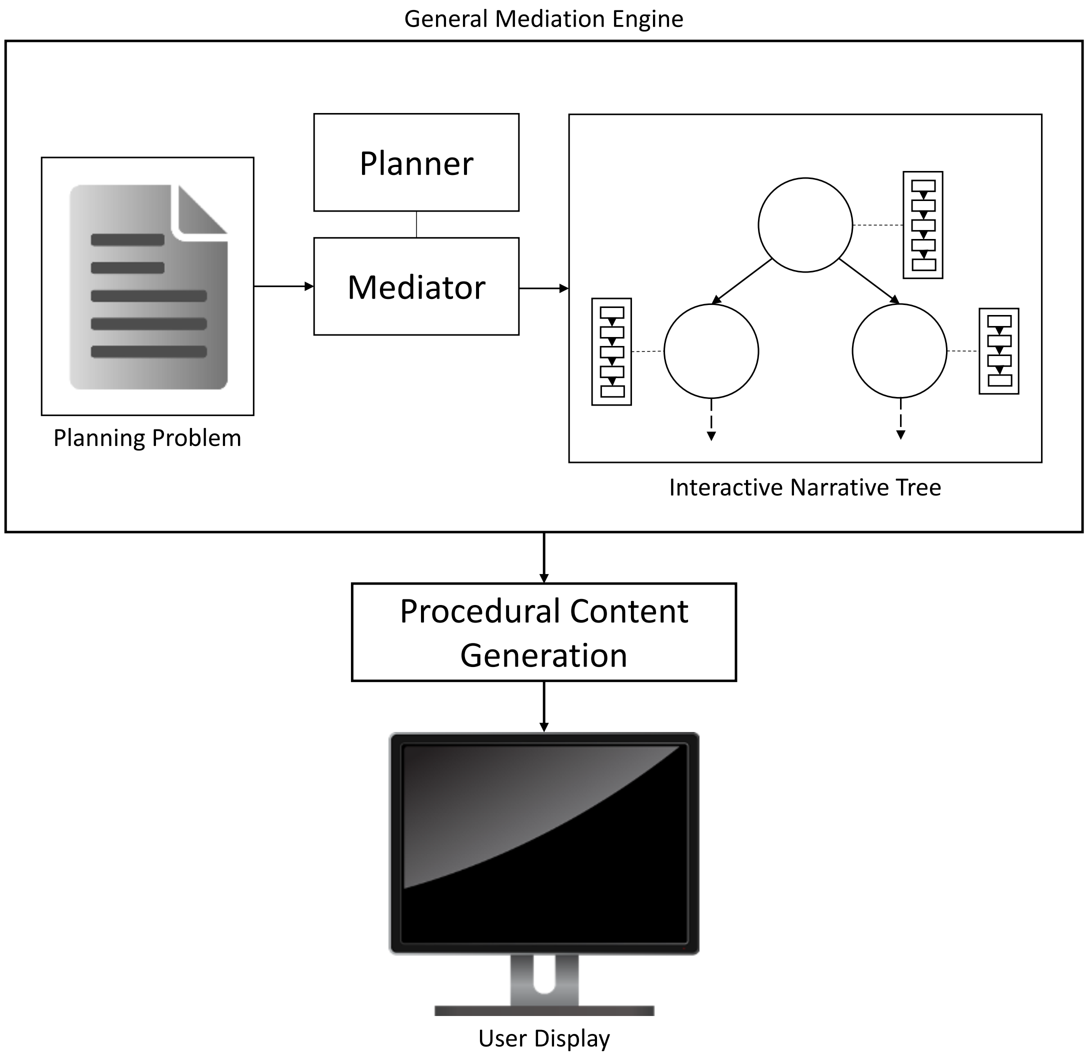


Figure 5.1: The General Mediation Engine architecture. A planning problem is given as input to a mediator which has access to a planner. The mediator builds an interactive narrative tree which is visualized to a player through a PCG pipeline. Actions the user takes through the interface influence how the mediator expands the interactive narrative tree.

engine. Each of these interfaces uses a library of assets, be it text templates or Unity prefabs, to dynamically configure a game world based on GME's current world state and convey player actions from the interface to the underlying framework.

5.2 GME Backend

The first component of the General Mediation Engine is its backend. The GME backend reads and writes planning domains and problems to and from file, constructs data structures that represent domain and problem objects, sends the domain and problems to an external planner, receives solution plans from the external planner, constructs trees that represent paths through the narrative world defined by the domain and problem, chooses NPC actions based on the current plan, takes input from a player client, constructs a model of player knowledge, identifies exceptional player actions, queries for plan updates when exceptional action occur, and constructs a perceptual interactive narrative simulation by utilizing event and domain revision when the planner fails to find a solution.

5.2.1 PDDL Tools

The first set of components for GME's backend machinery allow GME to model and generate solutions to planning problems. In order to do this, GME reads and writes PDDL domain and problem files, models data structures that represent planning problem information, queries and receives plans from a planner. GME's suite of tools begins with its IO package. The IO package contains two classes, a parser and a writer. The parser reads planning domains, problems, and plans in from files and returns internal GME data structures that model the external information. The writer class contains methods to write internal planning problem and domain data structures to external PDDL files, visualize mediation trees with hyperlinked HTML files, and a method to write test statistics to disk in the form of a comma separated values document. In order to store and manipulate PDDL information, GME contains a suite of data structures that model PDDL components. GME's Plan Tools package contains data structures for axioms, dependencies, domains, flaws, intentions, objects, operators, plans, predicates, problems, states, and terms. Figure 5.2 shows a diagram of interfaces these data structures implement. Together, these implemented interfaces form the building blocks from which GME builds interactive narrative trees through mediation.

The GME object with the least dependencies is the PDDL domain, represented by the *IDomain* interface. All domain objects have a unique name, stored in its *Name* property. The primary responsibility of GME's domain object is to store the set of action operators for a particular domain in its *Operators* property. This is implemented as a list of *IOperator* objects that can be publicly written and read. GME supports typed objects which help reduce the

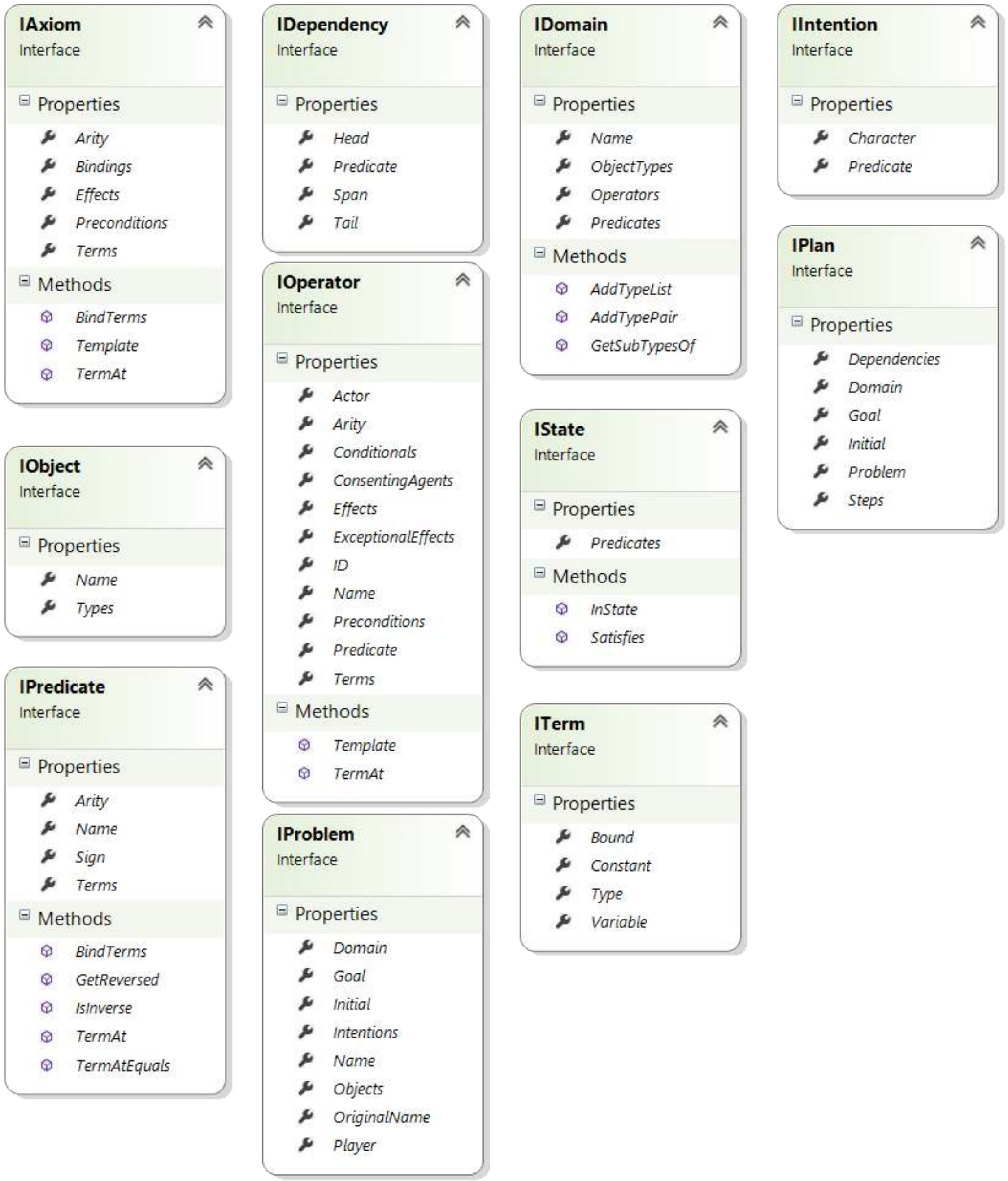


Figure 5.2: Diagram of interfaces for GME's PDDL data structures. The diagram is generated by Microsoft Visual Studio 2015. GME models axioms, dependencies, domains, intentions, objects, operators, plans, predicates, problems, states, and terms. When implemented, these interfaces form the basic building blocks from which interactive narrative trees are constructed.

space of applicable actions. Object types can be publicly retrieved as strings in a list called *ObjectTypes*. Types can be added through the *AddTypePair* and *AddTypeList* methods which take as input a single or multiple subtypes and a supertype. The *GetSubTypesOf* returns a list of subtypes for a given type. Finally, GME stores predicate information for a domain in an *IPredicate* list called *Predicates*. This list gives the set of predicates that can appear in operator precondition or effect lists and bound to world objects in order to create state literals.

Domains store a set of *IOperator*s that model an action template library. Each operator has a name and a list of *ITerms* that model terms. Together, the name and terms can be combined into a *IPredicate* to signify the operator or a fully ground action. For example, the Key Domain's *move-location* action in Appendix A.3.1 can be written with the predicate *(move-location ?mover ?to)* and the ground action of the *player* moving to location *B* in Key Domain Problem 01 shown in Appendix A.3.2 can be written with the literal *(move-location player B)*. This operator shorthand can be accessed in each operator's *Predicate* property. Operators store preconditions and effects as lists of *IPredicates* in their *Preconditions* and *Effects* properties. Operators may also have a list of *IAxiom*s that represent conditional effects. Intentional operators store a list of *ITerms* that represent the characters that must consent to the action. Operators also store bookkeeping information like a unique integer ID when they are instantiated into an action, what effects are exceptional if any, and an integer that represents the operator's arity, or how many terms the operator has. Finally, operators can return the term at a particular location in its list of terms and also a fully ground action can return a blank copy of the operator template it was instantiated from.

All domains are paired with at least one problem, represented by the *IProblem* interface. Like domains, problems also have a unique name stored by the *Name* property. This name may be altered in clones created by event revision, so problems also retain their original name in the *OriginalName* property. In addition to their own name, problems also store the name of the domain they are paired with in the *Domain* property and the name of the player's character in the *Player* property. The primary responsibility of the problem is to store the initial and goal states in its *Initial* and *Goal* properties, each a list of fully ground *IPredicate*s. Finally, problems store a list of *IObject*s in the *Objects* property that represent world objects and possibly a list of *IIntention*s in the *Intentions* property for character intentions in intentional domains.

Objects that implement the *IPredicate* interface allow operators to model precondition and effect lists. Predicates are also used by the *IState* interface to model state literals. All predicates have a unique name stored as a string. Predicates also have a list of *ITerm* objects that model bound or unbound variables, an *Arity* property that exposes the number of terms the predicate contains, and a *Sign* property that stores whether a predicate instance is true or false. The *IPredicate* interface also specifies a number of methods for simple operations on predicates and literals. Predicates can be bound with the *BindTerms* method that given a dictionary of variable-constant pairs. The *IsInverse* method checks whether a literal is the same as another, aside from their signs. The *GetReversed* method makes a copy of the literal object, flips its sign, and returns the inverse copy. Finally, the *TermAt* takes an integer and returns the term at that location and the *TermAtEquals* checks to see whether a given term matches the term at an integer's position in the predicate.

Predicates are built with objects that implement the *ITerm* interface. Term objects represent a variable that can be bound to an object. *ITerms* consist of strings that represent a variable, a constant, and a type, and a boolean value that records whether the term's variable has been bound to a constant or not. A similarly small interface is the *IObject* interface that contains a string for the object's name and a list of strings for the object's possible types. The smallest interface in the PDDL toolkit is the *IIntention*, which specifies a string to store the character's name and an *IPredicate* to store the literal the character intends. The *IDependency* interface is implemented by dependencies and causal links. It specifies a range in a plan over which a literal holds true. The head and tail are *IOperator*s where the span begins and ends, the predicate is the literal, and the *Span* property is a list of actions the link encompasses. A final small interface, the *IAxiom* object, specifies special conditional effects for operators.

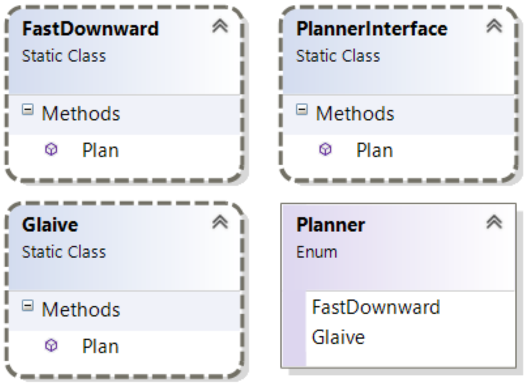


Figure 5.3: Diagram of GME's multiple planner system. The diagram is generated by Microsoft Visual Studio 2015. This diagram shows the system with two available planning interfaces: Fast Downward and Glaive.

All of these objects build up to states and plans. The *IState* object stores the conjunction of literals that are true in a certain world instance. Its methods, *InState* and *Satisfies*, will decide whether a single or set of literals match the state instance. Finally, the *IPlan* interface is implemented by plan objects. Plans have an initial and goal state, a problem object, a domain object, a set of dependencies, and a set of steps. The *Steps* property is an ordered list of fully ground operator objects that describe a set of actions that will transform the initial state into the goal state. GME generates plans by querying external planners with a PDDL domain and problem file. It does this by listing available planners in an enumeration, querying a general planner interface with a domain, problem, and planner enumeration, then the general interface passes the request to the particular planner's class where information is contained on how and where the planner can be reached and where to read the feedback. Once the planner is finished, the planning class reads in a plan object from the planner's output file and returns it as a plan object. A diagram of the planning system is given in Figure 5.3.

5.2.2 Mediation Trees

The second set of components for GME's backend machinery allow GME to build baseline mediation trees using accommodation. In order to do this, GME models the state space defined by a planning problem as a tree, takes action for NPC actors based on the current plan, identifies consistent, constituent, and exceptional player actions, and performs accommodation when necessary. A diagram of GME's mediation package is pictured in Figure 5.4. GME's mediation package begins with the *MediationTree* class. MediationTree stores information for a mediation tree and provides methods to external processes that allow the tree to be expanded. The class contains three private fields that allow it to track tree information: a *data* field that stores a *MediationTreeData* data structure, a *path* field that stores a path to the stored tree data on the computer, and a *root* field that stores a *MediationTreeNode* object that models the root of the mediation tree.

MediationTree contains a number of properties that expose information about the mediation tree contained in the tree object's private fields, including the MediationTreeData data structure. MediationTree also contains methods that allow mediation trees to be created and manipulated. Its constructor takes as input a domain and problem object and a disk path and initializes a new mediation tree with a single node, the root, saved as a binary file at the location specified by the path. The MediationTree class uses the *BinarySerializer* class to serialize and deserialize tree nodes and edges to and from binary files on disk. This process allows mediation tree data to be created and stored in long term memory for testing purposes and quick tree expansion of cached nodes. Given a tree node, MediationTree can determine its outgoing edges with the *GetOutgoingEdges* method which takes as input a node and an actor and returns a list of outgoing *MediationTreeEdge*s that correspond to enabled actions from the parent state. Given an outgoing edge, MediationTree can determine its child node with the *GetNode* method which takes as input a domain, problem, and incoming tree edge and returns a *MediationTreeNode* which it creates with the *CreateNode* method if it does not already exist in memory. Together, *GetOutgoingEdges* and *GetNode* can be used to expand a tree from its root down.

MediationTree contains a number of other methods that help the expansion and maintenance

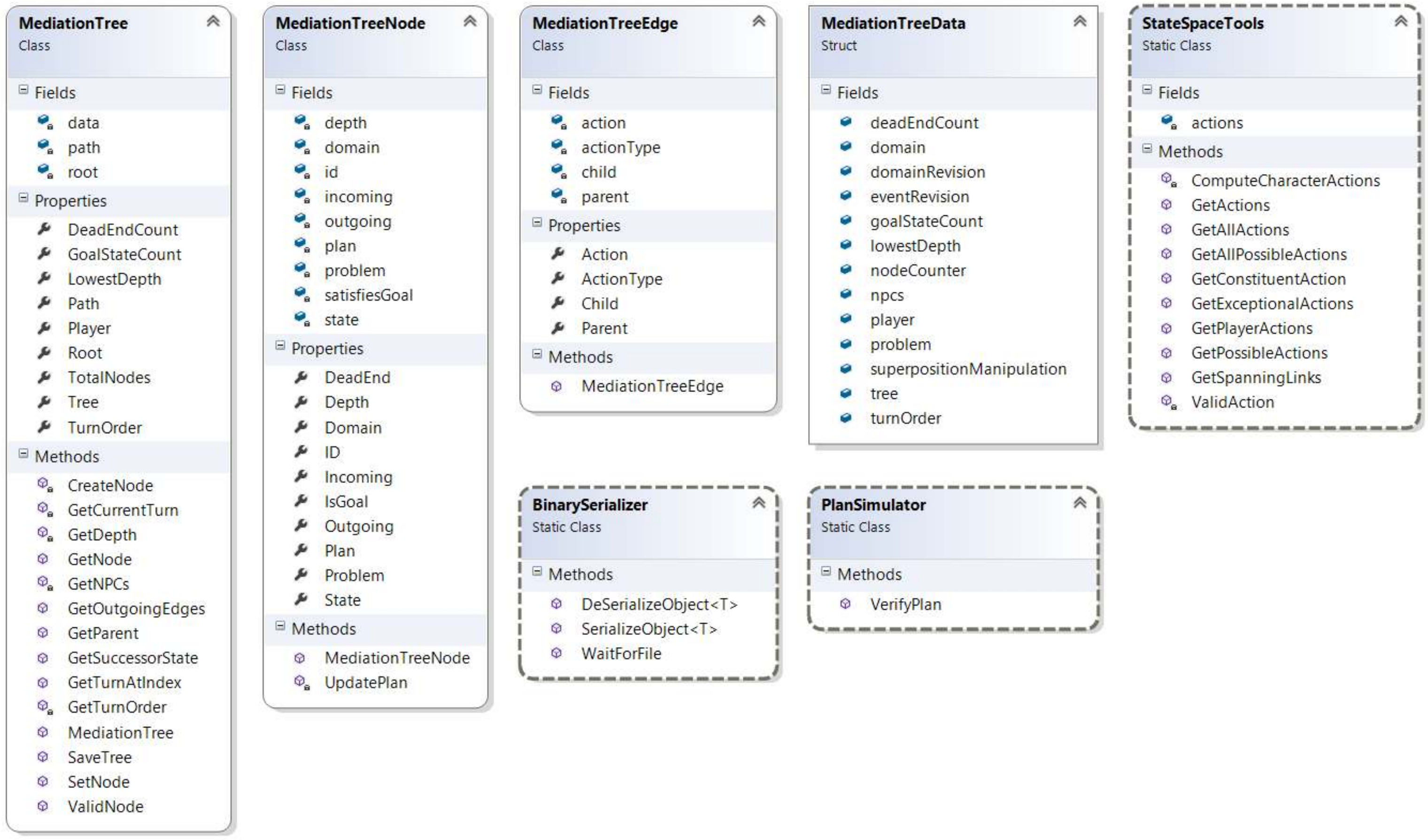


Figure 5.4: Diagram of classes that contribute to GME's mediation trees. The diagram is generated by Microsoft Visual Studio 2015. The MediationTree class generates the trees, the Node and Edge classes represent tree objects, the Data struct stores general information about the data structure, the BinarySerializer reads and writes tree information to disk, the PlanSimulator checks whether plans reach the goal state, and the StateSpaceTools class provides.

of mediation trees. *GetCurrentTurn* takes as input a tree node and returns the character who gets to act next based on the *TurnOrder* list and the depth of the given node in the tree. The *GetDepth* method exposes the depth of a given node. *GetNPCs* returns a list of names of NPC characters. *GetParent* takes as input a node and returns its parent. *GetSuccessorState* takes as input a tree edge and returns the state that results when the edge's action is applied to the parent node's state. *GetTurnAtIndex* takes a depth index and returns the character whose turn it is to act at that layer. *GetTurnOrder* exposes the *TurnOrder* property. The *SaveTree* method saves the *data* field that stores tree statistics to disk. The *SetNode* method takes as input a *MediationTreeNode*, serializes it, and saves it to disk. Finally, the *ValidNode* method checks whether a given node ID has been expanded and saved to disk or not.

The MediationTree class operates on two data storage classes and a high-level tree data structure. The first of these is the *MediationTreeNode* class that stores information and performs operations for nodes in the mediation tree. Nodes have a number of private fields that store information about the node's depth, the domain and problem used to create the node, the node's unique ID, its incoming and outgoing edges, the system's plan, whether the node satisfies the problem's goal state, and the current world state. *MediationTreeNode*s expose their fields through a set of properties and calculate the information in a constructor. The one operation nodes perform is updating the current plan with the *UpdatePlan* method. If the incoming edge is consistent, the method does nothing. If the incoming edge is constituent, the method removes the edge's action from the current plan. If the incoming edge is exceptional, the method creates a new problem with the current state as the initial state and queries the planner interface for a plan using the new PDDL problem object. If no plan is returned, the node marks itself as a dead end.

The second class MediationTree operates on is *MediationTreeEdge* which stores information on edges in the mediation tree. Edges have a number of private fields that store information about the edge's action, whether the action is consistent, constituent, or exceptional, the child node the edge points to, and the parent node the edge points from. Each of these fields are exposed as a property and the information is set in the class constructor. The final component of the mediation tree data is the *MediationTreeData* data structure which stores high-level information about the mediation tree. The data structure stores how many dead end nodes have been encountered during expansion, the base domain and problem objects, the player's character name, the tree object, the turn order, whether domain revision, event revision, or superposition manipulation is enabled, how many goal states have been encountered during expansion, the lowest depth encountered during expansion, the number of nodes expanded, and the list of NPC names. Together, these three classes and one data structure allow GME to model and expand mediation trees.

In addition to its mediation tree suite, GME also uses a tool set for computing enabled actions and a simulator for determining exceptional actions. The *StateSpaceTools* class allows GME to compute character actions, determine which are enabled in a given state, and which enabled actions are constituent, consistent, and exceptional. *StateSpaceTools* has a single field, the actions dictionary which maps character names to the set of all possible actions the characters can perform in a given domain. This dictionary is filled at the start of each session by the *ComputeCharacterActions* method. The rest of the methods determine which actions for a character are enabled for a particular state and which of these are constituent, consistent, or exceptional. To determine whether actions are exceptional, dependencies are drawn between actions in the current plan. A dependency is a directed edge from an action that makes a literal true to any other action that uses the literal as a precondition. If an action breaks any of these dependencies, it is exceptional. If the action is in the current plan and if the *PlanSimulator* reaches a goal state after removing the action from the plan, then the action is constituent. If the action is neither exceptional nor constituent, it is consistent. With these components GME can generate accommodative mediation trees. The next section describes GME's knowledge microtheory, which is the component that enables perceptual experience management.

5.2.3 Knowledge Microtheory

The class that enables GME's ability to track and manipulate unobserved aspects of the story world is the knowledge microtheory. GME implements the microtheory and a suite of knowledge tracking tools in the same class, called the *KnowledgeAnnotator*. A diagram of GME's *KnowledgeAnnotator* is shown in Figure 5.5.

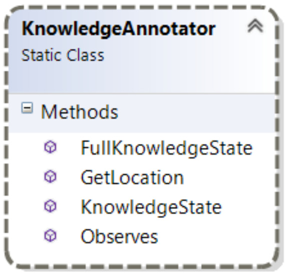


Figure 5.5: Diagram of GME's KnowledgeAnnotator class. The diagram is generated by Microsoft Visual Studio 2015. GME's KnowledgeAnnotator models a microtheory and marks state literals and actions as observed for a particular character given its microtheory rules.

The basis of the current microtheory is that things and events are observed when they are co-located with the player. This microtheory relies on determining the location of objects. The *KnowledgeAnnotator* calculates locations using the *GetLocation* method. The locations of most objects are determined by the *at* predicate bound to an object and a location. However, some domains allow objects to be in or on other objects or for a character to hold an object. The location of these objects are determined by recursively calling *GetLocation* for the object containing, supporting, or holding the original object. This process is repeated until an *at* literal is found. The *GetLocation* method is wrapped in the *Observes* method, which given a character, a state, and a literal, object, or action determines whether or not the final argument is observed by the character. *Observes* makes this determination by passing the character and the subject of the literal, the object, or the actor of the action to the *GetLocation* method and checking whether the two locations are equivalent. The *GetLocation* process can be changed to other methods of identifying location, or the *Observes* method can be changed to another means of determining observations altogether, and the way in which observed literals are identified will update without having to change the rest of the system.

The final two methods of the *KnowledgeAnnotator* return sets of observed literals given a character and a state. The *KnowledgeState* method takes as input a character and a state, iterates over the literals in the state, and checks each to see if the player observes it. This method is useful for determining the state literals to display to a user during gameplay. However, the states GME maintains make the closed world assumption. That is, anything not explicitly modeled in the state representation is false. So simply iterating through the true literals and marking them observed or unobserved only gives half of the full picture, it doesn't model what false literals can be observed. The *FullKnowledgeState* method addresses this problem by calculating the full set of true and false literals given a dictionary of typed objects and list of unbound possible predicates in addition to the state and character. Once the full state of true and false literals is calculated, all literals are marked as observed or unobserved by the character.

5.2.4 Event Revision

The first component of perceptual experience management is event revision. If event revision is enabled in the initial call to *MediationTree*'s constructor, event revision will be tried whenever accommodation fails to find a plan and a node is marked as a dead end. Instead of moving on, the dead end node is passed to a special class called the *EventRevisor. EventRevisor* creates a special event revision domain and problem pair that allows event revision to be performed by any regular, non-modified planner. If the planner finds a solution, an alternate series of past events consistent with the player's knowledge exists that enables a goal state where the original events did not. A diagram of GME's *EventRevisor* is shown in Figure 5.6. The process begins with a call to the *EventRevision* method that includes the dead end node, the mediation tree, and the planner that will perform event revision.

The *EventRevision* method begins by calling *GetEventRevisionPair* with the input node and

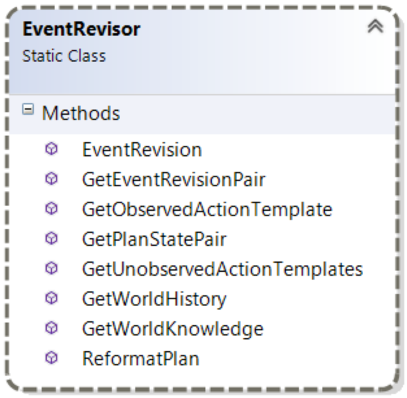


Figure 5.6: Diagram of GME's EventRevisor class. The diagram is generated by Microsoft Visual Studio 2015. GME's EventRevisor class provides a set of tools for creating special event revision planning problems and interpreting the results from a planner.

mediation tree. *GetEventRevisionPair* produces a domain and problem pair that represents the event revision problem. It begins by calling the *GetWorldKnowledge* method to retrieve a list of operator-state pairs that represent the player's observations throughout gameplay. *GetWorldKnowledge* begins by calling *GetWorldHistory* which iterates through the mediation tree object and returns the series of nodes traversed from the root to the current node by following incoming mediation tree edges upwards. Once the default world history has been computed, *GetWorldKnowledge* iterates through the world history adding actions to its list only if they are observed and pairing each action with the resulting full world state observation by the player from the *KnowledgeAnnotator*'s *FullKnowledgeState* method. This series of observed or null unobserved actions and full observation states is passed back to *GetEventRevisionPair*.

GetEventRevisionPair's task is to create a new planning domain and problem that bakes in the observations made by the player modeled by the series returned by *GetWorldKnowledge* such that any plan returned by the planner contains the same state and action information observed by the player. To do this, *GetEventRevisionPair* must bake event and state information into the new planning problem. Event information is retained by a special predicate called *state-depth* and a set of objects that represent each layer of the mediation tree from the root, *depth1*, to the current node at layer *n*, *depthn*. At this point, the planning problem is split into two parts: history and future. A special set of operators are created for future planning, each a clone of the original domain operators but with a special precondition of *(state-depth depthn)* where *n* is the number of nodes in the world history. This allows any operator to be instantiated as an action during planning as long as it is planned to take place after the current node.

To allow the planner to plan past events, *GetEventRevisionPair* iterates through the series of observed world history events and adds a new set of operators to the domain object. If the current action was observed by the player, the method calls *GetObservedActionTemplate*, and if the current action was unobserved by the player, the method calls *GetUnobservedActionTemplates*. *GetObservedActionTemplate* returns a single operator that models the action the player witnessed. The operator has no parameters, it acts as the fully ground action the player observed, adds the current depth level as its precondition, adds all of the original action's effects, and iterates the depth. If the action was observed, this single template is the only action the planner can add at its depth. However, if the action is unobserved then *GetUnobservedActionTemplates* is called. This method clones all the operators from the original domain and gives them preconditions of the current depth level as well as all observed literals from the previous state. This ensures that no action can be added that contradicts the player's observations when replanning past events.

Once it finishes iterating through the series of action observations, *GetEventRevisionPair* has compiled the completed event revision problem and domain. The method returns the pair to the *EventRevision* method, which passes them to its given planner. If the planner returns a plan, it successfully performed event revision and found an alternate history that enables a goal state to be reached. The plan returned by the planner is sent to the *GetPlanStatePair* method which in turn passes the plan to *ReformatPlan* which removes special event revision symbols from the plan results and casts actions in terms of their original domain operator names. The formatted plan is passed back to *GetPlanStatePair* where future plan steps are separated from past steps. The method then uses the past steps to simulate a world history in order to produce a new current state for the tree node. The rest of the planned actions form the plan associated with the node in the mediation tree. Once this process is finished, the node is no longer a dead end and its outgoing edges can be calculated. However, if event revision fails to find a plan GME can try its second perceptual experience management technique, domain revision.

5.2.5 Domain Revision

The second component of perceptual experience management is domain revision. If domain revision is enabled in the initial call to *MediationTree*'s constructor, domain revision will be tried whenever accommodation fails to find a plan and a node is marked as a dead end. Instead of moving on, the dead end node is passed to a special class called the *DomainRevisor*. *DomainRevisor* modifies the domain to remove possible unwanted effects. If the planner finds a solution, an alternate set of world mechanics consistent with the player's knowledge exists that enables a goal state where the original domain did not. A diagram of GME's *DomainRevisor* is shown in Figure 5.7. The process begins with a call to the *DomainRevision* method that includes the dead end node, the mediation tree, and the planner that will check domain revision.

GME models multiple domains in a single file with conditional effects. For example, the *cut-rope* operator in the Wild West domain shown in Appendix A.5.1 contains a conditional

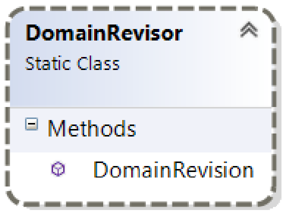


Figure 5.7: Diagram of GME's DomainRevisor class. The diagram is generated by Microsoft Visual Studio 2015. GME's domain revisor identifies and removes conditional effects that create exceptional actions if the effect rule has not been observed.

effect in its effect list. Domain revision allows these conditional effects to be removed from the domain if they contribute an exceptional effect and the condition has never been observed by the player. Given an exceptional action, the *DomainRevision* method loops through its set of exceptional effects. Exceptional effects are computed when the action is determined to be consistent, constituent, or exceptional by determining if the effect breaks a dependency used by a precondition of a later step. If the current effect is conditional, *DomainRevision* checks to see if the rule or any effect of the rule was observed by the player. If not, a copy of the original domain is created where the conditional effect is removed from its operator template. This domain is paired with the planning problem from the current node and these are given to the planner. If the planner returns a plan, domain revision was successful and the new domain, state, and plan are stored in the current node. The node is no longer a dead end and its outgoing edges can be calculated.

Event and domain revision are called when a dead end is reached by accommodation. However, this process may miss opportunities to shift between world histories and mechanics before the player takes an action that cannot be accommodated by the mediator. The final backend component, superposition manipulation, provides a framework that tracks sets of states consistent with a player's observations and shifts the player between these possible worlds as they make observations during gameplay.

5.2.6 Superposition Manipulation

The final component of GME's perceptual experience management tools proactively tracks and manipulates superpositions of possible states consistent with the player's observations. If superposition manipulation is enabled in the initial call to *MediationTree*'s constructor, superposition manipulation will be tried whenever the player makes a new observation. Superposition manipulation performs the same event and domain shifting as event and domain revision, the difference is it doesn't wait until a dead end to take action. Instead it estimates the utility of sets of states whenever the player makes a differentiating observation and transitions the player as soon as the observation is made. This allows superposition manipulation a larger space of action which potentially leads to more playtraces kept inside the bounds of author constraints but has the drawback of more costly computations to perform. A diagram of the superposition manipulation suite is presented in Figure 5.8.

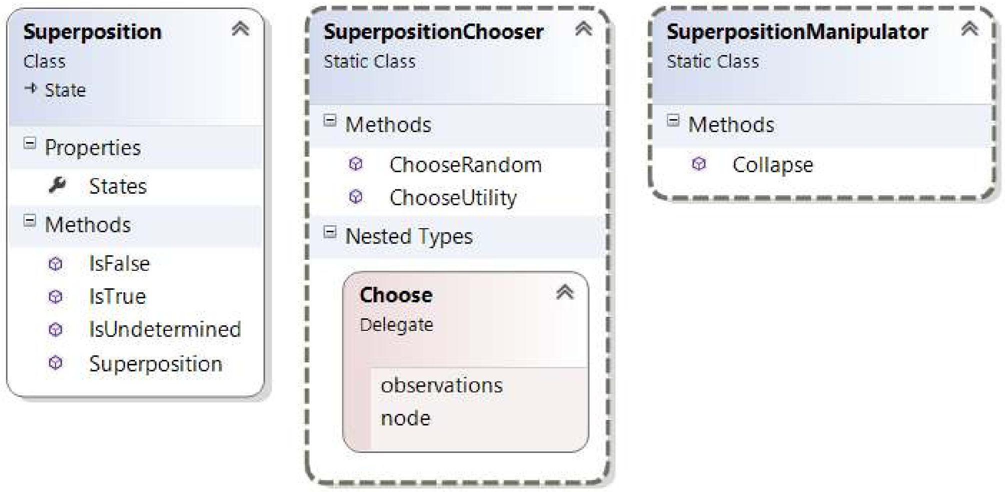


Figure 5.8: Diagram of GME's superposition manipulation suite. The diagram is generated by Microsoft Visual Studio 2015. GME tracks and answers questions about sets of states with its *Superposition* class and collapses superpositions with its *SuperpositionManipulator* and *SuperpositionChooser* classes.

The *Superposition* class tracks the different states consistent with player knowledge as play progresses. *Superposition* is a subclass of *State*, the class that implements *IState*, and takes *State*'s position in mediation tree nodes when superposition manipulation is enabled. *Superposition* is similar to state but instead of containing a single set of literals that represents a world state, it contains a set of states. Each state is consistent with what the player has observed in the game world. The superposition grows every time an NPC has an opportunity to take an unobserved action. Instead of creating a single successor state, *MediationTree* finds the successor of each outgoing unobserved edge and adds them to the parent superposition. In this way, superposition grow and account for current state that could be reached by a path through the mediation tree that is consistent with user observations. The *Superposition* class contains methods for construction and for answering whether a literal is true or false across all superposed states or undetermined. A literal is undetermined if it is true in some superposed states but false in others.

When the player makes an observation of an undetermined literal GME must decide whether the literal will be observed as true or false. If the literal is shown as true, only states in the superposition where the literal is true are now consistent with the player's observations and vice versa. This is called collapsing the superposition. It is up to GME to decide in what direction, true or false, the superposition will collapse and the *SuperpositionManipulator* is invoked to choose the direction. To determine which collapsed superposition has the best probability of reaching a goal state, the manipulator is passed a chooser method from the *SuperpositionChooser* class. Each choose has its own unique way of determining the utility of superposed states. For example, two current methods are *ChooseRandom* and *ChooseUtility*. Random chooses between the different superpositions according to a random number and Utility chooses based on a heuristic of how many states in each superposition produce a plan. Together, these classes form a superposition manipulation framework that given the right choice function can shift the player between superposed states as they play.

Superposition manipulation is the last component of GME's backend architecture. The next section describes the existing player interface pipelines that configure and display state and action information to a human user and expand the underlying mediation tree based on human feedback.

5.3 Interface Pipelines

Instead of sensing and affecting an external game environment, GME procedurally generates a game interface that matches the current PDDL world state and follows the rules of the planning domain and problem. This general process of constructing an interface for the player that matches the PDDL model can take many forms. This section presents a series of four interfaces for GME constructed from two types of media, text and visual 2D.

5.3.1 Text Interface

The first series of interfaces use text templates to display information to the player and allows the player to expand the interactive narrative tree by sending back text commands that correspond to enabled actions in the current world state.

Text Interface Architecture

Below each of the text interfaces is a set of shared tools that format state literals and observed actions into text descriptions and allow player commands to be parsed and translated to appropriate enabled actions in the interactive narrative tree. This tool system has four components: a control module, a knowledge model, a state and action description generator, and a command parser. Together these four components control operation, generate state descriptions, and control how GME expands its mediation tree based on user input. A diagram of the text interface architecture is shown in Figure 5.9.

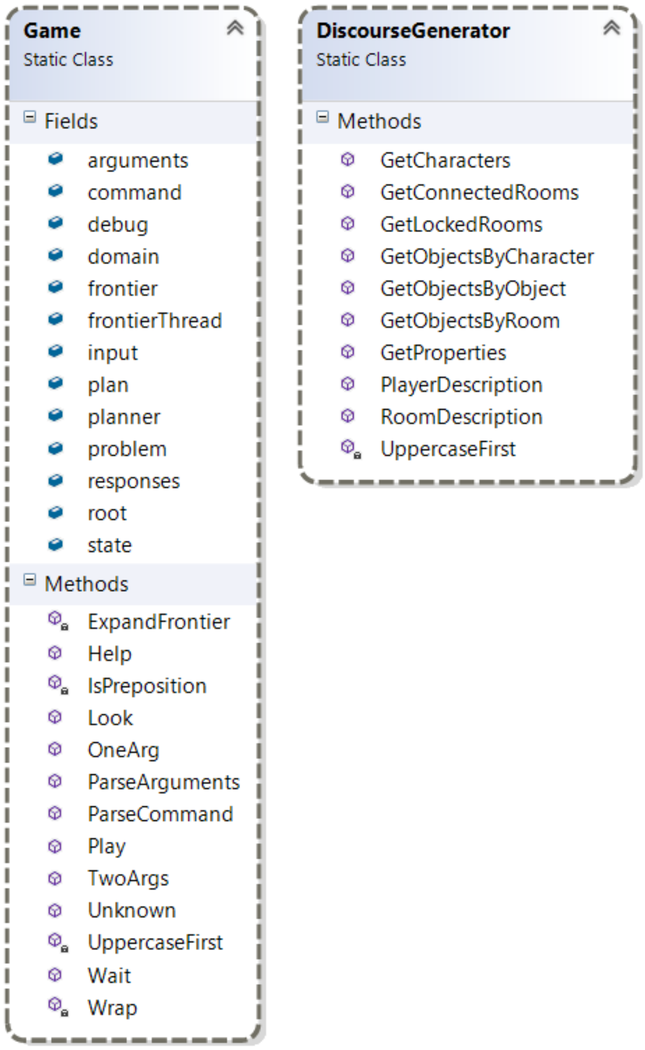


Figure 5.9: Diagram of GME's text interface architecture. The diagram is generated by Microsoft Visual Studio 2015. GME tracks state information and parses commands with the *Game* class and formats states and events into natural language with its *DiscourseGenerator* class.

The first component of the shared text interface tool system is a control module responsible for controlling the operational flow for a given session. This control module is started by calling the *Play* method of the *Game* class. A session begins by reading in a PDDL problem and domain from file, storing these in the *domain* and *problem* fields of the *Game* class, based on a selection made by the user or an external system request. Once the domain and problem has been selected, a plan is generated by the planner stored in the *planner* field, the result is stored in the *plan* field, the initial state is stored in the *state* field, and the root of a GME mediation tree is generated and stored in the *root* field. Once initialization is finished, the *Play* method shifts control to interactive narrative gameplay.

Gameplay begins by setting the *command* field to *look* and the *argument* field to an empty list. The lone *look* command is short hand for examining the player's current room. Once these fields are set, the *Look* method is called. The *Look* method checks the *command* and *argument* field and then asks the *DiscourseGenerator*'s *RoomDescription* method for a text template describing the initial state. The *PlayerDescription* method takes as input the current state and the player's character name and uses the *KnowledgeAnnotator* to format a description of the initial state for the player. To appropriately display actions and their outcomes or state literals to the player the system needs to know what world properties are observed by the player and what aren't. A model of user knowledge separates these two sets so only world components observed by the player are shown on screen. The text interface uses GME's knowledge microtheory, implemented by the *KnowledgeAnnotator*'s *Observes* method, to determine what state literals should be formatted and displayed to the user.

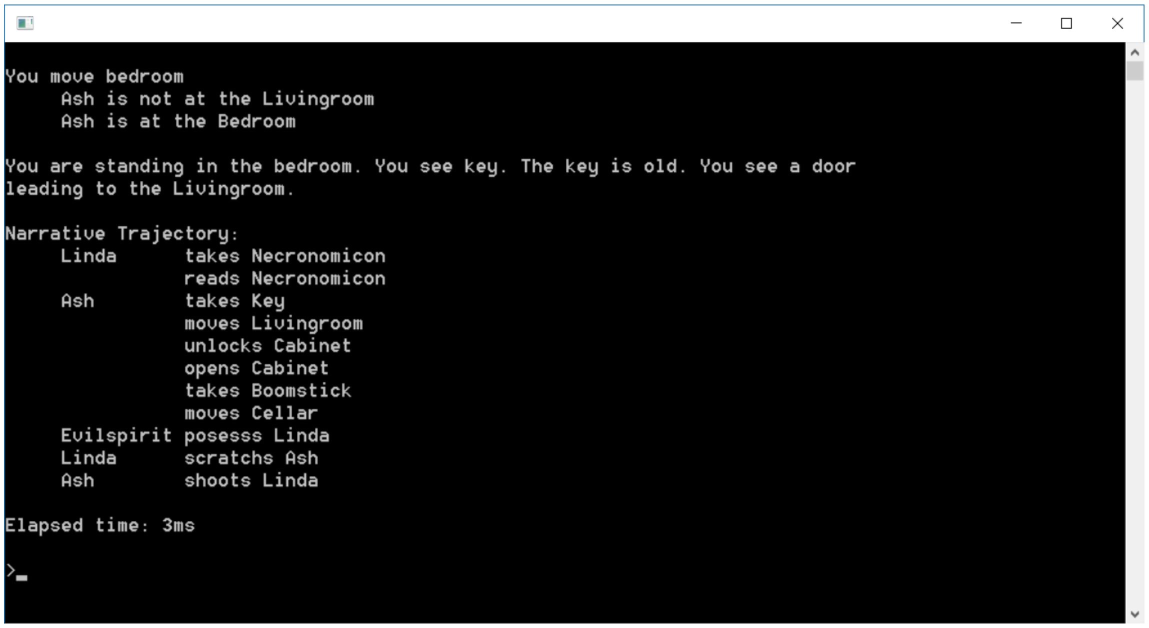
Once the model of user knowledge determines what state literals the user observes, the *DiscourseGenerator*'s *RoomDescription* method formats the set of observed state literals into a natural language description. The generator loops through the currently observed state literals and formats each literal into a natural language sentence using simple text template rules. These rules map literals like *(at player room)* to "You are standing in the room." and *(has player ball)* to "You are holding a ball." and *(green ball)* and *(bright ball)* to "The ball is green and bright." and *(at friend room)* to "You see a friend." Together, these templates configure and display in natural language a declarative state for the player. The *DiscourseGenerator* methods *GetCharacters*, *GetConnectedRooms*, *GetLockedRooms*, *GetObjectsByCharacter*, *GetObjectsByObject*, *GetObjectsByRoom*, and *GetProperties* are used by the *PlayerDescription* and *RoomDescription* methods to format each aspect of the observable state. The formatted text string is returned to the *Look* method which writes the string to output and returns control to a parse command-display state gameplay loop in the *Play* method.

The final component of the text interface architecture is an action parser, implemented by the *Game* class methods *ParseCommand*, *ParseArguments*, *OneArg*, *TwoArgs*, and *Unknown*. Instead of receiving an exact description of a fully ground action, something like *(move player room2 room1)*, the parser allows the player to specify an abbreviated description, like *move room2*, and fills in the rest of the details automatically. To work, domain authors must adhere to the convention that an action performer is the first parameter of each action. This convention allows the system to fill in the player's character for the first action parameter. The second convention domain authors should follow is use dashes in an action name to specify what level of detail is needed when typing in an action. For example, to move from one room to another a domain generally needs to know the character's name, the current room, and the room the character is moving to, something like *(move ?character ?to ?from)*. However, the parser can automatically fill in the player's name and where the player is moving from because this information is the same across all enabled actions. All the parser needs to know is the player wants to move and where the player wants to move to, or *move room2*. This level of detail can be specified by the domain author by listing the move action as *(move-location ?character ?to ?from)*. The parser knows that the first word is the name of the action and the number of dashes specify the number of parameters the player needs to specify when selecting the action. So with the command *move room2* the parser can successfully look up and identify the action *(move-location player room2 room1)* from other possibly enabled actions like *(move-location player room3 room1)*.

Once a command is issued, the *Play* method transitions down the associated edge in the mediation tree, updates its information, takes NPC actions, then formats and displays any observed actions for the player. Observed actions are determined by the *KnowledgeAnnotator* and are presented to the player by first displaying the action name and who took the action. Then each action outcome, or bound effect, of the action is printed. Once this is finished, the current state is printed by the *DiscourseGenerator* and the system loops back around to wait for command input by the player. Together, these components form a general text game architecture for displaying states, accepting user commands, and transitioning through a mediation tree. This general architecture allows a number of specific interfaces to be built. The next four subsections describe a series of text interfaces that use this architecture to create interaction.

Console Interface

GME's first interface, shown in Figure 5.10, is a windows console application [56]. The interface runs the GME library as a DLL inside the console application to create and maintain interactive narrative trees based on an input planning problem. The interface uses simple text templates to convey world states and character actions from the GME backend to the player. The player can choose to take actions by typing enabled actions into the text input. When the player types in an enabled action, the interface informs GME, which transitions down the corresponding edge in its branching story structure. GME takes action for its system-controlled characters according to the current plan and then the interface displays the system actions along with the new world state to the player. Figure 5.10 shows the system in debug mode, which lists the events of the current story branch the system is using as the narrative trajectory. Figure 5.10 shows GME running the *Zombie* domain, given in Appendix A.6.



```
You move bedroom
     Ash is not at the Livingroom
     Ash is at the Bedroom

You are standing in the bedroom. You see key. The key is old. You see a door
leading to the Livingroom.

Narrative Trajectory:
     Linda      takes Necronomicon
                reads Necronomicon
     Ash        takes Key
                moves Livingroom
                unlocks Cabinet
                opens Cabinet
                takes Boomstick
                moves Cellar
     Evilspirit posesss Linda
     Linda      scratchs Ash
     Ash        shoots Linda

Elapsed time: 3ms

>_
```

Figure 5.10: GME's text-based console interface.

GME Web Service

A web service version of GME exists to enable the system to function over the web. Instead of being compiled as a DLL, this online version of GME runs on an IIS server as a C# web service. The GME web service tracks sessions by IP address and exposes functionality through HTTP requests and responses. A diagram of the web service system is shown in Figure 5.11. The service's *TextGameService* class serves as a modified version of the original GME *Game* class. The two biggest changes are that all data storage is moved to an external *Data* class and that *TextGameService* tracks multiple sessions instead of only a single game.

The *Data* class contains fields and methods that model and expose all the same information tracked by the original *Game* class. The major change is that instead of only tracking a single state, plan, or mediation tree node, the *Data* class tracks one for each active session. It does this by containing dictionaries for each field that map a session identifier to specific data. The *Data* class exposes its fields through get and set methods that take as input a session identifier to store and lookup information in the field dictionaries. *Data* also contains methods for manipulating session information, like *ClearSession* for removing all data for a particular session, *IPInitialized* which checks whether an IP address is mapped to a session identifier, *SessionInitialized* which checks whether a session identifier is currently active, *GetIPs* which lists all active IP addresses, and *GetSession* which returns the session identifier for an IP.

The *TextGameService* class functions like the original *Game* class but exposes its function-

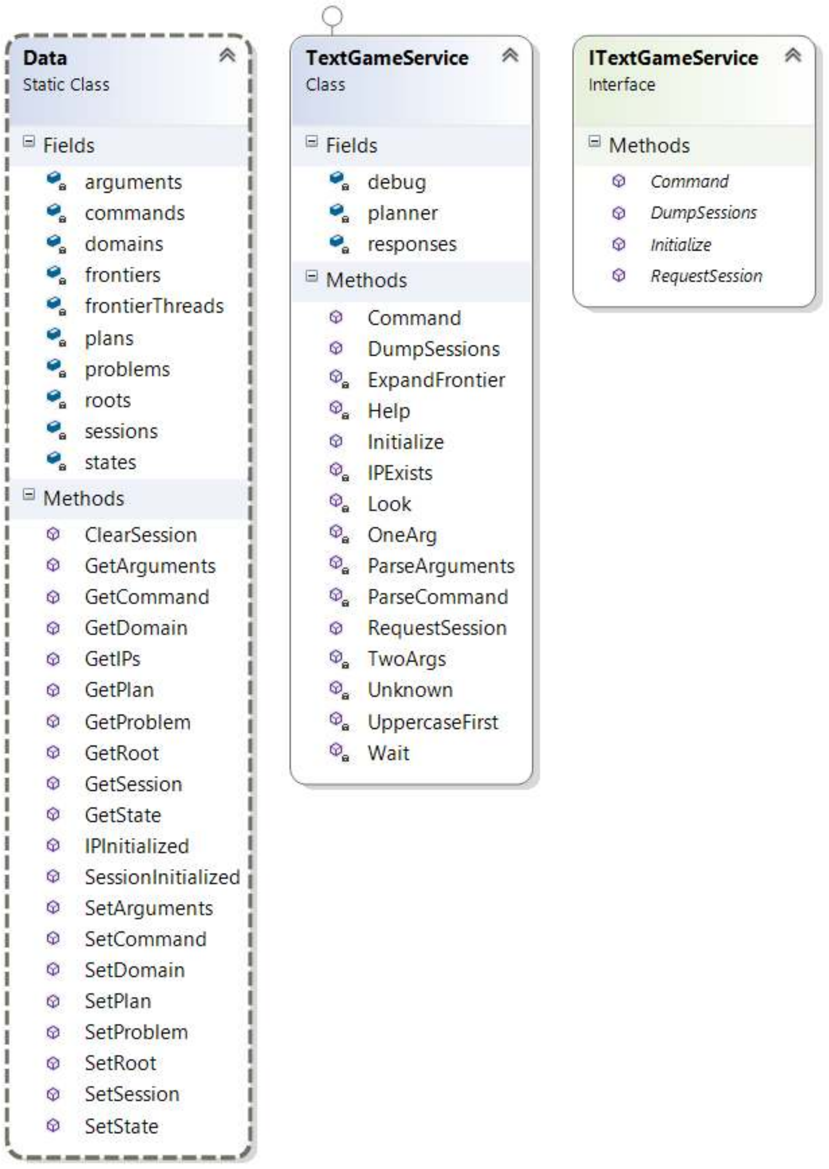
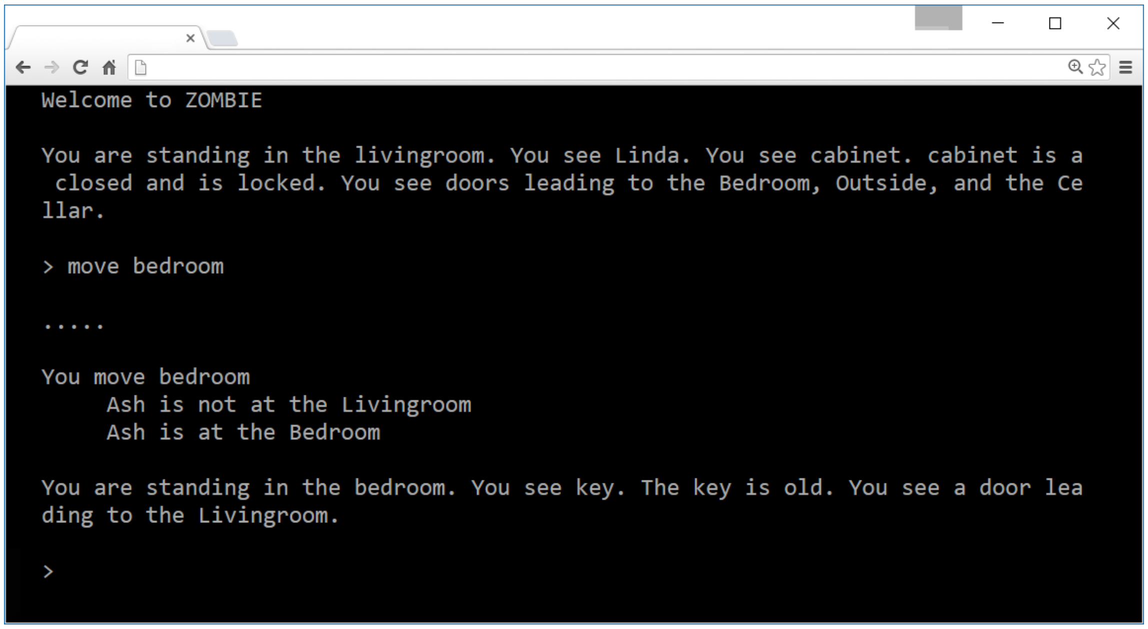


Figure 5.11: Diagram of GME's text interface web service. The diagram is generated by Microsoft Visual Studio 2015. The *TextGameService* class exposes the functionality of the original *Game* class as a service while also storing session information for multiple concurrent games.

ality as a webservice and is capable of managing many mediation tree games concurrently. The *ITextGameService* interface shows the four methods *TextGameService* exposes over the web, *Command* which takes as input a user command, transitions the mediation tree, and sends state feedback to the client, *DumpSessions* which clears all session data, *Initialize* which initializes a new mediation tree for a given domain, session, and IP address, and *RequestSession* which creates a session identifier for an IP address. Together, these classes allow commands and state feedback to be sent between a web service and a client over HTTP. This section described the service. The next two sections describe two clients that consume this service and provide gameplay functionality to the user.

Online Console Interface

The first online interface, shown in Figure 5.12, is a web version of the text-based GME console application. The GME web service is consumed by the interface through PHP scripts integrated into a JavaScript front end. The JavaScript interface mimics the GME console application in a web browser. The interface sends input player actions to the GME web service through HTTP via PHP, then formats and displays GME's returned results from PHP to the simulated javascript console. Like Figure 5.10, Figure 5.12 shows GME running the *Zombie* domain, given in Appendix A.6.

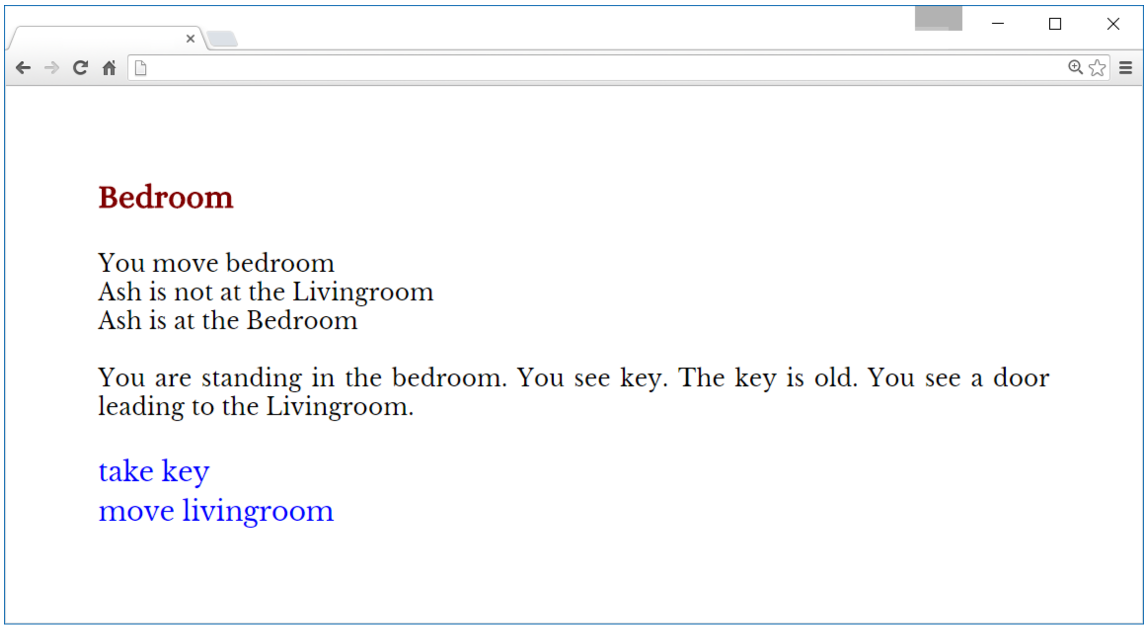


```
Welcome to ZOMBIE

You are standing in the livingroom. You see Linda. You see cabinet. cabinet is a
 closed and is locked. You see doors leading to the Bedroom, Outside, and the Ce
llar.

> move bedroom

.....

You move bedroom
     Ash is not at the Livingroom
     Ash is at the Bedroom

You are standing in the bedroom. You see key. The key is old. You see a door lea
ding to the Livingroom.

>
```

Figure 5.12: GME's online console interface.

Online CYOA Interface

The third interface, shown in Figure 5.13, is a second online version of the text-based GME interface. The interface is a Choose Your Own Adventure (CYOA) style web page that presents possible actions to players as hyper-links instead of console commands. The CYOA interface uses the same web service back-end as the online console interface, but presents the information to the user in a traditional HTML format with enabled actions presented as hyperlinks. When a hyperlink is clicked by the user, the system sends a command request to the web service using the formatted command the player clicked on. Because the online console and CYOA interfaces consume the same GME web service backend, the same backend game session can be used by both the console and CYOA front ends. Figure 5.13 shows the state reached by performing *move bedroom* in the *Zombie* domain from the online console interface, pictured in Figure 5.12. Both figures show the same backend session.



Bedroom

You move bedroom
Ash is not at the Livingroom
Ash is at the Bedroom

You are standing in the bedroom. You see key. The key is old. You see a door leading to the Livingroom.

take key
move livingroom

Figure 5.13: GME's online Choose Your Own Adventure interface.

This distinction between mediation tree expansion and the interface used to visualize state information and relay commands back to GME is expanded on in the next section which presents a visual 2D interface, implemented in Unity, for the same backend GME architecture.

5.3.2 Visual Interface

The final interface is a visual 2D system implemented in the Unity game engine [57], called the Unity General Mediation Engine (UGME). Instead of conveying world states and character actions with text templates, UGME constructs and controls a 2D environment from a GME DLL using the Unity game engine. UGME allows the player to expand the interactive narrative tree by taking action in a 2D game space that corresponds to enabled actions in the current declarative world state. This section describes the UGME architecture, beginning with the backend in Section 5.3.2, to the NPC architecture in Section 5.19, to the player interface in Section 5.20. The current UGME pipeline is tightly tied to a game called Base Case [9], a sneaking game that is similar to the original *The Legend of Zelda* and *Metal Gear* games for the Nintendo Entertainment System. The Base Case game is described in Section 5.3.2 and a domain and problems for Base Case are given in Appendix A.1. UGME is more tightly tied to a single game than the text pipeline because it's much easier to generate text assets from a PDDL domain and problem than art, animation, and code assets that represent a visual game world. The generality of UGME and future work involving the procedural content generation (PCG) community is discussed further in Section 5.3.3.

Visual Interface Architecture

Below the UGME's visual interface is a set of tools that interface with GME and format state literals into game world configurations. This architecture has two main components: a GME interface and a game world manager. The *Mediation* class interfaces with GME, receives declarative action requests from the player, transitions through the underlying mediation tree, updates world state information, and sends commands to NPCs based on transitions in the mediation tree. *Mediation* instantiates the game world using a class called the *LevelGenerator* which in turn uses a class called *MapNode* to determine game world layouts. Once generated, the game world is managed by the *StateManager* class. *StateManager* updates game and user interface objects based on the current declarative state. A diagram of these four classes is shown in Figure 5.14.

The core of UGME is the *Mediator* class which interfaces with GME to manage a mediation tree. The *Mediator*'s *Start* method is invoked by Unity when a game session begins. The *Start* method parses a GME domain and problem object into its *domain* and *problem* fields, finds and stores a plan object in its *plan* field, stores the initial state in its *state* field, finds references to UGME's state manager, level generator, and UI generators and stores them in its *stateManager*, *generator*, and *uiGenerator* fields. *Mediator* then initializes the game world with a class called *LevelGenerator* and the root of the GME mediation tree. As part of the initialization process, *Mediator* sets the *StateManager*'s set of initial world state literals.

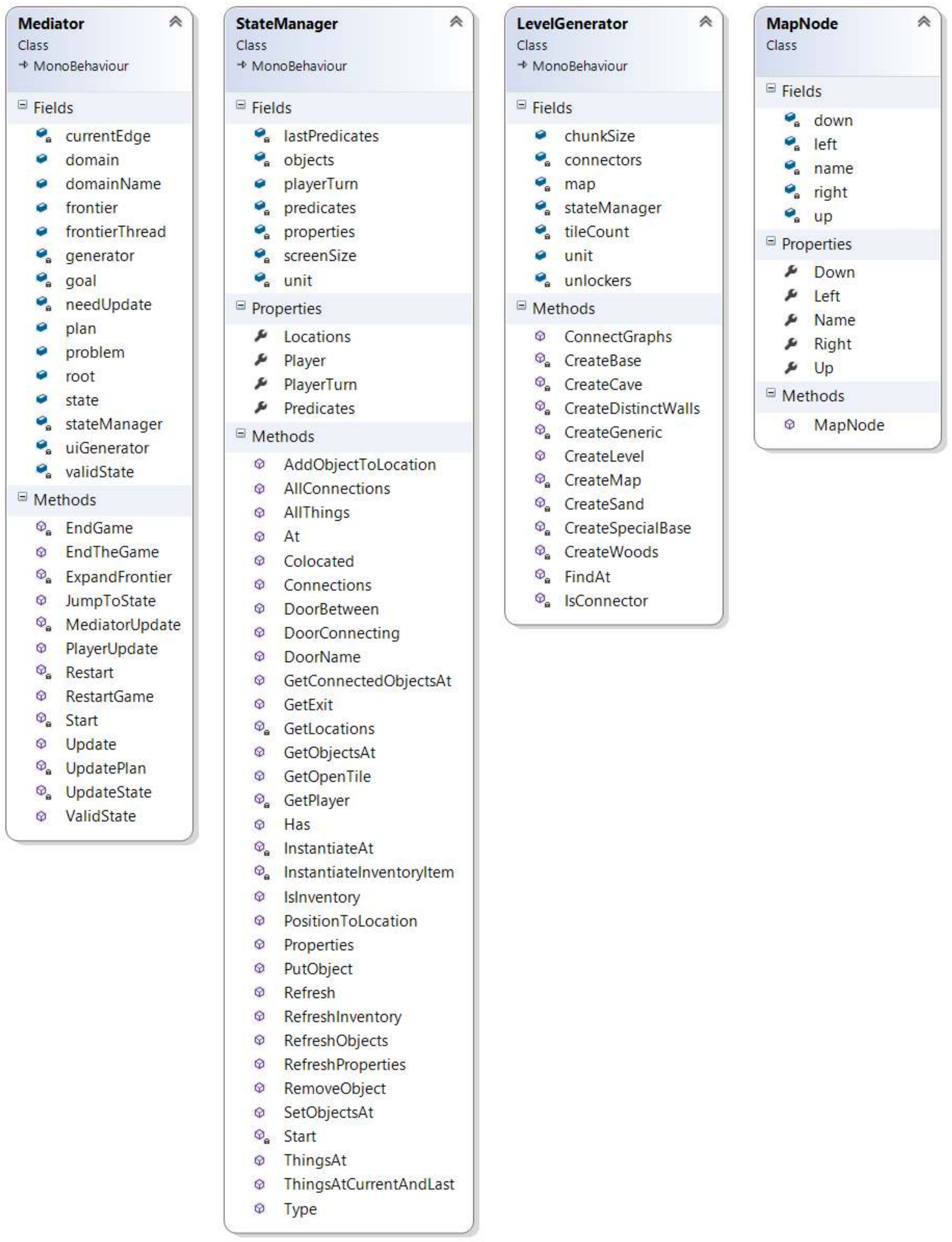


Figure 5.14: Diagram of UGME's backend architecture and level generator. The diagram is generated by Microsoft Visual Studio 2015. The *Mediator* and *StateManager* interface with the underlying GME mediation tree and automatically configure the Unity game environment to match the underlying declarative state. LevelGenerator and *MapNode* configure a world layout and generate individual rooms according to the planning problem's initial state.

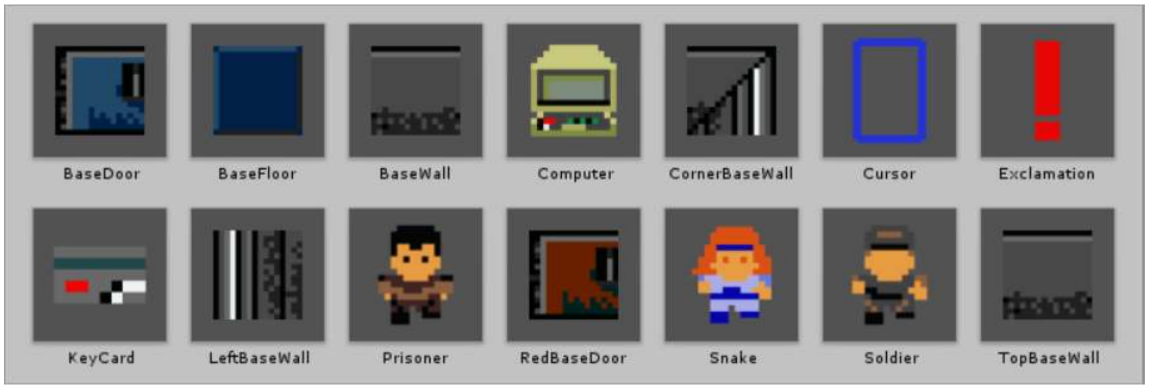


Figure 5.15: Base Case's resource library of Unity prefabs.

The *LevelGenerator* initializes the game world according to the initial state of the planning problem using special Unity resources called prefabs. A diagram of the *LevelGenerator* class and its *MapNode* helper class is shown in Figure 5.14. A prefab is a pre-assembled package of art and code assets that can be instantiated in a game world. UGME uses 2D prefabs of a fixed height and width that can be tiled on the game screen. A diagram of Base Case prefabs is shown in Figure 5.15. The *LevelGenerator* class assembles a configuration of rooms by instantiating and tiling prefabs based on a set of room connections and types listed in the initial state of a planning problem. First, *LevelGenerator* solves how the rooms will be laid out relative to one another on the screen. Domain authors define rooms relative to their connections between one another rather than explicitly saying how one is above or below another. The *LevelGenerator* first uses its *CreateMap* method to solve this problem by mapping a set of connections to a particular layout. Figure 5.16 shows one possible high level layout for the Base Case domain's first problem, presented in Appendix A.1.2.

The problem has seven room objects: cell, hub, dorm, exit, outside, computerroom, and realescape. The problem has bidirectional connections with doors between hub and cell, hub and exit, hub and dorm, exit and outside, and computerroom and realescape. The problem has a bidirectional connection with no door between dorm and computerroom. Each of these rooms is assigned a *MapNode* object by the *CreateMap* method which finds a particular up-down-left-right room configuration that matches the high level connections defined in the planning problem. In the case of Figure 5.16, the method places hub in the top middle with dorm to its left, cell to its bottom, and exit to its right. Outside is placed below exit and computerroom and realescape are placed below dorm. Once a valid map configuration is found by *CreateMap*, *LevelGenerator* places background tiles for each room given their particular type. The hub, cell, exit, dorm, and computerroom are of type base, the outside room is of type woods, and the realescape room is of type cave. *LevelGenerator* calls the *CreateBase*, *CreateCave*, and

CreateWoods methods to instantiate the room types at the correct positions. Each of these in turn call the *CreateGeneric* or *CreateDistinctWalls* methods with the different asset packages for the floor, walls, and doors in order to instantiate the different rooms.

At this point, the *Mediator* waits until the player takes an action. When the player takes some declarative action in the game world, the command is set to the *Mediator*'s *PlayerUpdate* method. The *PlayerUpdate* method interfaces with GME and checks whether the command matches any outgoing edge of the current mediation node. If so, *PlayerUpdate* transitions down the mediation tree edge and updates its *root*, *problem*, *plan*, and *state*. The *Mediator* loops through each NPC action it traverses in the mediation tree and sends a request to the NPC to perform the action. At this point, the *UpdateState* method is called which updates the *StateManager*'s set of state literals and calls its *Refresh* method. Whenever the *StateManager*'s *Refresh* method is called, it updates the configuration of game objects in Unity's game world.

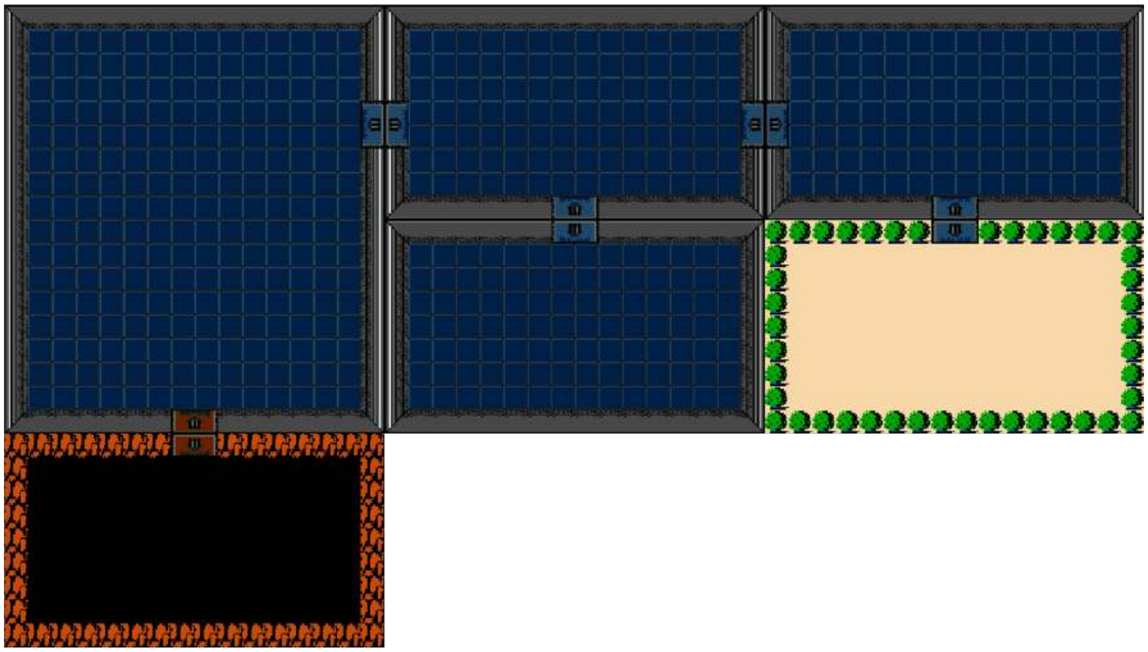


Figure 5.16: A layout of the Base Case level created from the problem in Appendix A.1.2.

StateManager serves as the glue between the declarative GME backend and the procedural Unity front end by providing methods that link declarative interactive narrative tree objects to Unity game objects and prefabs. The class contains information about every game object in the game world that is instantiated and updated to represent a constant in the underlying mediation tree. *StateManager* updates its representation with the *Refresh* method every time the underlying state is changed. *Refresh* in turn calls *RefreshObjects*, *RefreshInventory*, and *RefreshProperties*. *RefreshObjects* loops through every world location, provided by *GetLocations*, and finds a list of all game objects at that location, provided by *GetObjectsAt*. The method also finds a list of all declarative objects at the location using the *ThingsAt* method. The method then loops through the game objects at the location in the game world and destroys any that aren't there in the declarative state and loops through the objects at the location in the declarative state and uses the *InstantiateAt* method to add any that aren't there in the game world, which uses the *GetOpenTile* method to find the game world location of empty tiles in the game object's room. Finally, the *SetObjectsAt* method is used to associate any newly instantiated objects with the current location in *StateManager*'s object tracking dictionary. Figure 5.17 shows game world from Figure 5.16 once the set of game world objects have been instantiated by the *StateManager*.

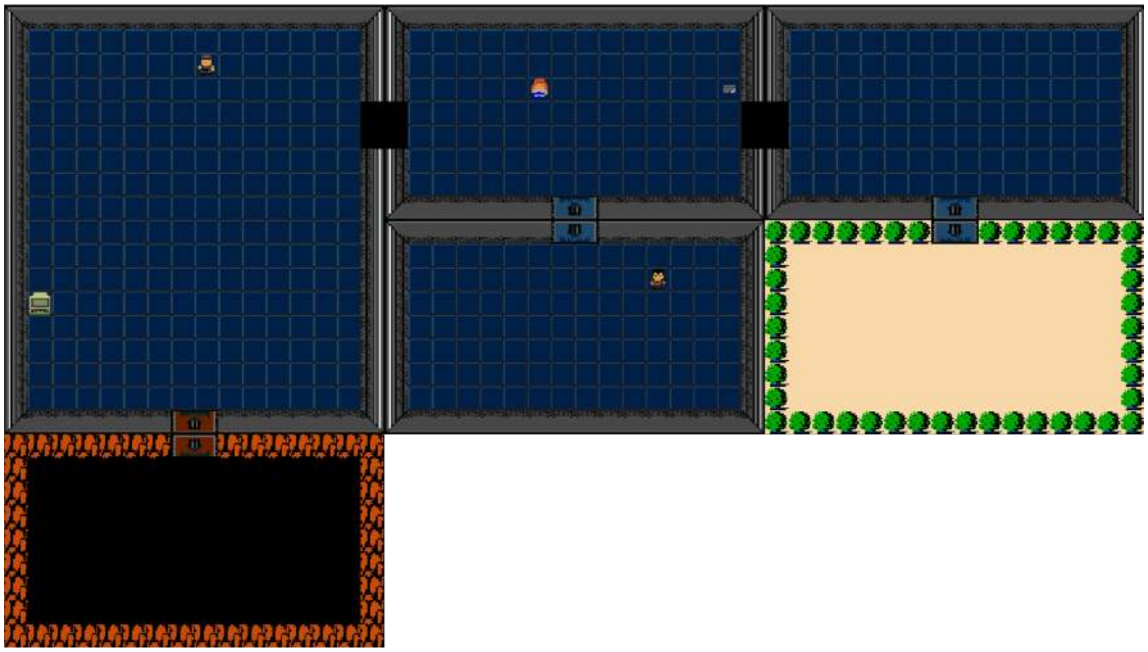


Figure 5.17: The initial state of the level created from the problem in Appendix A.1.2.

Similar to *RefreshObjects*, *RefreshInventory* iterates through and updates the objects not at a location, but in the user's inventory which is displayed by the UI. The method begins by using the *GetObjectsAt* method to find and store the list of game world objects located at the player and using the *Has* method. *RefreshInventory* loops through the inventory game objects and destroys those that are no longer in the declarative representation, then loops through the declarative representation and uses *InstantiateInventoryItem* to create any game objects that are not yet represented. *RefreshInventory* ends by using the *SetObjectsAt* method to set newly instantiated objects to belong to the player's character in the object tracking dictionary.

The final refresh method, *RefreshProperties*, is different than the other two methods. Instead of instantiating and destroying objects based on the underlying declarative state, this method updates a game object's animator based on property information. The method begins by looping through all currently instantiated objects, found by the *AllThings* method. It then uses the *Properties* method to get a list of properties for the current objects, compares the stored properties to the current ones in the declarative state, and creates a list of properties to add and those to remove. Once this process is finished, the method finds the game object that corresponds to the current declarative object and finds the game object's animator. Every game object's animator is responsible for updating the image displayed by the game object. The animator is a finite state machine where every state corresponds to a still frame or animation and edges specify how the animator switches between states. UGME uses animation parameters to update game object properties based on the underlying state.

For example, Figure 5.18 shows a simple Unity animator for a door. The door can be open or closed. The open state corresponds to a blank black tile that can be traversed by the player. The closed state corresponds to a door tile that the player collides with and cannot pass. Whether the animator draws the open or closed door depends on a boolean value named *locked.* This boolean value corresponds to a literal, *(locked door)*, in the underlying declarative state that can be true or false. When this literal changes value in the mediation tree state, the *RefreshProperties* method will change the boolean value in the corresponding game object's animator so that it animates according to its underlying declarative properties. As an example of this process, notice the difference between doors in Figures 5.16 and 5.17. Figure 5.16 shows the Base Case world after *LevelGenerator* finishes processing the game world but before *StateManager* instantiates game objects and refreshes properties so all the door objects created by *LevelGenerator* are in their default locked state. Figure 5.17 shows the game world once *StateManager* finishes its update including its *RefreshProperties* method where it updates the animation of two doors to be open according to the initial state.

Together, *Mediator*, *LevelGenerator*, *MapNode*, and *StateManager* interface with GME to build a mediation tree and instantiate, destroy, and update game objects based on GME's declarative state representations. The next section describes UGME's NPC architecture, a method of manipulating NPC characters according to local and planned behaviors.

NPC Architecture

Once the world is generated, UGME must control NPC actions according to edges traversed by GME's underlying mediation tree. Since NPCs don't actually change the game world, the declarative GME actions update state information, NPC actions can be thought of as animations

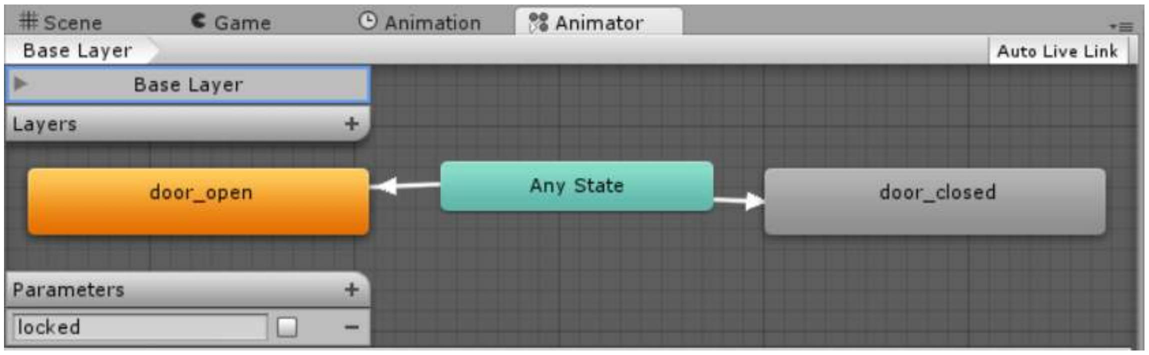


Figure 5.18: A simple Unity animator for the BaseDoor prefab.

that convey underlying transition information to the player and let GME know when to take transitions. The UGME NPC architecture has two basic components: an NPC controller class attached to each computer controlled character called *NPC* and a behavior library of actions the NPC is capable of taking called *Behaviors*. The Base Case implementation includes a subclass of *NPC* for the prisoner NPC and a *LevelTile*, *Pathfinding*, and *Exclamation* class used by the *Behaviors* class to provide animations. Together, these classes animate NPCs for the Base Case domain. A diagram of the NPC architecture is shown in Figure 5.19.

The core of the NPC architecture is the *NPC* class. The *NPC* class or a derived subclass is attached directly to each NPC prefab and is used to control its behavior during gameplay. The primary behavior of NPCs in the Base Case domain is navigation. UGME implements navigation through a the helper classes *LevelTile* and *Pathfinding*. The *LevelTile* class enables dynamic graph creation during level generation and is attached to every traversable floor tile object. As *LevelGenerator* lays floor tiles down during the room generation process, it lets each tile know about its surrounding level tiles using the *SetUp*, *SetDown*, *SetLeft*, and *SetRight* methods. Each *LevelTile* also contains a trigger collider and maintains whether an object is above it or not. Together, the tile's neighbor list and whether the tile is traversable allow a pathfinding algorithm to run on a dynamically created and updated graph whose nodes represent each tile in the game world space in order to find paths between game world locations. The *Pathfinding* class implements this behavior by providing an *AStar* method that uses the A* algorithm with game space distance as a heuristic to find the shortest path between two *LevelTile* objects. The *AStar* method returns a list of *LevelTile* objects that should be traversed one at a time by the NPC through game world space.

In general, the *NPC* class controls characters according to instructions from two sources: local behavior defined by *Behaviors* and planned behaviors received from *Mediator* and interpreted with *ActionLibrary*. When the *Mediator* transitions over an NPC action in the mediation tree, the class sends the fully ground action object to the corresponding *NPC*'s *AcceptInstruc-*

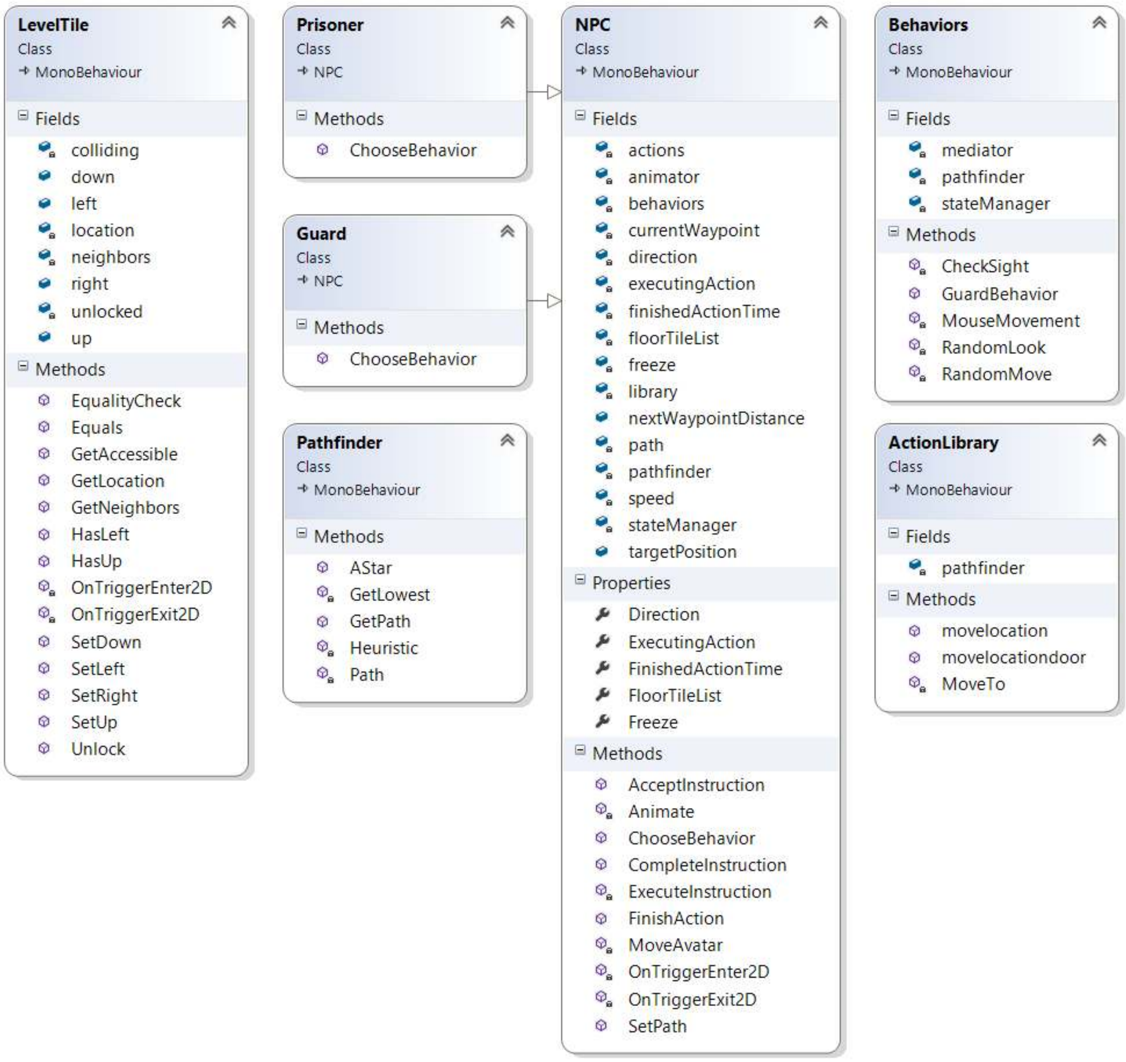


Figure 5.19: Diagram of UGME's NPC architecture. The diagram is generated by Microsoft Visual Studio 2015. The *NPC*, *Behaviors*, and *ActionLibrary* classes control local and planned NPC behavior. The *Prisoner* and *Guard* classes provide character-specific NPC behavior. *LevelTile* and *Pathfinding* provide pathfinding information for navigation animation.

tion method. The *AcceptInstruction* method adds the action object to a queue of accepted actions, stored in the *actions* field. During the *NPC*'s update cycle it checks whether any planned action is waiting in the *actions* queue. If so, *NPC* calls the *ExecuteInstruction* method which checks whether the NPC is currently executing a previously requested action. If not, *ExecuteInstruction* dequeues the first action from *actions* and uses reflection to call the appropriate method from *ActionLibrary* to execute the animation for the planned action.

The *ActionLibrary* provides public methods that correspond to PDDL action operators and private methods that provide instructions for executing the actions in the Unity game world. The Base Case *ActionLibrary* contains methods for animating *(move-location ?mover ?location ?oldlocation)* and *(move-location-door ?mover ?door ?location ?oldlocation)* actions from the Base Case PDDL domain shown in Appendix A.1.1. Both *ActionLibrary*'s *movelocation* and *movelocationdoor* methods map to a *MoveTo* method that finds an open tile in the room bound to *?location*, finds a *LevelTile* path to that location, instructs the NPC to move along the *LevelTile* path, and then calls the NPC's *CompleteInstruction* method to remove the instruction from its *actions* queue. This process is enabled by reflection from the *NPC* class's *ExecuteInstruction* method which finds the *ActionLibrary* method matching its current instruction's name (with dashes removed), and then invoking the method with itself and the action object as parameters. Using this method, NPCs can accept and execute game world commands from the mediator as long as they have an *ActionLibrary* that matches possible operators from the underlying PDDL domain.

Finally, the *Behaviors* class provides behaviors that do not change declarative state information that NPCs can take between instructions from the planner. The Base Case *Behaviors* class contains methods for moving a character between open tiles in a room, making the character look in a random direction, check whether the character sees another character along their sight line, and move to an open tile indicated by a mouse click. The class also contains a method for high-level guard behavior, in which the guard patrols a room by combining the *RandomMove*, *RandomLook*, and *CheckSight* behaviors. If the *NPC* class determines there are no current or queued actions from the *Mediator*, it transitions control to the *ChooseBehavior* method. The *Guard* and *Prisoner* classes are subclasses of *NPC* that provide different *ChooseBehavior* methods for different types of NPC characters. *Guard*'s *ChooseBehavior* calls the *Behaviors*' *GuardBehavior* method to create patrol their current room. The *Prisoner*'s *ChooseBehavior* method, on the other hand, is empty. This leaves *Prisoner* NPCs standing still when not acting out instructions from the *Mediator*.

Together, the *NPC*, *Behaviors*, and *ActionLibrary* classes allow NPCs to execute local game world as well as planned declarative world behaviors. The next section describes how UGME allows the player to take action and relays player activity as declarative actions to the *Mediator*.

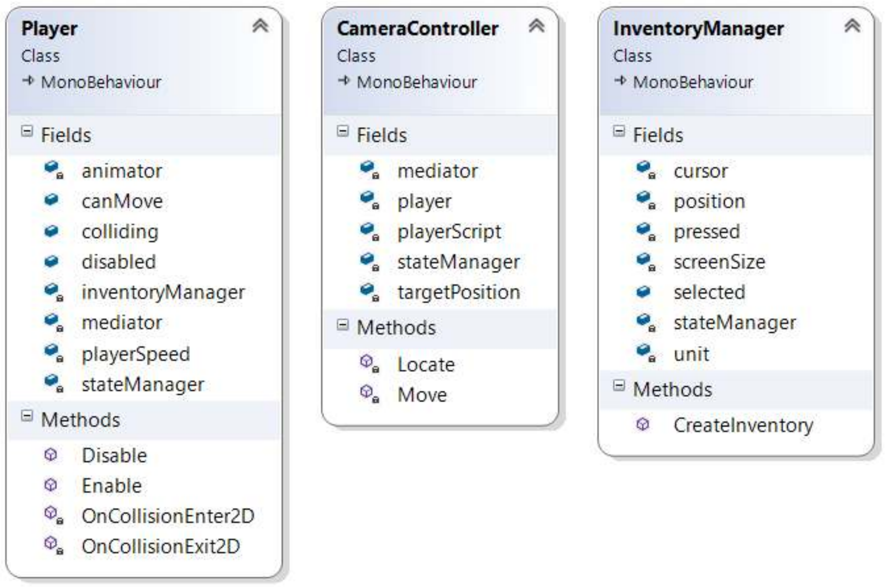
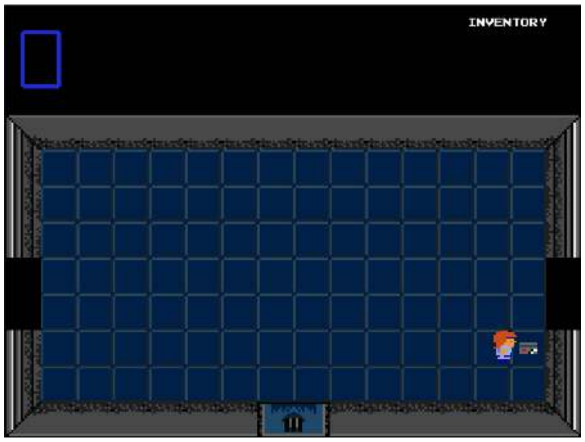


Figure 5.20: Diagram of UGME's player architecture. The diagram is generated by Microsoft Visual Studio 2015. The *Player*, *InventoryManager*, and *CameraController* relay declarative actions taken by the player in the game world to the *Mediator*.

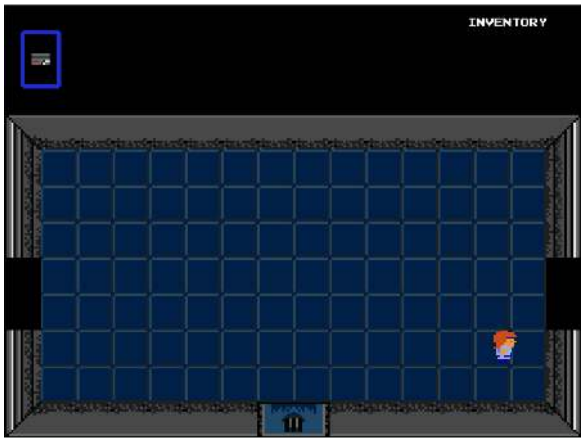
Player Architecture

The final structural component of UGME is the player architecture that recognizes when the player takes actions in the game world that correspond to a bound PDDL action and pass the information to the *Mediator*. Most of this process takes place in the *Player* class which is attached to the player character's game object and matches direct button input to animations and context-sensitive declarative actions. Some additional processing may take place in other places, like Base Case's *CameraController* which senses when the player has moved from one location to another and *InventoryManager* which draws and allows the player to select inventory items. A diagram of the *Player* and *CameraController* classes is given in Figure 5.20.

Most work of the player architecture is done in the *Player* class. This class maps action inputs from the keyboard or joystick, named by the Unity input system, to declarative action requests. For example, if the player presses the *A* button on their gamepad the Unity input maps this to the Take, Open, and Untie commands. The *Player* class update loop then checks what things the player is colliding with and sends a command for each matching declarative action to the *Mediator*. For example, if the player is standing in the *hub*, colliding with *cell_door*, and holding the *card* the *Player* class will send the following three commands to the *Mediator* when the player presses the *A* button: *(take-thing player cell_door hub)*, *(open-door player card cell_-door)*, and *(untie-person player cell_door hub)*. The *Mediator* checks each command, transitions



(a) Before the player takes the keycard.



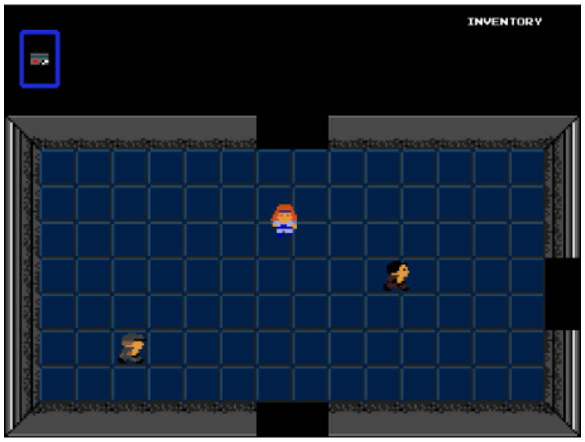
(b) After the player takes the keycard.

Figure 5.21: The Base Case domain generated from the problem in Appendix A.1.2 before and after the player presses a button that triggers the *(take-object player card hub)* action.

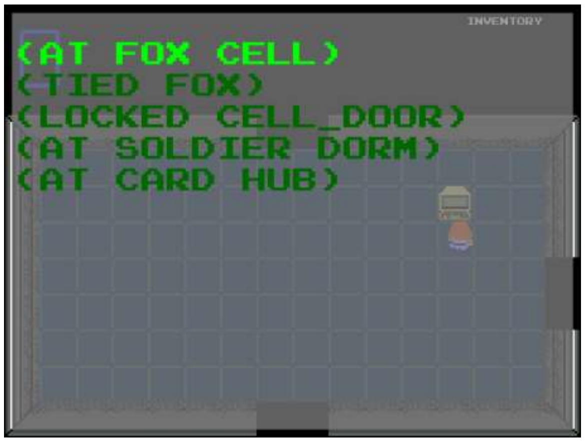
down the matching outgoing edge if the action is enabled, and does nothing if the action is not enabled. Of these, *(open-door player card cell_door)* is an enabled action. The *Mediator* transitions down the outgoing edge which updates the underlying declarative state changing *(locked cell_door)* from true to false. This literal change is tracked by the *StateManager* and used to update the animation property of Unity's *cell_door* object, changing its state from closed to open.

Two other classes, *CameraController* and *InventoryManager*, act as auxillary information relayers for the *Player* class. *CameraController* senses when the player touches the edge of the screen, locks the players movement, moves the camera's focus from the player's previous room to the new room, and restores control to the player. These transitions are similar to movement in *The Legend of Zelda*. When these transitions happen, the *CamerController* notifies the *Mediator* with a fully ground action that the player has moved to a new room. Another class, *InventoryManager*, displays items that exist in the player's inventory at the top of the screen. Figure 5.21 shows the player performing the *(take-object player card hub)* action, the *card* game object from the *hub* being destroyed based on the state update, and a new *card* object being instantiated in the inventory UI strip. The *InventoryManager* updates the UI strip based on the *StateManager*'s current inventory objects. *InventoryManager* also tracks a blue cursor that the player can move when multiple objects exist in the inventory. This cursor allows players to select items in order to bind with context-sensitive actions, like selecting the *card* inventory object in order to bind the second parameter in the *(open-door player card cell_door)* action.

Together, *Player*, *CameraController*, and *InventoryManager* comprise the player architecture that sense potential declarative actions in the game world and format the actions into



(a) UGME gameplay.



(AT FOX CELL)
(TIED FOX)
(LOCKED CELL_DOOR)
(AT SOLDIER DORM)
(AT CARD HUB)

(b) UGME graphical state display.

Figure 5.22: Two gameplay screenshots of the Base Case game, implemented with UGME.

commands for the *Mediator*. The next section discusses a game implementation of the UGME pipeline, called Base Case.

Base Case

Figure 5.22a pictures a demo sneaking game implemented in UGME called Base Case [9]. Base Case is a top-down sneaking game where the player must free their captured comrades from a military compound while avoiding detection by the guards that patrol the base. What makes Base Case unique is that the game is generated using a plan-based experience manager's declarative state transition system. This allows the experience manager to control not only the behavior of the game's NPCs but also the configuration of the procedurally generated game world. In order to escape the base for good, the player must find a special computer console that allows them to modify the underlying predicate state and open an otherwise hidden exit.

Base Case's gameplay begins with an initial game world state specified in PDDL [32]. Using this specification, a plan-based experience manager [55] creates a declarative state transition system and plan to guide the game's NPCs. Based on this state transition system generated by the experience manager, a procedurally generated content pipeline creates an interactive game world and populates it with game objects. The PCG pipeline is implemented with Unity which instantiates and destroys game objects based on the declarative state.

This declarative experience manager + PCG pipeline game engine is called the Unity General Mediation Engine and is used to create Base Case. Base Case is a sneaking game where the player moves through rooms patrolled by enemy NPCs in search of a prisoner. When the prisoner is found, the two must escape the base. When the player escapes the base with the rescued prisoner, the game restarts in a new configuration. This new configuration is one of several pre-authored PDDL problems the system chooses between on initialization. The game continues to repeat as the player explores new configurations of the base.

In order to finish the game, the player must find a special world object made specifically for Base Case that has a special privelege to directly modify the underlying declarative state. In every world state configuration there is a computer that allows the player to directly manipulate the truth value of fully ground predicates describing aspects of the world's state. In order to open the true exit and escape the base for good, the player must find the computer system and use it force the underlying world state into a configuration that will allow the player to access the hidden exit. Once the player escapes from the secret exit, the game stops repeating.

5.3.3 Discussion and Future Work

There are two main differences between conveying mediation tree information through the text interfaces described in Section 5.3.1 and the visual 2D interface described in Section 5.3.2 that drive future work for UGME. The first difference is that time in the text interface is discrete, time stops completely each time it's the player's turn, whereas time in the 2D representation is continuous, it takes time for animations to play out and the player can take action at the same time as NPCs. The turn-based GME backend works best for a discrete interface, but a few modifications would allow GME to accept asynchronous actions from the player and NPCs.

The second difference is that text can act as an almost exact interface for PDDL domain and problem information reasoned over by GME, but the metaphor of visual information is farther away. For example, it's really easy to convey a player's inventory with text, just list a natural language description of each item in a sentence but for a visual game many design decisions must be made on how to lay the list out to the player. The inventory could always be visible like in Base Case or it could be hidden in a menu the player can navigate to by hitting a special button. Furthermore, when the player wants to perform an action with an inventory item in text they can simply type a command with the item's name and it maps pretty simply to a bound domain operator but providing these same bindings through a visual interface requires more design decisions. Allowing GME to make these design decisions for assets, mechanics, and interfaces is the largest piece of remaining future work.

Asynchronous GME

The first component of future work is creating an asynchronous GME backend for games with continuous time. GME currently employs a turn taking system that allows the each character in the story world to take a single PDDL action per round in a consistent series. This setup works well for interfaces with discrete turn taking but less well for interfaces that allow more continuous movement, like most 2D and 3D games. This limitation can be solved by modifying

GME's backend to be asynchronous. That is, instead of having turn layers allow any character to take any enabled action at any time. GME would send instructions to NPCs based on the current plan and instead of immediately transitioning down the action's edge wait until the NPC sends confirmation that they have performed the action in the game world, similar to UGME's *Player* class. This asynchronous system would work better for game worlds with continuous time.

Procedural Asset Generation

The second component of future work is a robust visual procedural content generation pipeline. This pipeline can be broken into two parts: assets and world mechanics. The text-based GME interfaces can be used for a wide range of PDDL domain and problem pairs out of the box because the work it takes to create, animate, and allow interaction in a story world with text is small compared to the work it takes with visual assets. The first step for allowing UGME to produce as many experiences as GME's text interface out of the box is allowing UGME to automatically create visual assets that accurately represent the underlying story world object. For example, if a domain author wants an asset that represents a candle in text GME, they can simply create an object and give it the type *candle* and GME will print out to the player that the object is a candle in its state description. In order for UGME to have this same functionality, it must be able to produce a visual game asset that looks like a candle from the PDDL type. To do this, UGME needs some function that given a PDDL game object description is capable of producing a Unity game object that includes a base visualization as well as animations for each state the object can be in during gameplay.

Procedural Game Mechanic Generation

The second aspect of a fully functioning procedural content generation for UGME would be a game mechanic generator similar to ideas explored in Mark Nelson's work [36]. PDDL mechanics modeled with domain operators easily map to text commands. In a domain where there are rooms connected by doors that the player can navigate between it is easy to convey this information to the player and receive commands through text. If the player is standing in a room, this information can be conveyed to the player with a sentence, and a command of moving to another room can be conveyed to the system with a second sentence from the player. In a visual system there are many ways to visualize a graph of rooms connected by doors. For example, both 2D and 3D games in the *Metroid* and *The Legend of Zelda* serieses match the pattern of navigatable rooms with doors. The second portion of the PCG pipeline to make UGME work generally with PDDL domain and problems is a way of describing game world mechanics and mapping from PDDL domains to game world mechanics that implement those logical constraints in the game space. To do this, UGME needs some function that given a PDDL operator library is capable of producing a game world mechanic configuration that matches the PDDL. Given navigatable room objects connected by doors, the function should map to *Metroid* or *The Legend of Zelda* gameplay mechanics but not *Punch-Out!!*'s.

5.4 Summary

This chapter presented the General Mediation Engine, an implementation of a perceptual experience manager. An overview of the system was given in Section 5.1. Section 5.2 describes the implementation of the GME backend, including how it models PDDL objects, builds mediation trees, and performs perceptual experience management. The GME system architecture is uniquely suited to perceptual experience management due to a procedural content generation pipeline that automatically configures and updates a game world interface based on the underlying PDDL state and update mechanics. Section 5.3 described a series of text and visual interfaces that automatically configure themselves to represent state and update information from the underlying mediation tree. The next chapter presents a series of evaluations that test the research questions laid out in Chapter 1.

Chapter 6

Evaluation

This chapter presents a series of four evaluations that address the two research questions presented in Chapter 1. The first evaluation uses GME to generate data structures that address the first research question. The evaluation expands perceptual interactive narrative trees for a set of example PDDL problems and compares them to a baseline tree generated using accommodation alone. Since event and domain revision only occur when accommodation fails to find a solution, perceptual interactive narrative trees should have as many or more nodes per level when compared to a traditional accommodation tree. This hypothesis is tested by expanding trees in a breadth-first order and recording the number of nodes at each tree level. The results for each test provide evidence for the hypothesis.

The next three evaluations use human subjects gathered from Amazon's Mechanical Turk service to address the second research question. These studies are designed to test whether players notice when perceptual experience manager manipulations are performed unconditioned by a model of player knowledge. The first experiment is a Choose Your Own Adventure with three different versions, each testing whether participants notice domain revisions unconditioned by player knowledge. Surprisingly, the results show no difference between participant reactions to action outcomes in the baseline with no modifications, the intervention version with unconditioned modifications, and the domain revision version with modifications bounded by player knowledge.

The final two evaluations were designed as follow ups to the second evaluation. The third test was again on whether human participants would notice when domain revision was used unconditioned by a model of player knowledge. However, instead of measuring feedback with survey questions, the study instead records reading time as a measure of reading comprehension. Prior work in the reading comprehension literature shows that heightened reading times occur when participants must reconcile a statement that is inconsistent with past text. The study shows participants have a similar reading time increase in an interactive context when the inconsistency is introduced by domain revision unconditioned on player knowledge. The fourth study takes this principle and applies it to event revision with similar positive results.

6.1 Research Questions Revisited

Chapter 1 lays out two research questions based on this document's thesis statement that have guided the work presented in this document. The questions were:

1. How can story world manipulations be used to increase the number of interactive narrative branches consistent with authorial constraints?
2. How can a model of character knowledge be used to prevent story participants from noticing story world manipulations?

This chapter provides evaluations that show the perceptual experience management framework presented by this document answers both research questions laid out in the introduction. First, Section 6.2 evaluates perceptual experience management's ability to answer the first question by comparing interactive narrative trees generated by event and domain revision against a baseline of regular accommodative mediation. Since event and domain revision are only used when accommodation cannot create a new branch in line with author constraints there should be an equal or greater number of nodes in line with author constraints in each level of a perceptual interactive narrative tree when compared to a baseline when the two trees are expanded in a breadth first order.

Next, Sections 6.3, 6.4, and 6.5 present a series of three human subjects experiments that evaluate perceptual experience management's ability to answer the second question by measuring whether human participants notice manipulated events when unconstrained by a model of player knowledge. Section 6.3 shows that domain revision is not noticed by human participants, but casts doubt on whether participants notice unconstrained domain manipulations. Sections 6.4 and 6.5 control elements present in the first study and show that participants notice unconstrained domain and event modifications in short, single choice stories. Together, these four evaluations address the two research questions and provide evidence that this document's perceptual experience management framework fulfills the claims made by the thesis statement laid out in Chapter 1.

6.2 Structural Evaluation

This section presents an evaluation of event revision and domain revision as implemented in the General Mediation Engine by comparing metrics recorded from a breadth first expansion

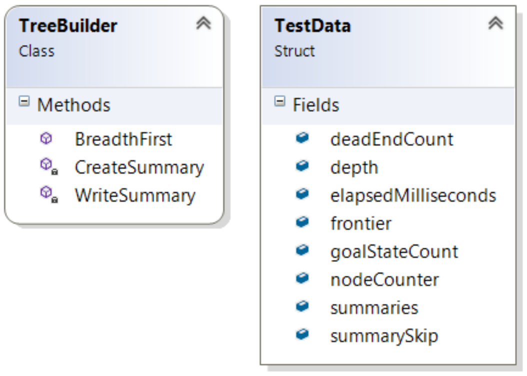


Figure 6.1: Diagram of GME's data collection suite. The diagram is generated by Microsoft Visual Studio 2015. GME's TreeBuilder creates a mediation tree using breadth first expansion to a given depth and writes tree statistics out to disk in the form of CSV spreadsheets. TestData records data for each tree building session.

of four interactive narrative trees. GME's suite for building trees and comparing data is shown in Figure 6.1. The *TreeBuilder* class provides the *BreadthFirst* method that takes as input a PDDL domain and problem, a test depth, and whether domain revision, event revision, both, or neither will be enabled. The method then uses the *MediationTree* suite to expand a tree in a depth first order and maintain statistics on the tree using the *TestData* struct. Once the given depth has been reached, *TreeBuilder* uses the *CreateSummary* and *WriteSummary* methods to create a spreadsheet of information about the tree from the *TestData* struct and write the sheet to disk. The *nodeCounter* entry in each spreadsheet, updated every time an outgoing edge is expanded and its child node is created, is the main metric this section uses to compare perceptual experience management trees to a baseline. *TreeBuilder* can also generate visualizations of its mediation trees using the Graphviz [15] graph vizualization software.

The first tree in each test is generated with GME's baseline accommodative mediation method. The baseline tree's metrics are then compared against a second tree that is generated with either event revision, domain revision, or both event and domain revision working together. Because event and domain revision monotonically add nodes to the baseline mediation game tree per level, we can expect the number of nodes in trees generated using these methods to always be equal to or more than the number of nodes per level in trees built with the baseline method. So, for each individual test our hypothesis is that the event revision, domain revision, or tree of both will have equal or more expanded nodes per level than the corresponding baseline tree that only uses accommodation. Sections 6.2.1, 6.2.2, and 6.2.3 all provide evidence that supports this hypothesis by examining trees built from three different PDDL domain and problem pairs set in a Batman, Wild West, and Spy world.

(a) Rachel (b) Batman (c) Harvey

Figure 6.2: The three main characters in the Batman domain.

6.2.1 Batman Domain

The first test domain is a situation set in a Batman universe. The problem first appeared as an example for event revision in the context POCL mediation [53, 54]. The Batman domain models a situation in the film *The Dark Knight* where Rachel Dawes and Harvey Dent are kidnapped by the Joker, who forces Batman to choose between saving them. Figure 6.2 shows the main characters in the Batman domain.

Problem Description

This problem is set in Gotham City. The Joker has been captured by the police and is being held at the Gotham Police Department. The player is Batman who is about to confront the Joker. The Joker's henchmen have kidnapped two important characters, Rachel Dawes and Harvey Dent, and are transporting the characters to hostage locations. During the confrontation, the Joker tells Batman about his hostages. The Joker says the hostages will be killed and Batman has time to only save one. In order to set up the final act where Harvey Dent is turned into Two Face, the system has the goal of Batman saving Harvey at the expense of Rachel. For this to happen, the system has the henchmen deliver Rachel to 52^{nd} Street and Harvey to Avenue X. The Joker tells Batman where the characters are and Batman decides who to save. In the film, Batman travels to Avenue X where he saves Harvey.

However, these world mechanics allow Batman to travel to 52^{nd} Street and save Rachel, which would kill Harvey and end the possibility of a final act with Two Face. If event revision is used the past events of the henchmen transporting Rachel and Harvey to their destinations are identified as outside the players knowledge of the world history. Once these events are identified they can be removed and replaced with the alternate perceptually consistent event sequence of the henchmen delivering Rachel to Avenue X and Harvey to 52^{nd} Street. In this world history, Joker lies about Rachel and Harvey's locations in order to reverse Batman's decision. Figure 6.3 illustrates the placement of Rachel and Harvey by the henchman retroactively by event revision once Batman makes his choice.

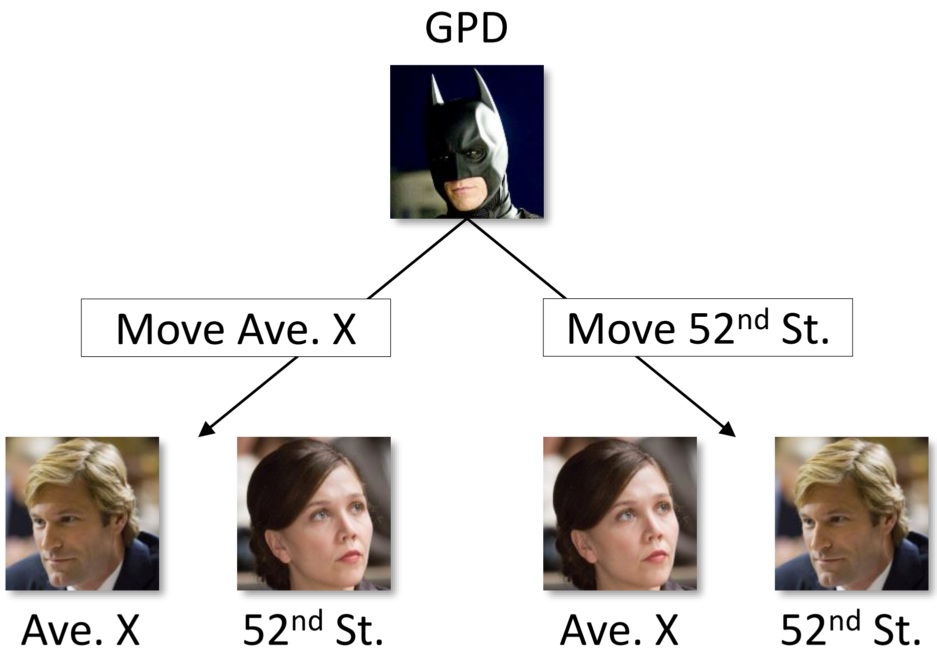
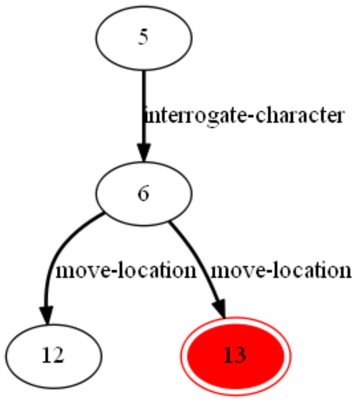


Figure 6.3: Event revision's ability to change the situation after receiving the player's action.

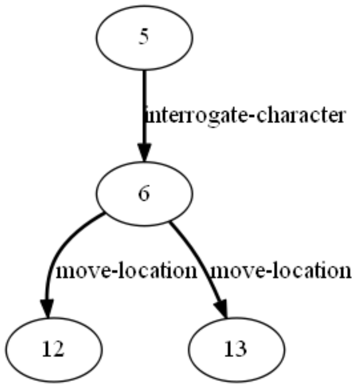
A PDDL description of the example is given in Appendix A sections A.2.1 and A.2.2. Section A.2.2 presents the PDDL problem, which contains a description of the initial world state and the goal state. Initially, Batman and Joker are at Gotham Police Department, the henchmen are in cars with Rachel and Harvey, who are both alive. The system's goal is to achieve a state where Harvey has been saved by Batman and Rachel is alive. Section A.2.1 presents the PDDL domain, which contains a description of the possible actions characters can take in the story world. Characters can move between locations if the locations are connected. Characters can interrogate apprehended characters. Victims can be placed at hostage locations. Finally, hostage victims can be saved by characters.

Test Results

GME generated two interactive narrative trees for the Batman Domain using breadth first search. The first tree was generated with event revision and the second tree with the baseline accommodation mediation method. The trees were generated from the example PDDL presented in Sections A.2.1 and A.2.2. The hypothesis is that GME will generate equal or more nodes per level in its event revision tree when compared to its accommodation tree. Since event revision allows states to remain on a valid path to a goal leaf when they would otherwise become a dead end, the number of nodes in each level of the event revision tree should always be equal to or



(a) Baseline Tree



(b) Event Revision Tree

Figure 6.4: Graph of player choices in a baseline and an event revision tree automatically created by GME from the Batman domain. The trees show only player actions, not planned NPC behaviors. States are indicated with circles. Player actions are indicated with directed edges. A dead end is indicated with an outlined, red circle.

greater than the number in the corresponding level in the baseline tree. The first 18 levels of each tree were generated, the point where a goal node is first encountered. As expected, event revision produces 23 valid nodes as opposed to the baseline method's 17 nodes. Level 12 is the level at which the player as Batman makes their decision whether to move to Avenue X or 52$^{\text{nd}}$ Street. At this point until the story ends at Level 18 the event revision tree has 100% more branches (2 to 1) when compared to the baseline tree. Figure 6.4 shows a single dead end is reached in the accommodation tree after the second player choice, where Rachel is saved, but no dead ends are reached in the event revision tree.

6.2.2 Wild West Domain

This section describes a domain and problem set in a generic Wild West story used to test domain revision. It describes the Wild West domain, problem, and provides test results that confirm domain revision performs better than a baseline accommodative mediation approach.

Problem Description

This domain is set in the Wild West. The player is a gun for hire who has been captured by a band of thieves. The thieves are holding the player prisoner in a small house outside the gang's main compound. The player's hands are tied by rope and a single bandit stands guard outside the house by a bonfire. Both the player and the guard have a concealed knife. The system's goal is to solidify trust between the player and a potential love interest who works with the thieves.

The player met the love interest earlier in the game and was surprised to find the love interest at the bandit camp before the player was captured. To solidify this trust, the system plans for the love interest to apprehend the guard, tie the guard's hands with rope, and release the player. The system would also like the player and love interest to escape together and return to the nearest town. To create this situation, the system plans for the guard to cut his bonds with the concealed knife, return to the compound, and warn his fellow bandits.

However, these world mechanics allow the player to cut their own bonds with a concealed knife and escape without the love interest. Using intervention, the system could prevent the player from cutting through the rope by replacing the original effect with one that does not update the world state. It's not unreasonable to show the player being unable to reach the knife or cut the rope. But if the player emerges from the house with their love interest to find ropes cut by the guard in order to escape, they might realize the system has created a double standard in which ropes can be cut by bound persons, but only when the system allows it. If domain revision is used, the dynamic conditional effect that models the rule - bound persons with a concealed knife can cut through their bonds - is erased from the domain when the intervention is used against the player. From that point on, no character would be able to free themselves with a concealed knife when bound with rope. Instead, the system would need to find some other way to free the guard so he could warn the other bandits. For example, he might instead burn his bonds using the bonfire and leave behind charred rope for the player to find. In this way, the system is free to use interventions and ensure the world continues to behave consistently from the player's perspective.

A PDDL description of the example is given in Appendix A sections A.5.1 and A.5.2. Section A.5.2 presents the PDDL problem, which contains a description of the initial world state and the goal state. Initially, the player is tied at the house and has a knife, the guard is outside by a bonfire with a second knife, and the love interest is at the bandit camp with a rope. The system's goal is to achieve a state where the guard escaped capture and is at the bandit camp, the rope holding the guard is now outside by the bonfire, and the love interest has rescued the player. Section A.5.1 presents the PDDL domain, which contains a description of the possible actions characters can take in the story world. Characters can move between two locations if they are connected and the character is not tied. A character can untie another character if they are co-located and the first character is not also tied. A character can tie another character if they are co-located and the first character has a rope. A character can burn their bonds if they are tied and co-located with a fire. Finally, a character can cut their bonds if they are tied and have a knife. The effects of this final action are dynamic conditional effects that can be removed by domain revision. The effects have an empty set of additional preconditions.

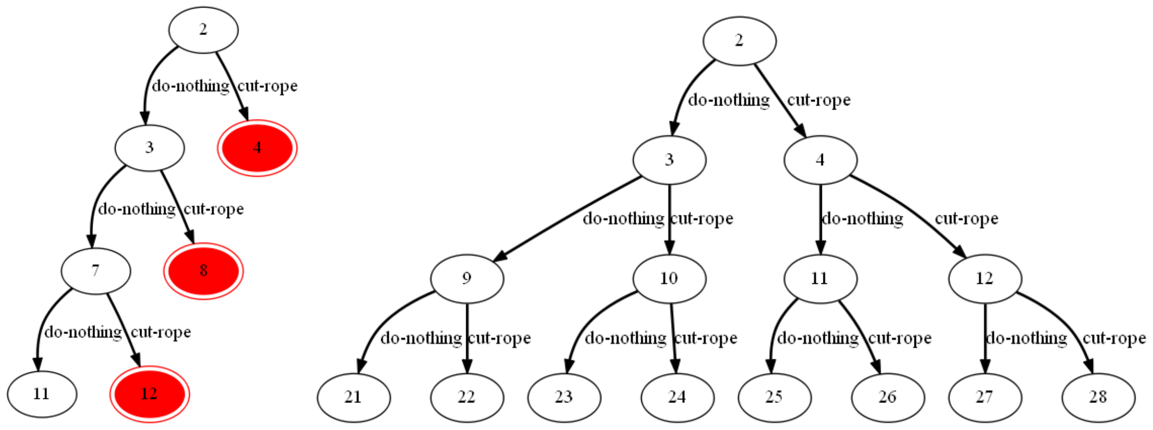


Figure 6.5: Graph of player choices in a baseline and a domain revision tree automatically created by GME from the Wild West domain. The trees show only player actions, not planned NPC behaviors. States are indicated with circles. Player actions are indicated with directed edges. A dead end is indicated with an outlined, red circle.

Test Results

GME generated two interactive narrative trees for the Wild West Domain using breadth first search. The first tree was generated with domain revision and the second tree with a baseline mediation method. The trees were generated from the example PDDL presented in Sections A.5.1 and A.5.2. The hypothesis is that GME will generate equal or more nodes per level in its domain revision tree when compared to its accommodation tree. Since domain revision allows states to remain on a valid path to a goal leaf when they would otherwise become a dead end, the number of nodes in each level of the domain revision tree should always be equal to or greater than the number in the corresponding level in the baseline tree. The first 11 levels of each tree were generated, the point where a goal node is first encountered. As expected, domain revision produces 36 valid nodes as opposed to the baseline method's 10 nodes. Levels 3, 6, and 9 are where the player makes a decision whether to sit still or make an escape. After Level 3 the domain revision tree has 100% more branches (2 to 1), after Level 6 the domain revision tree has 400% more branches (4 to 1), and after Level 9 the domain revision tree has 800% more branches (8 to 1) when compared to the baseline tree. Figure 6.5 shows a single dead end is reached in the accommodation tree at Level 3, 6, and 9, where the player has an opportunity to escape, but no dead ends are reached in the domain revision tree no matter what the player chooses to do.

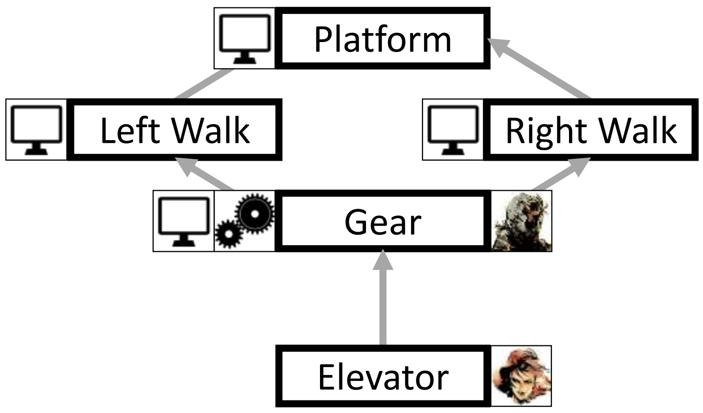


Figure 6.6: A diagram of the initial configuration of the *Spy* world given as the initial state in Appendix A.4.2. Rectangles are rooms, grey arrows are one-way connections between rooms, and squares are world objects in each room.

6.2.3 Spy Domain

This subsection presents the description and test results of a problem set in the Spy Domain. The Spy Domain is loosely based on the *Metal Gear* series and the final level of *GoldenEye 007* for the Nintendo 64 and was presented as an example of perceptual experience management mechanics in Section 4.6. The Spy Domain problem contains examples of both domain and event revision.

Problem Description

The Spy Domain is meant to represent a sneaking game and is modeled as a cross of *GoldenEye 007* and *Metal Gear*. The *Spy* game is an example world where the player, as a spy named Snake, must foil the final attempt of the computer-controlled antagonist, the Boss, to bring a weaponized satellite online. The confrontation takes place on a satellite dish antenna cradle with five discrete locations where the Snake and Boss can interact: the Elevator Room, Gear Room, Left and Right Walkways, and the Platform. The locations are connected by doors that can only be traversed in one direction. The initial world layout is pictured in Figure 6.6 using icons for world objects pictured in Figure 4.8. In the initial state figure, locations are labeled rectangles and doors are arrows. The doors can only be traversed in the direction arrows are facing. Snake begins the game in the Elevator Room. Her job is to disable the satellite dish's alignment mechanism in the Gear Room and eliminate the Boss. The Boss is trying to send instructions from his phone to the satellite by linking the phone to one of four computer terminals on the cradle. The domain author wants the Boss to build and set a trap to be disabled by the player before a final confrontation between the two on the Platform. These authorial constraints are coded as conditions in the PDDL goal state.

Snake starts off in the Elevator Room and the Boss begins in the Gear Room. Snake has a pistol (PP7) an explosive (C4) and a detonator for the explosive. The Boss has a laser rifle and a trip-wire for building a trap, and a phone that can be linked to the satellite through an activated computer terminal. There is a computer terminal on the Platform, Left and Right walkways, and in the Gear Room. Any of these four computer terminals can be activated by the Boss and used to send information to the satellite from his phone. This initial world configuration is shown in Figure 6.6. The game progresses by alternating between allowing the Boss and Snake to take an action that updates state information. The Boss is controlled by plots generated by the system's planner and connected through the experience manager's mediator. Snake is controlled by a player. The game continues until a goal state is reached or the author's constraints are broken.

Table 6.1: Spy Domain breadth first expansion node counts for the baseline accommodative tree, event revision tree, domain revision tree, and combined tree from Level 1 to 15.

Depth	Baseline	Domain	Event	Domain + Event
1	1	1	1	1
3	2	2	2	2
5	6	6	6	6
7	20	21	21	22
9	67	74	75	82
11	220	255	270	305
13	707	860	896	1049
15	2230	2852	2873	3497
Total	4276	5290	5415	6431

Table 6.2: Spy Domain percentage increase in branches over baseline accommodative tree for event revision, domain revision, and combined tree from Level 7 to 15.

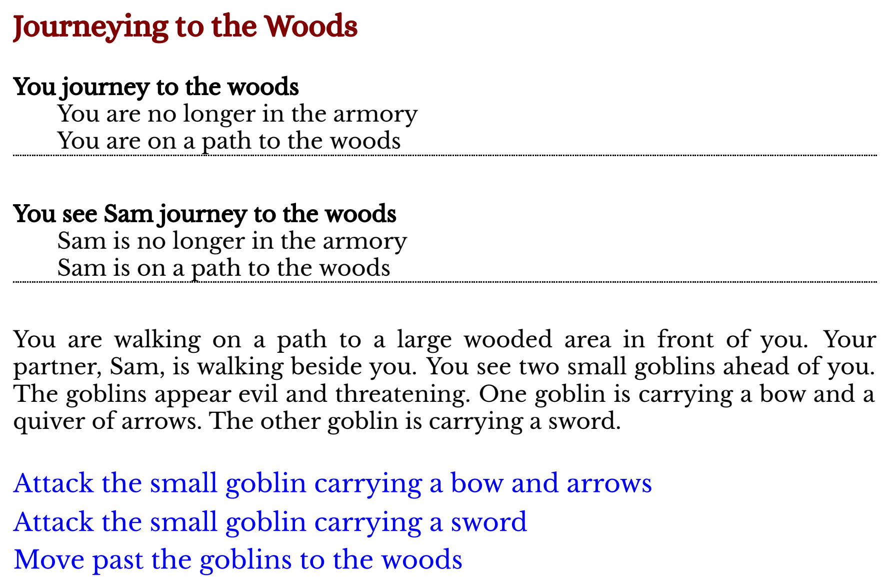
Depth	Domain	Event	Domain + Event
7	5	5	10
9	10	12	22
11	16	23	39
13	22	27	48
15	28	29	57

Test Results

GME generated four interactive narrative trees for the Spy Domain using breadth first search. The first tree was generated with domain revision, the second with event revision, the third with both, and the fourth tree with a baseline mediation method. The trees were generated from the example PDDL presented in Sections A.4.1 and A.4.2. The hypothesis is that GME will generate equal or more nodes per level in its event and domain revision trees when compared to its accommodation tree. Since event and domain revision allow states to remain on a valid path to a goal leaf when they would otherwise become a dead end, the number of nodes in each level of the event and domain revision trees should always be equal to or greater than the number in the corresponding level in the baseline tree. The first 15 levels of each tree were generated, past the point where a goal node is first encountered. The results are shown in Table 6.1. As expected, domain revision produces 5290 valid nodes, event revision produces 5415 valid nodes, and both produce 6431 valid nodes as opposed to the baseline method's 4276 nodes. Table 6.2 shows the percentage increase in valid branches from levels 7 to 15 in the domain, event, and combined trees when compared to the baseline tree. The percent increase in branches grows over time in each tree and reaches over a 50% increase in the combined tree at Level 15. This section concludes the structural evaluation. The next section describes the first of three human subjects evaluations meant to validate this document's second research question.

6.3 Human Subjects Evaluation 1

This section presents an evaluation of domain revision in the context of a Choose Your Own Adventure. The purpose of the experiment is to answer the second research question, whether a model of player knowledge can prevent participants from noticing when story world manipulations take place. This experiment evaluates domain revision modifications in the form of choice outcomes against two baselines. The first baseline is a story with no choice outcome manipulation and the second baseline is a story with a choice outcome manipulation unconstrained on user knowledge. The study also examines whether players notice a choice option manipulation when compared against two baselines. Participants were recruited through Amazon Mechanical Turk to play the CYOA on the Internet. Participants were asked survey questions about how believable choice options and choice outcomes were once the CYOA was over. It is expected that domain revision constrained on a model of user knowledge will be rated higher by user feedback on the believability of choice options and outcomes than the alternative methods, choice removal and intervention, which modify the story world against the player's expectations. The rest of this section describes the experiment in detail, beginning with the three treatments, moving to a description of the user survey, the hypotheses, the results, and ends with a discussion of the experiment that sets up the evaluations described in the next two sections.


Journeying to the Woods

You journey to the woods
You are no longer in the armory
You are on a path to the woods

You see Sam journey to the woods
Sam is no longer in the armory
Sam is on a path to the woods

You are walking on a path to a large wooded area in front of you. Your partner, Sam, is walking beside you. You see two small goblins ahead of you. The goblins appear evil and threatening. One goblin is carrying a bow and a quiver of arrows. The other goblin is carrying a sword.

Attack the small goblin carrying a bow and arrows
Attack the small goblin carrying a sword
Move past the goblins to the woods

Figure 6.7: A scene from the introductory segment of the CYOA. Outcomes from each character's last action are displayed at the top of the page. A paragraph describing the current situation is presented in the middle. The choices available to the player are listed at the bottom as hyperlinks that lead to the next page of content.

6.3.1 Materials

Figure 6.7 shows a screen from the CYOA game. The game is hand authored for speed and to reduce the possibility of bugs, but its interface is designed to look like the online GME CYOA interface, presented in Section 5.13, and function like the GME system. Like GME's text architecture, results from the last choice the player made, along with the actions and outcomes of actions performed by other story characters, are listed at the top of the page. A middle paragraph describes the current state of the world. A list of possible actions are listed as hyperlinks at the bottom of the page. When the participant clicks one of the hyperlinks, the story advances to a page containing the next set of choice outcomes, the new situation, and the new choice options. Study participants play through one of three story treatments, each a variation of a fantasy story where the player embarks on a quest with their partner Sam to confront an evil goblin.

Figure 6.8 shows a diagram of each of the three treatments used in the experiment. Each of the three treatments begin with the same introduction sequence, pictured in Figure 6.8a. Each story begins in an armory. The player is with their companion, Sam, and is tasked with vanquishing an evil goblin that resides in the woods near town. The player can choose to arm themselves with a bow, a sword, or proceed immediately to the woods. On the way to the woods, the player is confronted by two small goblins, one carrying a bow and one carrying a sword. The player can attack either of the goblins or proceed past the goblins to the woods. This is the choice depicted in Figure 6.7. Once the player reaches the woods, they find a large goblin wearing armor. At this point, the three versions of the story begin to differ. All three versions share the same three authorial constraints: the armored goblin should vanquish Sam in the woods, the player should be in the woods when Sam is vanquished, and the player should vanquish the armored goblin at the clearing beyond the woods.

Choice Removal

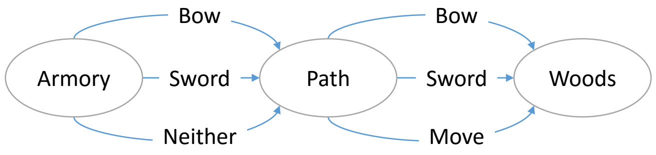
The choice removal treatment, pictured in Figure 6.8b, is a version of the story where choice options are taken away from the player. The treatment forces the player to wait a turn in the woods since both attacking the goblin and moving to the clearing would contradict authorial constraints. As the player waits, the goblin attacks and vanquishes their companion Sam. The system then forces the player to move to the clearing beyond the woods by removing the attack action, since attacking the goblin in the forest would contradict the constraints. Finally, the player attacks and vanquishes the large goblin in the clearing beyond the woods. The successful attack outcome is pictured in Figure 6.9b.

Intervention

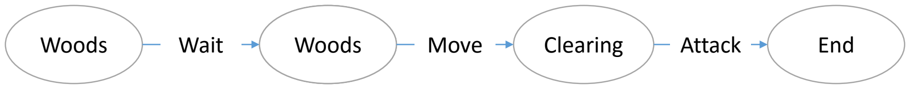
The intervention treatment, pictured in Figure 6.8c, is a version of the story where the player's action outcomes are modified. The treatment allows the player to attack or move past the large, armored goblin in the woods. However, if the player attempts to attack the goblin, their attack is intervened against and fails. The failure outcome is pictured in Figure 6.9a. The CYOA tells the player not only that their attack fails to pierce the goblin's armor, but that the goblin's armor will always be resistant to player attacks. If the player attempts to move, the action will also be intervened against by the system by the goblin blocking the player's way. Once the large goblin vanquishes Sam, the player has another opportunity to attack the goblin or move around the goblin to the clearing. If the player attacks the goblin, they experience a second intervention before moving to the clearing. If they choose to move around the goblin, they go directly to the clearing. At the clearing, the player attacks the goblin and their attack succeeds. The successful attack outcome is pictured in Figure 6.9b.

Domain Revision

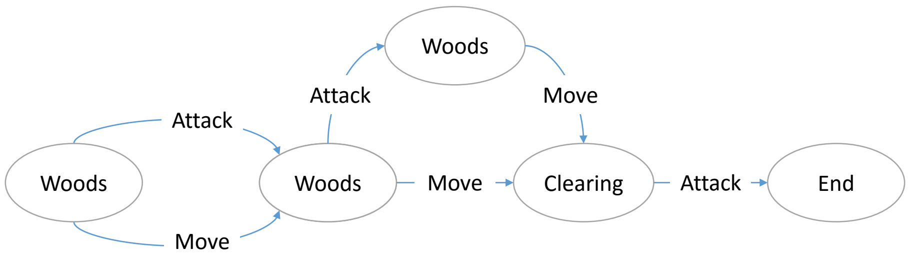
The domain revision treatment, pictured in Figure 6.8d, allows the player to attack or move past the large, armored goblin in the woods. If the player attempts to attack the goblin, as in the intervention treatment, their attack is intervened against and fails. The failure outcome is pictured in Figure 6.9a. If the player attempts to move, their action will also be intervened



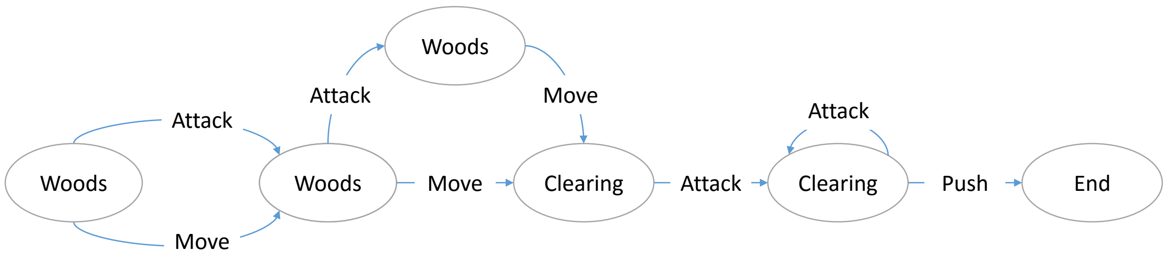
(a) The introduction sequence of the CYOA, constant across the three test versions.



(b) An overview of the removal version of the experiment.

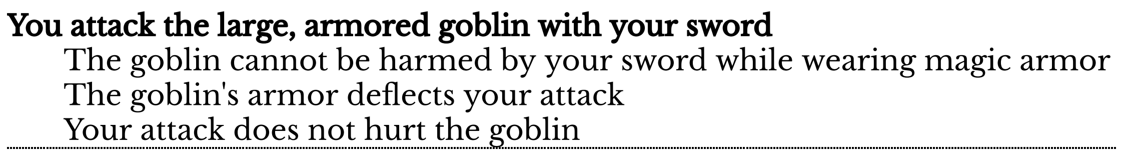


(c) An overview of the intervention version of the experiment.

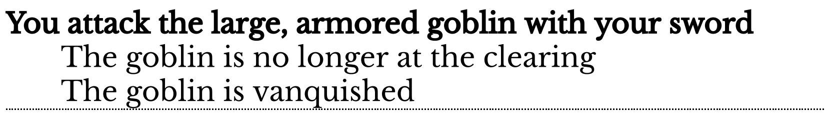


(d) An overview of the revision version of the experiment.

Figure 6.8: The flow of choice framings and options in the three versions of the CYOA. Figure 6.8a pictures the two choices used across all three story versions. Figures 6.8b, 6.8c, and 6.8d picture the choice framings and options from the introduction onward in each of the three versions.


You attack the large, armored goblin with your sword
The goblin cannot be harmed by your sword while wearing magic armor
The goblin's armor deflects your attack
Your attack does not hurt the goblin

(a) Unsuccesful attack on armored goblin.


You attack the large, armored goblin with your sword
The goblin is no longer at the clearing
The goblin is vanquished

(b) Succesful attack on armored goblin.

Figure 6.9: The two types of outcomes of attack actions on the armored goblin. Figure 6.9a pictures the feedback from unsuccessfully attacking the armored goblin. The game tells the player that the goblin cannot be harmed by their weapon while it wears magic armor. Figure 6.9b pictures the feedback from successfully attacking the armored goblin. The game tells the player that the goblin is vanquished by their weapon.

against by the system by the goblin blocking the player's way. Things proceed exactly the same in the domain revision treatment as the intervention treatment until the player arrives at the clearing. When the player attacks the armored goblin at the clearing, their attack continues to be deflected by the goblin's armor. After the player attacks the goblin, they are given the option to push the goblin off the cliff at the edge of the clearing. Pushing the goblin off the cliff vanquishes the goblin without using the player's weapon.

6.3.2 Procedure

After the participants played through the CYOA, they were asked survey questions about their choices and choice outcomes. For the choice questions, the player was presented with each story situation and set of choice options they encountered while playing the CYOA. For each choice situation-option set pair, participants were asked to rate their agreement with two statements on a 5-point Likert scale from "Strongly Disagree" to "Strongly Agree". The statements were:

1. These are reasonable choices for this situation.
2. These are the choices I would expect.

Below the first two statements, the survey shows the player the choice they made and the outcome of their choice. Under the choice outcome, the survey asks participants to rate three additional statements concerning the outcome:

Question	Hypothesis	p	r
1	Intervention > Removal	2.9e-08	0.68
2	Intervention > Removal	7.2e-08	0.66
1	Revision > Removal	2.4e-10	0.75
2	Revision > Removal	4.2e-08	0.67

Table 6.3: Results of MWW U tests on choice option results. All four hypotheses are statistically highly significant ($p < .001$) and have a large effect size ($r > 0.5$).

1. I understand why my choice caused these outcomes.
2. These are reasonable outcomes for this choice.
3. These are all the outcomes I would expect.

6.3.3 Hypotheses

There are two locations in the CYOA where hypotheses are made and tested. For the believability of choice options, the choice framing and options when the player first encounters the large, armored goblin in the woods is examined. The choice removal treatment forces the player to wait a turn, where the intervention and domain revision treatments offer the attack and move options the player experiences earlier on their way to the woods. The hypothesis is players will rate the set of choice options in the intervention and domain revision treatments higher on the two survey questions (reasonableness, expectedness) than the set of choice options in the choice removal treatment. For believability of choice outcomes, the outcome of the player attacking the armored goblin at the clearing is examined. This is the first attack outcome a choice removal participant encounters. The intervention and domain revision treatments previously tell players they will never be able to harm the goblin wearing magic armor. The outcome of the choice removal and intervention treatments is a successful attack, which is consistent with previous feedback in the removal case but inconsistent in the intervention case. The outcome of the domain revision treatment is an unsuccessful attack, which is consistent with previous feedback. The hypothesis is players will rate outcomes in the choice removal and domain revision treatments higher on the three survey questions (understandability, reasonableness, expectedness) than outcomes in the intervention treatment.

A Mann-Whitney-Wilcoxon (MWW) U test [27, 64] is used to determine whether a statement is more agreed with in one treatment when compared to another.

Question	Hypothesis	p
1	Removal > Intervention	0.27
2	Removal > Intervention	0.18
3	Removal > Intervention	0.98
1	Revision > Intervention	0.94
2	Revision > Intervention	0.43
3	Revision > Intervention	0.95

Table 6.4: Results of MWW U tests on choice outcome results. None of the six hypotheses are statistically significant ($p < .05$).

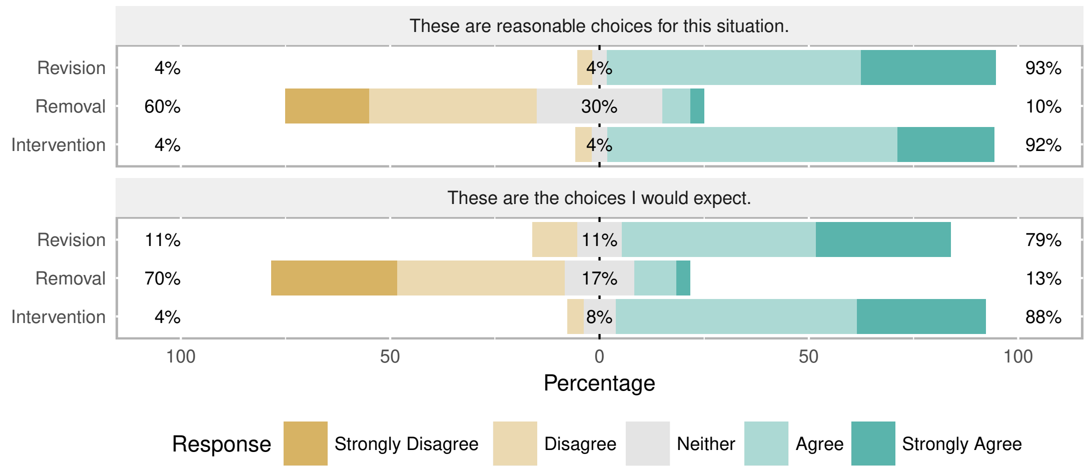
6.3.4 Results

Ninety subjects participated in the experiment. After filtering out inattentive subjects that incorrectly answered a trick question (Please answer this question "Strongly Disagree") and subjects that went down unusable paths (never attacking the armored goblin in the woods), data from 30 participants was collected in the removal treatment, 27 in the intervention treatment, and 28 in the revision treatment for the choice option statements. For the outcome statements, data from 30 participants was collected in the removal treatment, 24 in the intervention treatment, and 21 in the revision treatment.

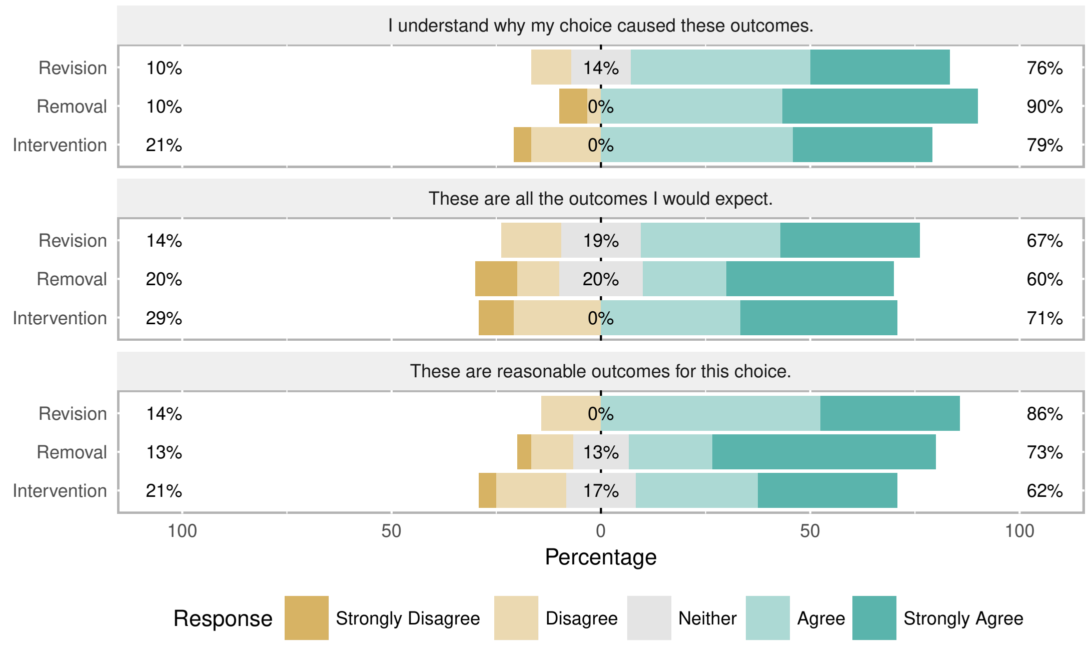
The data gathered support the choice option hypotheses, that intervention and domain revision would be rated higher than choice removal in responses to the choice option Likert survey. A summary of the data is shown in Table 6.3 and a graph of the data is shown in Figure 6.10a. The choice removal treatment's set of choice options for the first encounter with the armored goblin in the woods is rated lower on both statements than the intervention and revision treatments. All four hypotheses have p-values under .01 and effect sizes greater than 50%. More interestingly, the data do not support any choice outcome hypotheses, that choice removal and domain revision would be rated higher than intervention in responses to the choice outcome Likert survey. A summary of the data is shown in Table 6.4 and a graph of the data is shown in Figure 6.10b. The hypotheses are not supported because the intervention treatment is rated to be just as believable as the choice removal and domain revision treatments, even though event outcomes in the intervention story violate earlier player observations about story world mechanics.

6.3.5 Discussion

The data do not support the hypothesis that story world inconsistencies introduced by domain revision unconstrained on a model of player knowledge are easily detected by interactive narra-



(a) Choice option results.



(b) Choice outcome results.

Figure 6.10: Graphs of Likert response data for choice option and choice outcome results. Figure 6.10a shows a graph of choice option results for the three versions of the test. As predicted, the removal case performs worse than the intervention and revision cases. Figure 6.10b shows a graph of choice outcome results for the three versions of the test. Unexpectedly, the intervention version performs just as well as removal and revision.

tive participants. However, there are a number of confounding factors that could account for the negative data. The first confounding factor is that the effect could exist but its effect size could be small enough that it went undetected in this data set. Only 30 participants were recruited for each treatment. These data were enough to provide large effect sizes for the choice option hypotheses but it could be that choice outcomes are harder for participants to notice which would lower the effect size and require more data to detect. The second confounding factor is that participants may not have had enough time to learn the effect rule that attacking a goblin wearing magic armor results in a deflected attack since they only saw the outcome once. If players didn't have enough time to learn the rule, they wouldn't notice when the rule was violated later in the story. A third confounding factor is the story takes place in a fantasy world where goblins and magical armor exist. Even if the player learns the rule, they may reason that some magical element caused the attack to succeed. The final confounding factor is this study relies on participants to self report whether they thought outcomes were believable as a measure for whether they noticed an inconsistency. It could be that players registered the inconsistent outcome but quickly moved on or rationalized that the inconsistent outcome was still believable during the survey.

The next section presents a follow up experiment designed to address the four confounding factors of this original study. First, each study participant plays seven short stories in the new design instead of the original single story. This increases the data per participant by a factor of seven allowing for a much larger data set. Second, instead of expecting participants to learn a rule that is unique to the story world like magic armor repels attacks, the followup study produces inconsistencies using rules from the real world like snake oil can't heal people or a bad shot can't hit a hard target. Next, instead of setting the story in a fantasy world where magic can happen, the followup study uses a Wild West setting closely grounded in the mechanics of our real world. Finally, instead of relying on players to self report how believable the outcomes of actions are, the followup study directly measures their reactions to outcomes by measuring the time it takes for them to read each sentence in each story.

6.4 Human Subjects Evaluation 2

This section presents an evaluation of domain revision in the context of short, one-choice Choose Your Own Adventures. The purpose of the experiment is to measure whether human participants notice the outcomes of domain revision actions when they conflict with the system's model of player knowledge. When a revision takes place that breaks some previously established world rule, it introduces a logical inconsistency into the story world that should be noticed by a participant. These are the types of interventions domain revision avoids. Work done in the area of reading comprehension [39, 1] provides evidence that readers are sensitive to logical inconsistencies in the context of short stories. These inconsistencies impact reading comprehension by breaking story coherence, which can be measured through reading time. Previous work shows an increase in the average amount of time it takes participants to read a sentence that introduces an inconsistency compared to the same sentence when it is consistent with the story.

In this study there are two types of short CYOA stories: those with *consistent* action outcomes and those with *inconsistent* action outcomes. The consistent action outcomes are modeled on worlds where domain revision maintains a single set of mechanics throughout an entire story. The inconsistent action outcomes are modeled on worlds where domain revision is free to move back and forth between two sets of contradicting outcomes for the same action. Participants should take longer on average to read inconsistent outcomes when compared with consistent outcomes. This delay should account for the time participants need to reconcile conflicting world models in their head while reading inconsistent stories.

6.4.1 Materials

Each of the stories used in the experiment take place in a Wild West setting and are comprised of five parts. A full list of the stories used in this experiment is given in Appendix B.1. The stories begin with an introduction that describes who the participant's character is and what they do. As an example, here is the introduction for one of the stories used in the study:

Introduction You sell snake oil liniment to the men and women of the pioneer. You travel near and far in your wagon selling what you pitch as a magical, cure-all elixir.

Once the introduction has established the participant's character and occupation, information is given that creates an expectation about the participant's character and actions.

Differentiating Information Of course, your elixirs don't actually cure anything. You make the concoction from mineral oil, red pepper, and turpentine. No matter what you claim, your mixtures can't heal anything. In fact, they usually make people sick.

This differentiating information section corresponds to an initial intervention that sets up an expectation about an outcome of a future action. In each story, there are two models of world mechanics. In World A, which corresponds to the domain's default operator, snake oil elixirs have magical properties that cure story characters. In World B, which corresponds to a failure mode that could be used by domain revision, snake oil elixirs are inert and do not cure story characters. The expectation created by the differentiating information section is that the participant exists in World B. In this instance, if the participant ever treats an illness in the story world with their snake oil elixir, the participant should expect the elixir to be inert and have no effect (or actually be detrimental to the patient's health). Once this World B expectation has been created, a situation is described in which the participant makes a choice.

Choice Frame Today you arrive at a small town called Slate. You hear gunshots as you pull in to town. A young woman runs over to your wagon and implores you to bring elixir to the local tavern. As you enter the tavern with a bottle of elixir, you see the local sheriff and a young man laying on the floor. Each man has a fatal bullet wound in their stomach. The young woman asks you to save them with your elixir.

At this point, the participant is presented with their choice.

Choice Save the sheriff. Save the young man.

Each choice is designed such that no matter what option the player chooses, they are shown an outcome that is either consistent or inconsistent with the expectation created in the differentiating information section. A consistent outcome is one that uses the mechanics from the failure mode that models World B. In this case, a consistent outcome would be for the snake oil to heal neither the sheriff nor the young man, no matter who the participant chooses. An inconsistent outcome is one that uses the mechanics from the original operator that models World A. In this case, an inconsistent outcome would be for the snake oil to heal either the sheriff or the young man. Inconsistent outcomes can only be produced by unconstrained domain revision, which can jump between alternate mechanics. Once domain revision has removed a possible action outcome modeled as a dynamic conditional effect, it will not be revisited and the outcome will remain consistent with the original intervention. The consistent and inconsistent conditions of each story diverge once the participant makes a choice. In the consistent condition, the participant is presented with an outcome for their choice that aligns with the World B expectation created earlier in the story:

Consistent 1 [You help the sheriff drink the elixir but he dies from the fatal wound.] The young man also succumbs to his bullet wound. You tell the town people that the elixir did not have time to fully restore their wounds. You sell several crates of elixir to protect the people from possible bullet wounds.

Consistent 2 [You help the young man drink the elixir but he dies from the fatal wound.] The sheriff also succumbs to his bullet wound. You tell the town people that the elixir did not have time to fully restore their wounds. You sell several crates of elixir to protect the people from possible bullet wounds.

No matter what the player decides, the sheriff and young man succumb to their fatal wounds. This outcome is consistent with the expectation created in the differentiating information portion of the story. These outcomes correspond to domain revision constrained on player knowledge. Conversely, the inconsistent outcomes explicitly break the expectation created earlier in the story:

Inconsistent 1 [You help the sheriff drink the elixir and it soon heals the fatal wound.] You run out to your wagon and bring back another bottle for the young man. He is fully healed as well. You sell several crates of elixir to protect the people from any additional bullet wounds.

Inconsistent 2 [You help the young man drink the elixir and it soon heals the fatal wound.] You run out to your wagon and bring back another bottle for the sheriff. He is fully healed as well. You sell several crates of elixir to protect the people from any additional bullet wounds.

No matter what the player decides, the sheriff and young man are healed by the elixir. This outcome is inconsistent with the expectations created in the differentiating information portion of the story. These outcomes correspond to domain revision unconstrained by player knowledge. The first sentence of each outcome, surrounded with brackets in the example, is the target sentence. Participant reading time on target sentences is what is measured and compared for the analysis. All of the target sentences are 18 syllables long and as similar as possible to control for text-level factors [34]. Target sentences are expected in consistent outcomes to be read at a faster average pace than target sentences in inconsistent outcomes. Finally, every story has a comprehension question that tests whether the participant was paying attention:

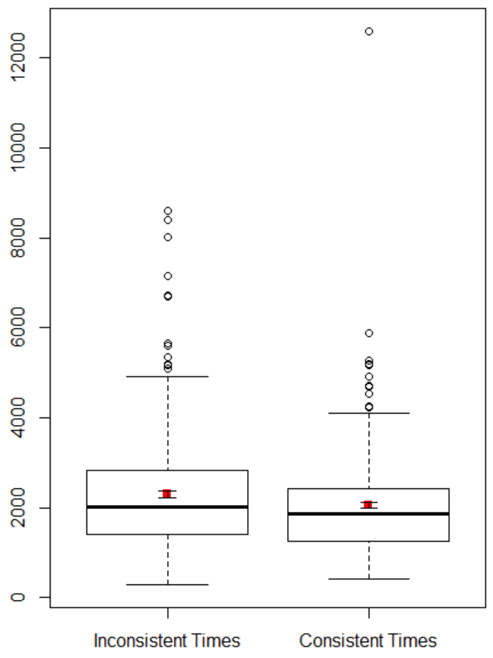
Question Did you visit a small town called Slate? Yes

The comprehension questions are all yes/no and concern some event or detail presented in the introduction or choice frame. These questions are used to gauge whether participants are paying attention to the stories they read.

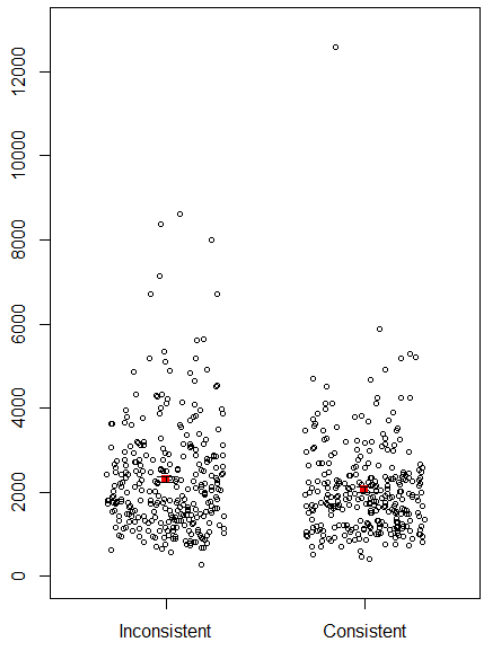
6.4.2 Procedure

An online testing apparatus was built and participants were recruited through Amazon's Mechanical Turk service to play through a series of ten short, one-choice Choose Your Own Adventure stories. Participants were told they would read a series of 10 short Wild West stories, make a choice that affects the outcome, and then answer a question about each experience. The study was advertised as taking 10 to 20 minutes and participants were paid a reward on completion. No mechanism prevented participants from skipping through the stories to get the reward, but each session was timed and participants were asked a simple comprehension question after each story. To begin, participants were presented with a short tutorial that instructed them to place their thumbs on the spacebar and index fingers on the 'f' and 'j' keys.

The tutorial and stories were presented to the participants one line at a time. Each press of the spacebar erased the current line and presented the next. Near the end of each story



(a) Box plot of data.



(b) Scatter plot with data jittered on the x-axis.

Figure 6.11: A box plot and scatter plot of the results. The y-axis is reading time in milliseconds. The red dots represent the mean of each group and the whiskers represent the standard error.

participants were presented with a two-option choice. They pressed the 'f' key to select the first option or the 'j' key to select the second option. Once the participant made their choice, they were presented with the story conclusion. Once finished, the participants answered a yes or no comprehension question by pressing the 'f' or 'j' keys. Participants were randomly assigned to one of two sequences of consistent and inconsistent test stories. The first three stories were used for training to ensure participants were comfortable with the system. These stories are not included in the analysis. The following seven stories were a series of test stories in the consistent or inconsistent condition. Each of the seven test stories conformed to the story structure laid out in the previous section. Reading times for each sentence were timed in milliseconds and stored on a database. Each comprehension response was also saved on the database.

6.4.3 Hypothesis

The hypothesis is that consistent target sentences will be read at a faster rate than inconsistent target sentences. To test this hypothesis a t-test was performed on the two groups.

6.4.4 Results

115 participants were collected on Mechanical Turk. The participants had to have a HIT approval rating 95% or above and have at least 500 HITs approved. The comprehension questions were used to screen for inattentive participants by only accepting those who answered at least 5 out of 10 questions correctly. Of the 115 original participants, 86 answered at least 50% of the comprehension questions correctly (acceptance rate: 75%). With seven stories per participant, 602 reading time data points are left. Of those, 11 were rejected for being outside three standard deviations from the mean (acceptance rate: 98.2%). The results of the t-test support the hypothesis that target sentences in the consistent group are read at a faster rate than sentences in the inconsistent group. There was a significant difference between the consistent ($M = 2058, SD = 1143$) and inconsistent ($M = 2302, SD = 1315$) groups, $t(578) = 2.40$, $p = 0.01667$ and it took participants on average 244ms longer to read the inconsistent target sentences than the consistent target sentences. Figure 6.11a shows a box plot of the data and Figure 6.11b shows a scatter plot of the data. These results are consistent with prior work, which found reading time differences from ∼200ms [39] to ∼300ms [1] for textually distant inconsistencies.

6.4.5 Discussion

The results of this more highly controlled study answer the second research question as applied to domain revision but it leaves open the question of whether event revision modifications may be noticed by story participants when they are unconstrained on a model of player knowledge. The next section modifies the study methodology presented here to test whether players notice unconstrained event revisions.

6.5 Human Subjects Evaluation 3

This section presents an evaluation of event revision in the context of short, one-choice Choose Your Own Adventures. The purpose of the experiment is to measure whether human participants notice the outcomes of inconsistent event revision actions. When an event revision takes place that contradicts the player's experience, it introduces a logical inconsistency into the story world that should be noticed by a participant. These are the events that are held static by a model of player knowledge. In this study there will be two types of short CYOA stories: those with *consistent* action outcomes and those with *inconsistent* action outcomes. The consistent action outcomes are modeled on worlds where domain revision maintains intervention mechanics throughout an entire story. The inconsistent action outcomes are modeled on worlds where intervention is free to move back and forth between two sets of contradicting outcomes for the same action. Participants should take longer on average to read inconsistent outcomes when compared with consistent outcomes. This delay should account for the time participants need to reconcile conflicting world models in their head while reading inconsistent stories.

6.5.1 Materials

A full list of the stories used in this experiment is given in Appendix B.2. Unlike the domain revision stories, which begin with an introduction and differentiating information, the event revision stories begin with a choice frame that is uniform across all participants:

Choice Frame You are a banker from New York visiting a small pioneer town named Slate. You are visiting Slate to inspect a new branch of your bank that has recently opened in the town. Your train arrives at the Slate station before noon and you spend some time checking into the town's hotel and storing your belongings. As you unpack, you try to decide whether you should get something to eat for lunch or go straight to the bank and introduce yourself. As you step outside the hotel you see the saloon in front of you and the bank next door.

Once the choice has been framed for the player, they are presented with a choice.

Choice Choice Go to the saloon. Go to the bank.

The two versions of the story diverge once the player makes their choice. The player is presented with the natural outcome of the action they selected in the consistent version:

Consistent 1 You walk over to the saloon and step through its swinging doors. [Inside the saloon you find a table, take a seat, and order a meal.] You relax and enjoy yourself for an hour until you feel recuperated from the long train ride. After your meal, you walk over to the bank, introduce yourself, and get to work.

Consistent 2 You walk over to the bank and open its large wooden door. [Inside the bank you are greeted by a clerk sitting alone at a desk.] The clerk introduces himself and shows you around the bank's offices and vault. After your tour, the clerk takes you to the saloon and buys you lunch.

In the inconsistent version, the player is presented with an initial action outcome that fits the choice they made. However, the next sentence is inconsistent with their choice, as if event revision modified their last action to be opposite the one they chose.

Inconsistent 1 You walk over to the saloon and step through its swinging doors. [Inside the bank you are greeted by a clerk sitting alone at a desk.] The clerk introduces himself and shows you around the bank's offices and vault. After your tour, the clerk takes you to the saloon and buys you lunch.

Inconsistent 2 You walk over to the bank and open its large wooden door. [Inside the saloon you find a table, take a seat, and order a meal.] You relax and enjoy yourself for an hour until you feel recuperated from the long train ride. After your meal, you walk over to the bank, introduce yourself, and get to work.

It is this second sentence, surrounded in brackets, that is the target sentence for this study. As in the domain revision study, participant reading time on target sentences is what is measured and compared in the analysis. All of the target sentences are 18 syllables long and as similar as possible to control for text-level factors [34]. Target sentences in consistent outcomes are expected to be read at a faster average pace than target sentences in inconsistent outcomes. Finally, every story has a comprehension question that tests whether the participant was paying attention.

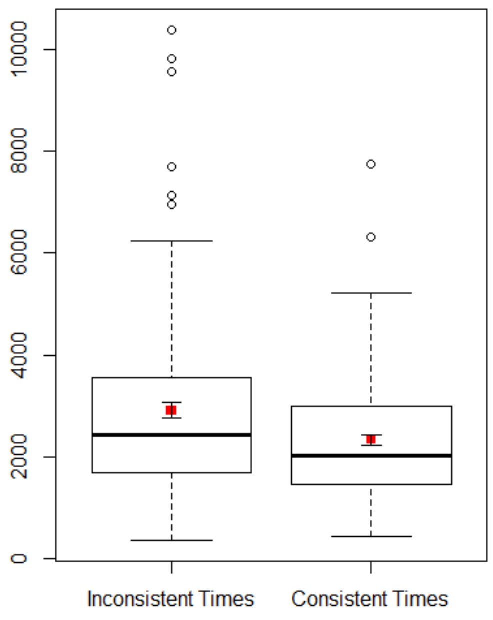
Question Were you a banker? Yes

The comprehension questions are all yes/no and concern some event or detail presented in the introduction or choice frame. These questions are used to gauge whether participants are paying attention to the stories they read.

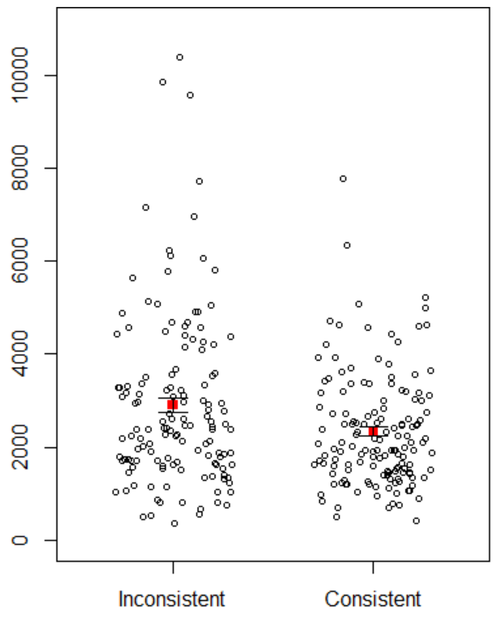
6.5.2 Procedure

An online testing apparatus was built and participants were recruited through Amazon's Mechanical Turk service to play through a series of ten short, one-choice Choose Your Own Adventure stories. Participants were told they would read a series of 10 short Wild West stories, make a choice that affects the outcome, and then answer a question about each experience. The study was advertised as taking 10 to 20 minutes and participants were paid a reward on completion. No mechanism prevented participants from skipping through the stories to get the reward, but each session was timed and participants were asked a simple comprehension question after each story. To begin, participants were presented with a short tutorial that instructed them to place their thumbs on the spacebar and index fingers on the 'f' and 'j' keys.

The tutorial and stories were presented to the participants one line at a time. Each press of the spacebar erased the current line and presented the next. Near the end of each story participants were presented with a two-option choice. They pressed the 'f' key to select the first option or the 'j' key to select the second option. Once the participant made their choice, they were presented with the story conclusion. Once finished, the participants answered a yes or no comprehension question by pressing the 'f' or 'j' keys. Participants were randomly assigned



(a) Box plot of data.



(b) Scatter plot with data jittered on the x-axis.

Figure 6.12: A box plot and scatter plot of the results. The y-axis is reading time in milliseconds. The red dots represent the mean of each group and the whiskers represent the standard error.

to one of two sequences of consistent and inconsistent test stories. The first three stories were used for training to ensure participants were comfortable with the system. These stories are not included in the analysis. The following seven stories were a series of test stories in the consistent or inconsistent condition. Each of the seven test stories conformed to the story structure laid out in the previous section. Reading times for each sentence were timed in milliseconds and stored on a database. Each comprehension response was also saved on the database.

6.5.3 Hypothesis

The hypothesis is that consistent target sentences will be read at a faster rate than inconsistent target sentences. To test this hypothesis a t-test was performed on the two groups.

6.5.4 Results

49 participants were collected on Amazon MTurk. The participants had to have a HIT approval rating 95% or above and have at least 500 HITs approved. The comprehension questions were used to screen for inattentive participants by only accepting those who answered at least 5 out of 10 questions correctly. Of the 49 original participants, 41 answered at least 50% of the comprehension questions correctly (acceptance rate: 84%). With seven stories per partici-

pant, that left 287 reading time data points. Of those, 4 were rejected for being outside three standard deviations from the mean (acceptance rate: 98.7%). The results of the t-test support the hypothesis that target sentences in the consistent group are read at a faster rate than sentences in the inconsistent group. There was a significant difference between the consistent ($M = 2328, SD = 1200$) and inconsistent ($M = 2908, SD = 1829$) groups, $t(235) = 3.12$, $p = 0.002$ and it took participants on average 579ms longer to read the inconsistent target sentences than the consistent target sentences. Figure 6.11a shows a box plot of the data and Figure 6.11b shows a scatter plot of the data. These results are more striking than the domain revision data. However, this result is expected because the inconsistencies introduced in the event revision study are one sentence away from contradicting information, whereas the ones in the domain revision study were farther away. Prior work [39] has shown this effect degrades as participants read.

6.5.5 Discussion and Future Work

These results provide evidence that interactive story participants notice when their action outcomes are modified from a model they were led to expect from the story world. This opens several interesting avenues for future work. First, these studies don't measure whether noticed inconsistencies lead to a decrease in player experience. Inconsistencies introduced by interventions are thought to be detrimental because system manipulations noticed by the player should lower their feelings of *agency*. Agency [35] is the satisfying power to take meaningful action and see the results of our decisions and choices. If players notice their actions are being manipulated by the storytelling system, it is expected to break the illusion that their actions are meaningful. Future work should explore whether noticed interventions cause a decrease in player agency.

A second avenue for future work is whether participants notice inconsistencies in learned rules that govern action outcomes as opposed to pre-existing real world rules. All of the World B models in the stories mapped onto external real world mechanics. For example, the participants probably already knew that snake oil elixirs are not magical cure-all potions from their knowledge of the real world. This leaves an open question of whether participants can identify inconsistencies in world mechanic rules they learn during the course of play rather than those that necessarily map onto their pre-existing world knowledge. For example, in an inversion of the snake oil example in a fantasy domain, would players of a *Legend of Zelda* [38] game notice if drinking a potion resulted in no status change as opposed to the normal behavior of restoring their health meter?

A third avenue for future work is to broaden the current approach outside of text-based games and measure whether players notice inconsistencies in visual 2D or 3D games. The main limiting factor of the current approach is it can only be applied to text domains and forces players to read the story line by line in order to accurately record reading comprehension times.

This is not possible for visual 2D or 3D games and is not feasible for longer text stories with many decisions. It would be helpful if another method existed to accurately measure whether participants notice inconsistencies in the game world. Finally, a set of experiments could be tied to what an automated system produces rather than this set of experiments which were handcrafted.

6.6 Summary

This section presents a series of four experiments that validate the two research questions laid out in Chapter 1. The first experiment used GME to expand and compare perceptual experience management trees to a baseline. It showed that perceptual experience management modifications create trees with more branches than those produced by accommodative mediation alone. The next three experiments use human subjects to validate the second research question. The first experiment cast doubt on whether participants distinguish between a unconstrained revision that introduces an inconsistency into an interactive story from event revision which does not introduce inconsistencies and a baseline that performs no interventions. Follow up studies control for several confounding factors present in the first study and provide evidence that participants take longer to comprehend stories with inconsistencies that could be produced by domain and event revision unconstrained by a model of player knowledge.

Chapter 7

Conclusion

This chapter contains two sections. Section 7.1 gives a summary of the dissertation, working through this document's problem, algorithm, implementation, and evaluation chapters. Section 7.2 lays out a body of future work enabled by this dissertation that ranges from a more robust model of player knowledge, to the superposition manipulation framework, and to UGME's PCG pipeline.

7.1 Summary

This document presents a framework for widening the branching factor of interactive narrative trees produced by automated, strong story experience management systems by using a model of player knowledge. This framework consists of two methods: event revision and domain revision. Event revision alters past events to find alternate possible world histories that are consistent with player observations. Domain revision alters action outcomes to find alternate possible world mechanics that are consistent with player observations. Together, these two methods of altering world histories and mechanics are called *perceptual experience management.* Strong story interactive narrative systems produce trees that branch for choices the player can make while interacting in a virtual environment. The perceptual experience manager presented in this document widens the branching factor of trees produced by strong story systems.

7.1.1 Problem Description

Chapter 3 describes the strong story perceptual experience management problem. It lays out a PDDL-based world representation, action-oriented world dynamics, introduces planning problems, and interactive planning problems. The chapter describes how these elements can be built up into interactive narrative play with a human participant. The chapter also describes an extensible model of player knowledge and how the model can be used to identify perceptual subsets of world information that are consistent with player knowledge. The chapter finally builds these elements into the interactive narrative task: to ensure during gameplay that $VH_{player} \neq \emptyset$. That is, it is the experience manager's job to ensure there is some path from the player's current position to a state where all goal conditions are enabled at all times during gameplay. The rest of the dissertation document sets out to show that perceptual experience management allows VH_{player} to contain at least one element in more situations than regular accommodation-driven mediation by allowing the set to contain stories with altered world histories and mechanics.

7.1.2 Perceptual Experience Management

Chapter 4 describes perceptual experience management, a method of modifying world mechanics and events in order to widen the number of interactive narrative tree branches consistent with author constraints. The chapter lays out event and domain differences between trajectories and describes how these differences can be found by perceptual experience management components to widen consistent interactive narrative branches. The chapter describes basic mediation that uses accommodation to build interactive narrative trees by classifying user actions as consistent, constituent, or exceptional. The chapter then describes event revision, its method for changing past events that are unobserved by the player and replacing them with alternate actions. It then describes domain revision, a method for shifting between alternate outcomes for actions as long as these shifts do not conflict with what the player has observed in the story world. The chapter then describes an integrated framework called superposition manipulation that performs event and domain revision equivalents every time the player makes an observation. The framework operates by maintaining a set of possible world states that are consistent with player observations and collapsing the set based on a utility estimation every time the player makes an observation that differentiates between stored superposition states. The chapter closes with an example set in the Spy Domain, a PDDL domain and problem pair that showcases event revision, domain revision, and superposition manipulation at work. The rest of the document describes a plan-based implementation of perceptual experience management and a series of evaluations.

7.1.3 Implementation

Chapter 5 describes a plan-based implementation of perceptual experience management, called the General Mediation Engine (GME). GME has many modules and this chapter steps through each one in order to provide a complete system description. The chapter begins with an overview of the system architecture and then describes the interfaces that model PDDL objects in GME's backend. The chapter describes how the objects are used to read and write PDDL domains and problems from and to disk, and how these PDDL files are used to query external planners for solution plans through a special GME interface. The chapter describes how these objects and the planner are used to build structures called mediation trees that represent interactive narratives. GME can build mediation trees with regular accommodation or it can use its model of player knowledge to build trees with event revision, domain revision, and superposition manipulation. GME also contains a series of interfaces that exposes underlying mediation tree states and transitions to a player. The chapter describes GME's text interface pipeline and three implementations: an offline console, online console, and online Choose-Your-Own-Adventure interface. Finally, the chapter ends by describing the visual 2D Unity General Mediation Engine (UGME) and the Base Case game. UGME dynamically configures and updates a visual world created with the Unity game engine according to underlying GME mediation trees.

7.1.4 Evaluation

Chapter 6 describes a series of structural and human subjects evaluations of perceptual experience management components. First, GME is used to evaluate event revision and domain revision against basic accommodation by measuring the branching factors of the trees each method produces. The chapter shows that GME's results match expectations in experiments using the Batman, Wild West, and Spy domains given in Appendices A.2, A.5, and A.4. The chapter then describes a series of three human subjects experiments that test whether human participants notice event revision and domain revision manipulations when they are not constrained to a model of player knowledge. The first experiment shows that domain revision outcomes are not noticed by human participants in a GME CYOA like environment, but neither are intervention outcomes. This result leads to a follow up study that leverages work in reading comprehension literature to test whether participants notice unconstrained domain revisions in short, single-choice interactive stories. The study matches up with previous results in a non-interactive setting and provides evidence that participants do notice unconstrained domain revisions. A final study tests event revision in a similar short, single-choice interactive story environment and finds evidence that participants notice unconstrained event revisions as well. This final section concludes the document and presents avenues for future work.

7.2 Future Work

This section presents areas of future work that can be explored to expand perceptual experience management, the General Mediation Engine, and its evaluations. This section explores three general areas of work that can be expanded on and improved in the future: the model of player knowledge, the superposition manipulation framework, and GME's PCG pipeline.

7.2.1 Player Knowledge

The framework for using a microtheory of user knowledge to generate a perceptual experience manager will remain the same, but the exact axioms that constitute the microtheory can be change to better reflect what players know in the story world. GME's current microtheory predicts that users perfectly observes and remembers everything they are co-located with. However, this is not always the case. First of all, players don't always observe everything they are co-located with. A more robust model could predict something is known to the player only if it has been presented on screen. Additionally, people don't perfectly observe and remember everything that is presented during gameplay. A further refinement would take care of a player's imperfect recall as well as their incomplete knowledge by probabilistically predicting which world state literals have appeared on screen but were not fully observed and remembered by the player.

7.2.2 Superposition Manipulation

The second component that can be improved with future work is superposition manipulation. There are three components that make up superposition manipulation: superposed state model and mechanics, player choice predictor, and a model of state utility. Of these, the player choice predictor and model of state utility can be improved to make the process more accurate and efficient. Finally, this document presents superposition manipulation in the context of interactive narrative domains. However, the process can be applied to any domain where there player autonomy must be balanced with system control and the experience manager can dynamically reconfigure the game environment.

Player Choice

The first superposition manipulation component that can be improved is adding a model of player choice. The system currently assumes a uniform distribution over actions a player could make. If an accurate model of player choice was incorporated, whether a personally tailored learned model or a general psychological one, or both, then state transitions could be weighted with probability values during utility evaluation when choosing between superpositions. With this model of player choice, the system could accurately predict what the player will do in the future which could make the process of collapsing superpositions more efficient and raise the expected probability that the interactive story will reach a goal state.

Choice Heuristic

The second superposition manipulation component that can be improved is adding a heuristic model of utility to superposition states. The system currently has no quick way to determine the ratio of *wins*, paths underneath a state where author constraints hold, to *losses*, paths underneath a state where author constraints are irrevocably broken, without fully expanding all successor paths. In order to determine utility the system must fully expand all paths, which is infeasible in many situations. To use the superposition manipulation framework effectively, a heuristic is needed that can quickly and accurately guess the win-loss ratio of states and sets of states when collapsing superpositions.

Non-Narrative Domains

This document presents superposition manipulation solely in the context of interactive narrative domains. However, the method can be applied to any domain where there is a tension between player choice and author constraints. For example, an educational or training domain where the tutor wants a particular pattern of events or world state to take place or an adversarial game player that wants to reach a winning world state in worlds with incomplete information like *Stratego* or *Star Craft*.

7.2.3 General PCG Pipeline

The final GME component that can be improved is a general pipeline for translating PDDL state and mechanics into a playable interface. The two general interfaces GME has right now is a text interface and a visual 2D interface. The text system uses predefined text templates along with state literal names to convey state and action update information. The visual 2D system uses prefabs, a predefined generator that assembles the level, and predefined action interfaces that convey declarative player actions to the experience manager. A full, robust PCG pipeline would also automatically determine these now predefined components.

First, a method to automatically generate general game mechanics for a visual space could be used in conjunction with GME to automatically map PDDL domain and problem pairs to game worlds. One way to do this would be to use GME in conjunction with a game description language from which game mechanics could be generated. In this case, the problem would be to find a mapping from the PDDL description to a corresponding game description that matches the PDDL. For example, if the PDDL describes a world with connected rooms that a player can navigate between, both *The Legend of Zelda* and *Metroid* would fit that description. A mapping algorithm would automatically match a given PDDL model to a game world description from which the interface can be generated.

7.3 Final Remarks

This document describes perceptual experience management, a method of furthering authorial control by creating a perceptual interactive narrative, and the General Mediation Engine, an implementation of a perceptual experience manager connected to a procedural generation pipeline that automatically configures interfaces for the client. Perceptual simulations are one tool that experience managers can use to create experiences that allow the player to retain autonomy and experience agency while also allowing the system to retain authorial control by accomplishing authorial goals.

REFERENCES

[1] Jason E. Albrecht and Edward J. O'Brien. Updating a Mental Model: Maintaining Both Local and Global Coherence. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 19(5):1061–1070, 1993.

[2] Ruth Aylett. Narrative in Virtual Environments - Towards Emergent Narrative. In *Working Notes of the Narrative Intelligence Symposium*, 1999.

[3] Ruth S. Aylett, Sandy Louchart, Joao Dias, Ana Paiva, and Marco Vala. FearNot! – an Experiment in Emergent Narrative. In *Intelligent Virtual Agents*, pages 305–316, 2005.

[4] Joseph Bates. The Nature of Characters in Interactive Worlds and the Oz Project. Technical Report CMU-CS-92-200, Carnegie Mellon University, 1992.

[5] Avrim L. Blum and Merrick L. Furst. Fast Planning Through Planning Graph Analysis. *Artificial intelligence*, 90(1):281–300, 1997.

[6] Amy Bruckman. The Combinatorics of Storytelling: Mystery Train Interactive. Master's thesis, MIT Media Laboratory, 1990.

[7] Rogelio E. Cardona-Rivera and R. Michael Young. Symbolic Plan Recognition in Interactive Narrative Environments. In *The Eight Intelligent Narrative Technologies Workshop at AIIDE*, 2015.

[8] Rogelio Enrique Cardona-Rivera, Justus Robertson, Stephen G. Ware, Brent E. Harrison, David L. Roberts, and Robert Michael Young. Foreseeing Meaningful Choices. In *Tenth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2014.

[9] Michael Cook, Squirrel Eiserloh, Justus Robertson, R. Michael Young, Tommy Thompson, David Churchill, Martin Cerny, Sergio Poo Hernandez, and Vadim Bulitko. Playable Experiences at AIIDE 2015. In *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2015.

[10] Ignacio X. Domınguez, Rogelio E. Cardona-Rivera, James K. Vance, and David L. Roberts. The Mimesis Effect: The Effect of Roles on Player Choice in Interactive Narrative Role-Playing Games. In *Proceedings of the 34th Annual CHI Conference on Human Factors in Computing Systems*, 2016.

[11] Richard Evans and Emily Short. VersuA Simulationist Storytelling System. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(2):113–130, 2014.

[12] Matthew William Fendt, Brent Harrison, Stephen G. Ware, Rogelio E. Cardona-Rivera, and David L. Roberts. Achieving the Illusion of Agency. In *Interactive Storytelling*, pages 114–125. Springer Berlin Heidelberg, 2012.

[13] Richard E. Fikes and Nils J. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2(3):189–208, 1972.

[14] Epic Games. Unreal Tournament, 1999.

[15] Emden R. Gansner and Stephen C. North. An Open Graph Visualization System and its Applications to Software Engineering. *SOFTWARE - PRACTICE AND EXPERIENCE*, 30(11):1203–1233, 2000.

[16] Ramanathan V. Guha and Douglas B. Lenat. Cyc: A Midterm Report. *Readings in Knowledge Acquisition and Learning*, pages 839–866, 1993.

[17] Justin Harris and R. Michael Young. Proactive Mediation in Plan-Based Narrative Environments. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(3):233–244, 2009.

[18] Ken Hartsook, Alexander Zook, Sauvik Das, and Mark O. Riedl. Toward Supporting Stories with Procedurally Generated Game Worlds. In *Computational Intelligence and Games*, pages 297–304. IEEE, 2011.

[19] Malte Helmert. The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[20] Jörg Hoffmann. FF: The Fast-Forward Planning System. *AI magazine*, 22(3):57, 2001.

[21] Konami. Metal Gear, 1987.

[22] Boyang Li, Stephen Lee-Urban, George Johnston, and Mark Riedl. Story Generation with Crowdsourced Plot Graphs. In *AAAI*, 2013.

[23] Boyang Li, Mohini Thakkar, Yijie Wang, and Mark O. Riedl. Data-Driven Alibi Story Telling for Social Believability. In *Social Believability in Games*, 2014.

[24] Sandy Louchart and Ruth Aylett. Solving the Narrative Paradox in VEs – Lessons from RPGs. In *Intelligent Virtual Agents*, pages 244–248, 2003.

[25] Bryan A. Loyall and Joseph Bates. Hap A Reactive, Adaptive Architecture for Agents. Technical Report CMU-CS-91-147, Carnegie Mellon University, 1991.

[26] Brian Magerko. Evaluating Preemptive Story Direction in the Interactive Drama Architecture. *Journal of Game Development*, 2(3):25–52, 2007.

[27] Henry B. Mann and Donald R. Whitney. On a Test of Whether One of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics*, pages 50–60, 1947.

[28] Michael Mateas and Andrew Stern. A Behavior Language for Story-Based Believable Agents. *IEEE Intelligent Systems*, 17(4):39–47, 2002.

[29] Michael Mateas and Andrew Stern. Façade: An Experiment in Building a Fully-Realized Interactive Drama. In *Game Developers Conference*, volume 2, 2003.

[30] Peter Mawhorter, Michael Mateas, Noah Wardrip-Fruin, and Arnav Jhala. Towards a Theory of Choice Poetics. In *Proceedings of the 9th International Conference on the Foundations of Digital Games*, 2014.

[31] Peter Andrew Mawhorter, Michael Mateas, and Noah Wardrip-Fruin. Generating Relaxed, Obvious, and Dilemma Choices with Dunyazad. In *Eleventh Conference on Artificial Intelligence for Interactive Digital Entertainment*, pages 58–64, 2015.

[32] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. *PDDL - The Planning Domain Definition Language*, 1998.

[33] Drew M. McDermott. The 1998 AI Planning Systems Competition. *AI Magazine*, 21(2):35, 2000.

[34] M. Windy McNerney, Kerri A. Goodwin, and Gabriel A. Radvansky. A Novel Study: A Situation Model Analysis of Reading Times. *Discourse Processes*, 48(7):453–474, 2011.

[35] Janet H. Murray. *Hamlet on the Holodeck: The Future of Narrative in Cyberspace*. Simon and Schuster, 1997.

[36] Mark J. Nelson and Michael Mateas. Towards Automated Game Design. In *Congress of the Italian Association for Artificial Intelligence*, pages 626–637. Springer, 2007.

[37] Nils J. Nilsson. Shakey the Robot. Technical report, SRI International, 1984.

[38] Nintendo. The Legend of Zelda, 1986.

[39] Edward J. O'Brien and Jason E. Albrecht. Comprehension Strategies in the Development of a Mental Model. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 18(4):777, 1992.

[40] Edward Packard. *The Cave of Time.* Choose Your Own Adventure. Bantam Books, 1979.

[41] Edwin Pednault. ADL: Exploring the Middle Ground Between STRIPS and the Situation Calculus. In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, pages 324–332, 1989.

[42] J. Scott Penberthy and Daniel S. Weld. UCPOP: A Sound, Complete, Partial Order Planner for ADL. *KR*, 92:103–114, 1992.

[43] Julie Porteous, Marc Cavazza, and Fred Charles. Applying Planning to Interactive Storytelling: Narrative Control Using State Constraints. *ACM Transactions on Intelligent Systems and Technology*, 1(2):10, 2010.

[44] Alejandro Ramirez and Vadim Bulitko. Automated Planning and Player Modelling for Interactive Storytelling. *IEEE Transactions on Computational Intelligence and AI in Games*, 2014.

[45] Rare. GoldenEye 007, 1997.

[46] Mark Riedl and Vadim Bulitko. Interactive Narrative: An Intelligent Systems Approach. *AI Magazine*, 34(1):67–77, 2013.

[47] Mark Riedl, C. J. Saretto, and R. Michael Young. Managing Interaction Between Users and Agents in a Multi-Agent Storytelling Environment. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems*, pages 741–748, 2003.

[48] Mark O. Riedl. Towards Integrating AI Story Controllers and Game Engines: Reconciling World State Representations. In *IJCAI Workshop on Reasoning, Representation and Learning in Computer Games*, 2005.

[49] Mark O. Riedl, Andrew Stern, Don M. Dini, and Jason M. Alderman. Dynamic Experience Management in Virtual Worlds for Entertainment, Education, and Training. *International Transactions on Systems Science and Applications*, 4(2):23–42, 2008.

[50] Mark O. Riedl and R. Michael Young. Open-World Planning for Story Generation. In *IJCAI*, pages 1719–1720, 2005.

[51] Mark O. Riedl and R. Michael Young. From Linear Story Generation to Branching Story Graphs. *Computer Graphics and Applications*, 26(3):23–31, 2006.

[52] Mark O. Riedl and R. Michael Young. Narrative Planning: Balancing Plot and Character. *Journal of Artificial Intelligence Research*, 39(1):217–268, 2010.

[53] Justus Robertson and R. Michael Young. Modelling Character Knowledge in Plan-Based Interactive Narrative to Extend Accomodative Mediation. In *Intelligent Narrative Technologies 6*, pages 93–96, 2013.

[54] Justus Robertson and R. Michael Young. Finding Schrödinger's Gun. In *Artificial Intelligence and Interactive Digital Entertainment*, 2014.

[55] Justus Robertson and R. Michael Young. Gameplay as On-Line Mediation Search. In *The First Experimental AI in Games Workshop at the Tenth Artificial Intelligence and Interactive Digital Entertainment Conference*, page 42, 2014.

[56] Justus Robertson and R. Michael Young. The General Mediation Engine. In *The First Experimental AI in Games Workshop at the Tenth Artificial Intelligence and Interactive Digital Entertainment Conference*, page 65, 2014.

[57] Justus Robertson and R. Michael Young. Automated Gameplay Generation from Declarative World Representations. In *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2015.

[58] Ben Sunshine-Hill and Norman I. Badler. Perceptually Realistic Behavior through Alibi Generation. In *Artificial Intelligence and Interactive Digital Entertainment*, pages 83–88, 2010.

[59] Ivo Swartjes, Edze Kruizinga, and Mariët Theune. Lets Pretend I Had a Sword. *Interactive Storytelling*, pages 264–267, 2008.

[60] David Thue, Vadim Bulitko, Marcia Spetch, and Eric Wasylishen. Interactive Storytelling: A Player Modelling Approach. In *In Proceedings of the 3rd Artificial Intelligence and Interactive Digital Entertainment Conference*, pages 43–48, 2007.

[61] Stephen G. Ware and R. Michael Young. Rethinking Traditional Planning Assumptions to Facilitate Narrative Generation. In *AAAI Fall Symposium: Computational Models of Narrative*, 2010.

[62] Stephen G. Ware and R. Michael Young. CPOCL: A Narrative Planner Supporting Conflict. In *Artificial Intelligence and Interactive Digital Entertainment*, pages 97–102, 2011.

[63] Stephen G. Ware and R. Michael Young. Glaive: A State-Space Narrative Planner Supporting Intentionality and Conflict. In *Artificial Intelligence and Interactive Digital Entertainment*, 2014.

[64] Frank Wilcoxon. Individual Comparisons by Ranking Methods. *Biometrics Bulletin*, 1(6):80–83, 1945.

[65] R. Michael Young and Johanna D. Moore. DPOCL: A Principled Approach to Discourse Planning. In *International Workshop on Natural Language Generation*, pages 13–20, 1994.

[66] R. Michael Young, Mark O. Riedl, Mark Branly, Arnav Jhala, R. J. Martin, and C. J. Saretto. An Architecture for Integrating Plan-Based Behavior Generation with Interactive Game Environments. *Journal of Game Development*, 1(1):51–70, 2004.

[67] R. Michael Young, Stephen G. Ware, Brad A. Cassell, and Justus Robertson. Plans and Planning in Narrative Generation: A Review of Plan-Based Approaches to the Generation of Story, Discourse and Interactivity in Narratives. *SDV. Sprache und Datenverarbeitung*, 2013.

[68] Hong Yu and Mark O. Riedl. Data-Driven Personalized Drama Management. In *Ninth Conference on Artificial Intelligence for Interactive Digital Entertainment*, pages 191–197, Boston, Massachusetts, 2013.

[69] Hong Yu and Mark O. Riedl. Optimizing Players Expected Enjoyment in Interactive Stories. In *Eleventh Conference on Artificial Intelligence for Interactive Digital Entertainment*, pages 100–106, 2015.

[70] Alexander Zook and Mark O. Riedl. Automatic Game Design via Mechanic Generation. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence*, 2014.

APPENDICES

Appendix A

Planning Domains and Problems

This appendix contains planning domains and problems referenced in the main document during examples and evaluations. The domains and problems are written in PDDL [33].

A.1 Base Case

This section contains a domain and problems used by the Base Case game to configure game worlds, control NPC behavior, and update world states according to player actions.

A.1.1 Base Case Domain

```
;;;
;;; The Base Case domain
;;; Created by Justus Robertson
;;; Describes actions that can be taken in the Base Case game.
;;;
(define (domain basecase)
   (:requirements :adl :disjunctive-preconditions
                  :negative-preconditions)
   (:predicates (at ?character ?location)
                (between ?door ?to ?from)
                (character ?character)
                (connected ?to ?from)
                (door ?door)
                (has ?character ?thing)
                (key ?key)
                (location ?location)
                (locked ?door)
                (nodoor ?to ?from)
```

```
             (opens ?key ?door)
             (thing ?thing)
             (tied ?character)
             (type ?character ?type))

;; A character moves from one place to another.
(:action move-location
    :parameters   (?mover ?location ?oldlocation)
    :precondition (and (character ?mover)
                       (at ?mover ?oldlocation)
                       (not (at ?mover ?location))
                       (not (tied ?mover))
                       (location ?location)
                       (connected ?location ?oldlocation)
                       (nodoor ?location ?oldlocation)
                       (location ?oldlocation))
    :effect       (and (not (at ?mover ?oldlocation))
                       (at ?mover ?location)))

;; A character moves from one place to another through an open door.
(:action move-location-door
    :parameters   (?mover ?door ?location ?oldlocation)
    :precondition (and (character ?mover)
                       (at ?mover ?oldlocation)
                       (not (at ?mover ?location))
                       (not (tied ?mover))
                       (door ?door)
                       (between ?door ?location ?oldlocation)
                       (not (locked ?door))
                       (location ?location)
                       (connected ?location ?oldlocation)
                       (location ?oldlocation))
    :effect       (and (not (at ?mover ?oldlocation))
                       (at ?mover ?location)))

;; A character can open a door with a key.
(:action open-door
    :parameters   (?opener ?key ?door)
    :precondition (and (character ?opener)
                       (has ?opener ?key)
                       (not (tied ?opener))
                       (door ?door)
```

```
                         (locked ?door)
                         (key ?key)
                         (opens ?key ?door)
                         (not (type ?opener soldier)))
    :effect       (not (locked ?door)))

;; A character can untie another character.
(:action untie-person
    :parameters   (?untier ?untied ?location)
    :precondition (and (character ?untier)
                         (at ?untier ?location)
                         (not (tied ?untier))
                         (not (captor ?untier ?untied))
                         (character ?untied)
                         (tied ?untied)
                         (at ?untied ?location)
                         (location ?location))
    :effect       (not (tied ?untied)))

;; A character can pick something up.
(:action take-thing
    :parameters   (?taker ?thing ?location)
    :precondition (and (character ?taker)
                         (at ?taker ?location)
                         (thing ?thing)
                         (at ?thing ?location)
                         (not (enchanted ?thing))
                         (location ?location))
    :effect       (and (not (at ?thing ?location))
                         (has ?taker ?thing))))
```

A.1.2 Base Case Problems

Problem 01

```
;;;
;;; The first of a series of Base Case problems the game cycles through.
;;;
(define (problem 01)
   (:domain basecase)
   (:objects player fox soldier hal
             cell_door exit_door dorm_door
             outside_door real_escape_door
             computer card
             cell hub dorm exit outside
             computerroom realescape)
   (:init (player player) (character player) (at player hub)
          (type player snake) (inventory)
          (character fox) (at fox cell)
          (type fox prisoner) (tied fox)
          (character soldier) (at soldier dorm)
          (type soldier soldier) (captor soldier fox)
          (computer computer) (at computer computerroom)
          (thing computer) (type computer computer)
          (key card) (thing card)
          (type card keycard) (at card hub)
          (opens card cell_door) (opens card outside_door)
          (door cell_door) (thing cell_door)
          (type cell_door door)
          (between cell_door hub cell)
          (between cell_door cell hub)
          (locked cell_door)
          (door exit_door) (thing exit_door)
          (type exit_door door)
          (between exit_door hub exit)
          (between exit_door exit hub)
          (door dorm_door) (thing dorm_door)
          (type dorm_door door)
          (between dorm_door dorm hub)
          (between dorm_door hub dorm)
          (door outside_door) (thing outside_door)
          (type outside_door door)
          (between outside_door exit outside)
          (between outside_door outside exit)
```

```
       (locked outside_door)
       (door real_escape_door) (thing real_escape_door)
       (type real_escape_door door)
       (between real_escape_door computerroom realescape)
       (between real_escape_door realescape computerroom)
       (locked real_escape_door)
       (location cell) (type cell base)
       (connected cell hub)
       (location hub) (type hub base)
       (connected hub cell) (connected hub exit)
       (connected hub dorm)
       (location exit) (type exit base)
       (connected exit hub) (connected exit outside)
       (location outside) (type outside woods)
       (connected outside exit)
       (location dorm) (type dorm base)
       (connected dorm hub) (connected dorm computerroom)
       (nodoor dorm computerroom)
       (location computerroom) (type computerroom specialbase)
       (connected computerroom dorm) (nodoor computerroom dorm)
       (connected computerroom realescape)
       (location realescape) (type realescape cave)
       (connected realescape computerroom))
(:goal (and (at fox outside) (at player outside) (at soldier hub))))
```

Problem 02

```
;;;
;;; The second of a series of Base Case problems the game cycles through.
;;;
(define (problem 02)
   (:domain basecase)
   (:objects player fox soldier blart paul
             main_screen cell_door armory_door
             exit_door entrance_door
             real_escape_door
             cell entrance hub jail armory
             dorm exit realescape
             cell_door_key)
   (:init (player player) (character player)
          (at player entrance) (type player snake) (inventory)
          (character fox) (at fox cell)
          (type fox prisoner) (tied fox)
          (character soldier) (at soldier hub)
          (type soldier soldier) (captor soldier fox)
          (character blart) (at blart dorm)
          (type blart soldier) (captor blart fox)
          (character paul) (at paul dorm)
          (type paul soldier) (captor paul fox)
          (computer main_screen) (at main_screen armory)
          (type main_screen computer)
          (key cell_door_key) (thing cell_door_key)
          (type cell_door_key keycard)
          (opens cell_door_key cell_door)
          (opens cell_door_key exit_door)
          (at cell_door_key dorm)
          (door cell_door) (thing cell_door)
          (type cell_door door)
          (between cell_door jail cell)
          (between cell_door cell jail)
          (locked cell_door)
          (door armory_door) (thing armory_door)
          (type armory_door door)
          (between armory_door jail armory)
          (between armory_door armory jail)
          (door exit_door) (thing exit_door)
          (type exit_door door)
```

```
       (between exit_door hub exit)
       (between exit_door exit hub)
       (locked exit_door)
       (door entrance_door) (thing entrance_door)
       (type entrance_door door)
       (between entrance_door entrance hub)
       (between entrance_door hub entrance)
       (location entrance) (type entrance specialbase)
       (connected entrance hub) (connected entrance realescape)
       (location cell) (type cell base)
       (connected cell jail)
       (location jail) (type jail base)
       (connected jail cell) (connected jail hub)
       (nodoor jail hub) (connected jail armory)
       (location hub) (type hub base)
       (connected hub jail) (nodoor hub jail)
       (connected hub entrance) (connected hub exit)
       (connected hub dorm) (nodoor hub dorm)
       (location exit) (type exit woods)
       (connected exit hub)
       (location dorm) (type dorm base)
       (connected dorm hub) (nodoor dorm hub)
       (location armory) (type armory base)
       (connected armory jail)
       (door real_escape_door) (thing real_escape_door)
       (type real_escape_door door)
       (between real_escape_door entrance realescape)
       (between real_escape_door realescape entrance)
       (locked real_escape_door)
       (location realescape) (type realescape cave)
       (connected realescape entrance))
(:goal (and (at soldier dorm) (at fox exit)
            (at player exit) (at blart entrance))))
```

Problem 03

```
;;;
;;; The third of a series of Base Case problems the game cycles through.
;;;
(define (problem 03)
   (:domain basecase)
   (:objects player fox soldier blart paul
             main_screen cell_door armory_door
             exit_door entrance_door
             real_escape_door
             cell entrance hub jail armory
             dorm exit realescape
             cell_door_key)
   (:init (player player) (character player) (at player entrance)
          (type player snake) (inventory)
          (character fox) (at fox cell)
          (type fox prisoner) (tied fox)
          (character soldier) (at soldier hub)
          (type soldier soldier) (captor soldier fox)
          (character blart) (at blart cell)
          (type blart soldier) (captor blart fox)
          (character paul) (at paul armory)
          (type paul soldier) (captor paul fox)
          (computer main_screen) (at main_screen armory)
          (type main_screen computer)
          (key cell_door_key) (thing cell_door_key)
          (type cell_door_key keycard)
          (opens cell_door_key cell_door)
          (opens cell_door_key exit_door)
          (at cell_door_key entrance)
          (door cell_door) (thing cell_door)
          (type cell_door door)
          (between cell_door jail cell)
          (between cell_door cell jail)
          (locked cell_door)
          (door armory_door) (thing armory_door)
          (type armory_door door)
          (between armory_door jail armory)
          (between armory_door armory jail)
          (door exit_door) (thing exit_door)
          (type exit_door door)
```

```
       (between exit_door hub exit)
       (between exit_door exit hub)
       (locked exit_door)
       (door entrance_door) (thing entrance_door)
       (type entrance_door door)
       (between entrance_door entrance hub)
       (between entrance_door hub entrance)
       (location entrance) (type entrance specialbase)
       (connected entrance hub)
       (location cell) (type cell specialbase)
       (connected cell jail) (connected cell realescape)
       (location jail) (type jail base)
       (connected jail cell) (connected jail hub)
       (nodoor jail hub) (connected jail armory)
       (location hub) (type hub base)
       (connected hub jail) (nodoor hub jail)
       (connected hub entrance) (connected hub exit)
       (connected hub dorm) (nodoor hub dorm)
       (location exit) (type exit woods)
       (connected exit hub)
       (location dorm) (type dorm base)
       (connected dorm hub) (nodoor dorm hub)
       (location armory) (type armory base)
       (connected armory jail)
       (door real_escape_door) (thing real_escape_door)
       (type real_escape_door door)
       (between real_escape_door cell realescape)
       (between real_escape_door realescape cell)
       (locked real_escape_door)
       (location realescape) (type realescape cave)
       (connected realescape cell))
(:goal (and (at soldier dorm) (at fox exit)
            (at player exit) (at blart entrance))))
```

Problem 04

```
;;;
;;; The fourth of a series of Base Case problems the game cycles through.
;;;
(define (problem 04)
   (:domain basecase)
   (:objects player fox soldier blart paul
             main_screen cell_door armory_door
             exit_door entrance_door
             real_escape_door
             cell entrance hub jail armory
             dorm exit realescape
             cell_door_key)
   (:init (player player) (character player) (at player entrance)
          (type player snake) (inventory)
          (character fox) (at fox cell)
          (type fox prisoner) (tied fox)
          (character soldier) (at soldier hub)
          (type soldier soldier) (captor soldier fox)
          (character blart) (at blart dorm)
          (type blart soldier) (captor blart fox)
          (character paul) (at paul dorm)
          (type paul soldier) (captor paul fox)
          (computer main_screen) (at main_screen armory)
          (type main_screen computer)
          (key cell_door_key) (thing cell_door_key)
          (type cell_door_key keycard)
          (opens cell_door_key cell_door)
          (opens cell_door_key exit_door)
          (at cell_door_key dorm)
          (door cell_door) (thing cell_door)
          (type cell_door door)
          (between cell_door jail cell)
          (between cell_door cell jail)
          (locked cell_door)
          (door armory_door) (thing armory_door)
          (type armory_door door)
          (between armory_door jail armory)
          (between armory_door armory jail)
          (door exit_door) (thing exit_door)
          (type exit_door door)
```

```
       (between exit_door hub exit)
       (between exit_door exit hub)
       (locked exit_door)
       (door entrance_door) (thing entrance_door)
       (type entrance_door door)
       (between entrance_door entrance hub)
       (between entrance_door hub entrance)
       (location entrance) (type entrance specialbase)
       (connected entrance hub)
       (connected entrance realescape)
       (location cell) (type cell base)
       (connected cell jail)
       (location jail) (type jail base)
       (connected jail cell) (connected jail hub)
       (nodoor jail hub) (connected jail armory)
       (location hub) (type hub base)
       (connected hub jail) (nodoor hub jail)
       (connected hub entrance) (connected hub exit)
       (connected hub dorm) (nodoor hub dorm)
       (location exit) (type exit woods)
       (connected exit hub)
       (location dorm) (type dorm base)
       (connected dorm hub) (nodoor dorm hub)
       (location armory) (type armory base)
       (connected armory jail)
       (door real_escape_door) (thing real_escape_door)
       (type real_escape_door door)
       (between real_escape_door entrance realescape)
       (between real_escape_door realescape entrance)
       (locked real_escape_door)
       (location realescape) (type realescape cave)
       (connected realescape entrance))
(:goal (and (at soldier dorm) (at fox exit)
            (at player exit) (at blart entrance))))
```

Problem 05

```
;;;
;;; The fifth of a series of Base Case problems the game cycles through.
;;;
(define (problem 05)
   (:domain basecase)
   (:objects player fox soldier blart paul
             main_screen cell_door armory_door
             exit_door entrance_door
             real_escape_door
             cell entrance hub jail armory
             dorm exit realescape
             cell_door_key)
   (:init (player player) (character player)
          (at player entrance) (type player snake) (inventory)
          (character fox) (at fox cell)
          (type fox prisoner) (tied fox)
          (character soldier) (at soldier hub)
          (type soldier soldier) (captor soldier fox)
          (character blart) (at blart dorm)
          (type blart soldier) (captor blart fox)
          (character paul) (at paul dorm)
          (type paul soldier) (captor paul fox)
          (computer main_screen) (at main_screen armory)
          (type main_screen computer)
          (key cell_door_key) (thing cell_door_key)
          (type cell_door_key keycard)
          (opens cell_door_key cell_door)
          (opens cell_door_key exit_door)
          (at cell_door_key dorm)
          (door cell_door) (thing cell_door)
          (type cell_door door)
          (between cell_door jail cell)
          (between cell_door cell jail)
          (locked cell_door)
          (door armory_door) (thing armory_door)
          (type armory_door door)
          (between armory_door jail armory)
          (between armory_door armory jail)
          (door exit_door) (thing exit_door)
          (type exit_door door)
```

```
        (between exit_door hub exit)
        (between exit_door exit hub)
        (locked exit_door)
        (door entrance_door) (thing entrance_door)
        (type entrance_door door)
        (between entrance_door entrance hub)
        (between entrance_door hub entrance)
        (location entrance) (type entrance specialbase)
        (connected entrance hub)
        (connected entrance realescape)
        (location cell) (type cell base)
        (connected cell jail)
        (location jail) (type jail base)
        (connected jail cell) (connected jail hub)
        (nodoor jail hub) (connected jail armory)
        (location hub) (type hub base)
        (connected hub jail) (nodoor hub jail)
        (connected hub entrance) (connected hub exit)
        (connected hub dorm) (nodoor hub dorm)
        (location exit) (type exit woods)
        (connected exit hub)
        (location dorm) (type dorm base)
        (connected dorm hub) (nodoor dorm hub)
        (location armory) (type armory base)
        (connected armory jail)
        (door real_escape_door) (thing real_escape_door)
        (type real_escape_door door)
        (between real_escape_door entrance realescape)
        (between real_escape_door realescape entrance)
        (locked real_escape_door)
        (location realescape) (type realescape cave)
        (connected realescape entrance))
(:goal (and (at soldier dorm) (at fox exit)
            (at player exit) (at blart entrance))))
```

A.2 Batman

This section contains a modified Batman domain meant for testing event revision. This modified domain contains henchmen that move Rachel and Harvey to different kidnapping locations instead of the Joker directly manipulating their locations from jail.

A.2.1 Batman Domain

```
;;;
;;; The Batman domain
;;; Created by Justus Robertson
;;; Deluxe version of the Batman domain with henchmen
;;;
(define (domain batman)
   (:requirements :adl :disjunctive-preconditions
                  :negative-preconditions :typing)
   (:types character location - thing)
   (:predicates (alive ?character - character)
                (apprehended ?character - character)
                (at ?thing - character ?location - location)
                (captured ?character - character)
                (connected ?to - location ?from - location)
                (has ?character - character ?character2 - character)
                (henchman ?henchman - character)
                (knows ?character - character)
                (player ?character - character)
                (saved ?saved - character ?saver - character))

   ;; A character moves from one place to another.
   (:action move-location
       :parameters   (?character - character ?to - location ?from - location)
       :precondition (and (player ?character)
                          (connected ?to ?from)
                          (at ?character ?from)
                          (knows ?character))
       :effect       (and (not (at ?character ?from))
                          (at ?character ?to)))

   ;; A character interrogates another character.
   (:action interrogate-character
    :parameters   (?interrogator - character ?interrogated - character
                   ?location - location)
    :precondition (and (player ?interrogator)
                     (at ?interrogator ?location)
                     (at ?interrogated ?location)
                     (apprehended ?interrogated))
  :effect       (knows ?interrogator))

;; A character places a kidnapped character at a location.
 (:action place-victim
     :parameters   (?placer - character ?victim - character
                    ?location - location ?placed - location)
     :precondition (and (henchman ?placer)
                        (has ?placer ?victim)
                        (at ?placer ?location)
                        (connected ?placed ?location))
     :effect       (and (not (has ?placer ?victim))
                        (at ?victim ?placed)
                        (captured ?victim)))

 ;; A character saves a captured character.
 (:action save-character
     :parameters   (?saver - character ?saved - character
                    ?notSaved - character ?location - location)
     :precondition (and (player ?saver)
                        (at ?saver ?location)
                        (at ?saved ?location)
                        (captured ?saved)
                        (captured ?notSaved)
                        (not (at ?notSaved ?location)))
     :effect       (and (saved ?saved ?saver)
                        (not (alive ?notSaved)))))
```

A.2.2 Batman Problems

Problem 01

```
;;;
;;; A Batman problem that allows event revision to take place
;;;
(define (problem 01)
   (:domain batman)
   (:objects batman joker harvey rachel henchman1 henchman2 - character
             52ndst avex gothampd car1 car2 - location)
   (:init (player batman) (at batman gothampd)
          (alive harvey)
          (alive rachel)
          (at joker gothampd) (apprehended joker)
          (at henchman1 car1) (henchman henchman1) (has henchman1 harvey)
          (at henchman2 car2) (henchman henchman2) (has henchman2 rachel)
          (connected 52ndst car1) (connected 52ndst car2)
          (connected 52ndst gothampd)
          (connected avex car1) (connected avex car2)
          (connected avex gothampd))
   (:goal (and (saved harvey batman)
               (not (alive rachel)))))
```

A.3 Key

This section contains a typed domain that models the Key example used in Chapters 3 and 4. Contains a problem for each of the three initial world configurations shown in Figure 3.6

A.3.1 Key Domain

```
;;;
;;; The Key domain
;;; Created by Justus Robertson
;;;
(define
   (domain KEY)
   (:requirements :adl :typing :universal-preconditions)
   (:types character key location door - thing)
   (:predicates (at ?thing - thing ?location - location)
                (connected ?location - location ?location - location ?door - door)
                (has ?character - character ?key - key)
                (locked ?door - door))

   ;; A character moves to a location through a door.
   (:action move-location
       :parameters   (?mover - character ?to - location
                      ?from - location ?door - door)
       :precondition (and (at ?mover ?from)
                          (connected ?from ?to ?door)
                          (not (locked ?door)))
       :effect       (and (not (at ?mover ?from))
                          (at ?mover ?to)))

   ;; A character takes a key.
   (:action take-key
      :parameters   (?taker - character ?key - key ?location - location)
      :precondition (and (at ?taker ?location)
                         (at ?key ?location))
      :effect       (and (has ?taker ?key)
                         (not (at ?key ?location))))

   ;; A character unlocks a door with a key.
   (:action unlock-door
      :parameters   (?unlocker - character ?door - door ?key - key
                     ?to - location ?from - location)
      :precondition (and (has ?unlocker ?key)
                         (at ?unlocker ?from)
                         (locked ?door)
                         (connected ?from ?to ?door))
      :effect       (not (locked ?door))))
```

A.3.2 Key Problems

Problem 01

```
;;;
;;; The first Key problem.
;;;
(define (problem 01)
   (:domain KEY)
   (:objects player - character
             key - key
             A B C D - location
             AB AC BD CD - door)
   (:init (at player A)
          (at key C)
          (connected A B AB)
          (connected A C AC)
          (connected B D BD)
          (connected C D CD)
          (locked CD)
          (locked BD))
   (:goal (at player D)))
```

Problem 02

```
;;;
;;; The second Key problem.
;;;
(define (problem 02)
   (:domain KEY)
   (:objects player - character
             key - key
             A B C D - location
             AB AC BD CD - door)
   (:init (at player A)
          (at key B)
          (connected A B AB)
          (connected A C AC)
          (connected B D BD)
          (connected C D CD)
          (locked CD)
          (locked BD))
   (:goal (at player D)))
```

Problem 03

```
;;;
;;; The third Key problem.
;;;
(define (problem 03)
   (:domain KEY)
   (:objects player - character
             key - key
             A B C D - location
             AB AC BD CD - door)
   (:init (at player A)
          (at key A)
          (connected A B AB)
          (connected A C AC)
          (connected B D BD)
          (connected C D CD)
          (locked CD)
          (locked BD))
   (:goal (at player D)))
```

A.4 Spy

This section contains a non-intentional domain loosely based on the final scene from the Nintendo 64 game, *GoldenEye 007*. This domain is meant for testing domain revision and event revision.

A.4.1 Spy Domain

```
;;;
;;; The SPY domain
;;; Created by Justus Robertson
;;;
(define (domain SPY)
   (:requirements :adl :typing :universal-preconditions)
   (:types character location - stat
           trap gun explosive detonator wire phone - item
           gears computer - object)
   (:predicates (alive ?character - character)
                (at ?character - character ?location - location)
                (at ?item - item ?location - location)
                (at ?object - object ?location - location)
                (connected ?location1 - location ?location2 - location)
                (destroyed ?item - item)
                (destroyed ?object - object)
                (disabled-trap ?character - character)
                (green ?explosive - explosive)
                (has ?character - character ?item - item)
                (linked ?phone - phone)
                (on ?explosive - explosive ?object - object)
                (player ?character - character)
                (red ?explosive - explosive)
                (used ?computer - computer ?user - character))

   ;; A character moves from one location to another.
   (:action move-location
       :parameters   (?mover - character ?to - location ?from - location)
       :precondition (and (at ?mover ?from)
                          (alive ?mover)
                          (connected ?from ?to))
       :effect       (and (not (at ?mover ?from))
                          (at ?mover ?to)))
```

```
;; One character with a gun shoots another character.
(:action shoot-character
    :parameters   (?shooter - character ?shot - character ?gun - gun
                   ?location - location)
    :precondition (and (has ?shooter ?gun)
                       (at ?shooter ?location)
                       (at ?shot ?location)
                       (alive ?shooter)
                       (alive ?shot)
                       (not (has ?shot ?gun)))
    :effect       (and (not (alive ?shot))))

;; A character with a gun and trip wire makes a trap.
(:action make-trap-wire
    :parameters   (?maker - character ?gun - gun ?wire - wire)
    :precondition (and (alive ?maker)
                       (has ?maker ?gun)
                       (has ?maker ?wire))
    :effect       (and (not (has ?maker ?gun))
                       (not (has ?maker ?wire))
                       (has ?maker trap)))

;; A character with a trap sets the trap.
(:action set-trap
    :parameters   (?setter - character ?trap - trap ?location - location)
    :precondition (and (alive ?setter)
                       (at ?setter ?location)
                       (has ?setter ?trap)
                       (not (at ?setter gear))
                       (not (at ?setter platform)))
    :effect       (and (not (has ?setter ?trap))
                       (at ?trap ?location)))

;; A character disables a trap.
(:action disable-trap
    :parameters   (?disabler - character ?trap - trap ?location - location)
    :precondition (and (alive ?disabler)
                       (at ?disabler ?location)
                       (at ?trap ?location))
    :effect       (and (not (at ?trap ?location))
                       (disabled-trap ?disabler)))
```

```
;; A character uses a computer.
(:action use-computer
    :parameters   (?user - character ?computer - computer ?location - location)
    :precondition (and (alive ?user)
                       (at ?user ?location)
                       (at ?computer ?location))
    :effect       (used ?computer ?user))

;; A character links their phone to an enabled computer.
(:action link-phone
    :parameters   (?user - character ?phone - phone ?computer - computer
                   ?location - location)
    :precondition (and (alive ?user)
                       (at ?user ?location)
                       (at ?computer ?location)
                       (has ?user ?phone)
                       (used ?computer ?user))
    :effect       (linked ?phone))

;; A character with an explosive places it on an object.
(:action place-explosive-thing
    :parameters   (?actor - character ?bomb - explosive ?thing - object
                   ?location - location)
    :precondition (and (alive ?actor)
                       (at ?actor ?location)
                       (has ?actor ?bomb)
                       (at ?thing ?location))
    :effect       (and (not (has ?actor ?bomb))
                       (on ?bomb ?thing)))

;; A character detonates a placed explosive.
(:action detonate-explosive
    :parameters   (?actor - character ?bomb - explosive ?detonator - detonator
                   ?thing - object)
    :precondition (and (alive ?actor)
                       (on ?bomb ?thing)
                       (detonates ?detonator ?bomb)
                       (has ?actor ?detonator))
    :effect       (and (when (red ?bomb)
                             (and (destroyed ?thing)
                                  (not (on ?bomb ?thing))))
                       (when (green ?bomb)
                              (and (destroyed ?thing)
                                   (not (on ?bomb ?thing))))))

;; A character toggles an explosive with a red light.
(:action toggle-green
    :parameters   (?toggler - character ?bomb - explosive)
    :precondition (and (alive ?toggler)
                       (has ?toggler ?bomb)
                       (red ?bomb))
    :effect       (and (not (red ?bomb))
                       (green ?bomb)))

;; A character toggles an explosive with a green light.
(:action toggle-red
    :parameters   (?toggler - character ?bomb - explosive)
    :precondition (and (alive ?toggler)
                       (has ?toggler ?bomb)
                       (green ?bomb))
    :effect       (and (not (green ?bomb))
                       (red ?bomb)))

;; A character picks up an explosive placed on an object.
(:action take-explosive
    :parameters   (?taker - character ?bomb - explosive ?detonator - detonator
                   ?thing - object ?location - location)
    :precondition (and (alive ?taker)
                       (on ?bomb ?thing)
                       (at ?taker ?location)
                       (at ?thing ?location)
                       (has ?taker ?detonator)
                       (detonates ?detonator ?bomb))
    :effect       (and (not (on ?bomb ?thing))
                       (has ?taker ?bomb))))
```

A.4.2 Spy Problems

Problem 01

```
;;;
;;; A SPY problem that presents opportunities for both event and domain revision
;;;
(define (problem 01)
   (:domain SPY)
   (:objects snake boss - character
             elevator gear left right platform - location
             gears - gears
             pp7 rifle - gun
             c4 - explosive
             detonator - detonator
             wire - wire
             phone - phone
             trap - trap
             terminal1 terminal2 terminal3 terminal4 - computer)
   (:init (player snake) (at snake elevator) (alive snake)
          (has snake pp7) (has snake c4) (has snake detonator)
          (red c4) (detonates detonator c4)
          (at boss gear) (alive boss)
          (has boss rifle) (has boss wire) (has boss phone)
          (at gears gear)
          (at terminal1 gear)
          (at terminal2 left)
          (at terminal3 right)
          (at terminal4 platform)
          (connected elevator gear)
          (connected gear left) (connected gear right)
          (connected left platform)
          (connected right platform))
   (:goal (and (alive snake)
               (not (alive boss))
               (at boss platform)
               (disabled-trap snake)
               (linked phone)
               (destroyed gears))))
```

A.5 Wild West

This section contains a non-intentional, typed domain set in a Wild West story. The story is used as an example for domain revision.

A.5.1 Wild West Domain

```
;;;
;;; The Wild West domain.
;;; Used as a domain revision example.
;;;
(define
   (domain WILDWEST)
   (:requirements :adl :typing :universal-preconditions)
   (:types character location - stat
           knife fire rope - item)
   (:predicates (at ?character - character ?location - location)
                (at ?item - item ?location - location)
                (burned ?rope - rope)
                (connected ?location1 - location ?location2 - location)
                (cut ?rope - rope)
                (escaped ?character - character)
                (has ?character - character ?item - item)
                (player ?character - character)
                (tied ?character - character)
                (tiedby ?character - character ?item - item)
                (rescued ?character - character ?rescued - character))

   ;; A character moves from one place to another.
   (:action move-location
       :parameters   (?mover - character ?to - location ?from - location)
       :precondition (and (at ?mover ?from)
                          (not (tied ?mover))
                          (connected ?from ?to))
       :effect       (and (not (at ?mover ?from))
                          (at ?mover ?to)))

   ;; One character unties a second character.
   (:action untie-character
      :parameters   (?untier - character ?untied - character
                     ?location - location ?rope - rope)
      :precondition (and (at ?untier ?location)
                    (at ?untied ?location)
                    (tiedby ?untied ?rope))
   :effect       (and (not (tied ?untied))
                    (not (tiedby ?untied ?rope))
                    (at ?rope ?location)
                    (rescued ?untier ?untied)))

;; One character ties another character with a rope.
(:action tie-character
   :parameters   (?tier - character ?tied - character
                  ?rope - rope ?location - location)
   :precondition (and (at ?tier ?location)
                    (has ?tier ?rope)
                    (at ?tied ?location))
   :effect       (and (tied ?tied)
                    (tiedby ?tied ?rope)
                    (not (has ?tier ?rope))))

;; A tied character cuts their rope with a knife.
;; Contains a dynamic conditional effect.
(:action cut-rope
   :parameters   (?cutter - character ?knife - knife
                  ?rope - rope ?location - location)
   :precondition (and (at ?cutter ?location)
                    (tiedby ?cutter ?rope))
   :effect       (and (when (has ?cutter ?knife)
                          (and (cut ?rope)
                               (at ?rope ?location)
                               (not (tied ?cutter))
                               (not (tiedby ?cutter ?rope))
                               (escaped ?cutter)))))

;; A tied character burns their rope with fire.
;; Contains a dynamic conditional effect.
(:action burn-rope
   :parameters   (?burner - character ?fire - fire
                  ?rope - rope ?location - location)
   :precondition (and (at ?burner ?location)
                    (tiedby ?burner ?rope))
   :effect       (and (when (at ?fire ?location)
                          (and (burnt ?rope)
                               (at ?rope ?location)
(not (tied ?burner))
(not (tiedby ?burner ?rope))
(escaped ?burner))))))
```

A.5.2 Wild West Problems

Problem 01

```
;;;
;;; The domain revision example problem set in the Wild West domain.
;;;
(define (problem 01)
   (:domain WILDWEST)
   (:objects player guard love - character
             house outside compound - location
             knife1 knife2 - knife
             bonfire - fire
             rope1 rope2 - rope)
   (:init (player player) (at player house) (tied player)
          (tiedby player rope1) (has player knife1)
          (at guard outside) (has guard knife2)
          (at love compound) (has love rope2)
          (at bonfire outside)
          (connected house outside)
          (connected outside house) (connected outside compound)
          (connected compound outside))
   (:goal (and (at guard compound)
               (rescued love player)
               (at rope2 outside)
               (escaped guard))))
```

A.6 Zombie

This section contains a non-intentional domain loosely based on the movie *The Evil Dead*. This domain is meant for testing GME gameplay mechanics for a text based adventure experience. This domain was created before GME supported PDDL typing.

A.6.1 Zombie Domain

```
;;;
;;; The ZOMBIE domain
;;; Created by Justus Robertson
;;;

(define
   (domain ZOMBIE)
   (:requirements :strips)
   (:predicates (alive ?character)
                (at ?character ?location)
                (axe ?axe)
                (book ?book)
                (character ?character)
                (closed ?thing)
                (evil ?thing)
                (free ?spirit)
                (gun ?gun)
                (has ?character ?thing)
                (hurt ?character)
                (key ?key)
                (location ?location)
                (locked ?thing)
                (object ?object)
                (open ?thing)
                (spirit ?spirit)
                (unlocks ?key ?thing)
                (zombie ?zombie))

   ;; A spirit posseses a person and turns them into a zombie.
   (:action posess-person
       :parameters   (?spirit ?person)
       :precondition (and (spirit ?spirit)
                          (alive ?person)
                          (free ?spirit))
```

```
    :effect       (and (not (alive ?person))
                           (zombie ?person)))

;; A character reads an evil book and releases the evil spirit.
(:action read-book
    :parameters   (?person ?book ?location)
    :precondition (and (character ?person)
                       (book ?book)
                       (evil ?book)
                       (has ?person ?book)
                       (at ?person ?location))
    :effect       (free evilspirit))

;; A character with a gun shoots a zombie.
(:action shoot-zombie
    :parameters   (?shooter ?zombie ?location ?gun)
    :precondition (and (character ?shooter)
                       (character ?zombie)
                       (location ?location)
                       (gun ?gun)
                       (alive ?shooter)
                       (zombie ?zombie)
                       (at ?shooter ?location)
                       (at ?zombie ?location)
                       (has ?shooter ?gun))
    :effect       (not (zombie ?zombie)))

;; A character with an axe chops a zombie.
(:action chop-zombie
    :parameters   (?chopper ?zombie ?location ?axe)
    :precondition (and (character ?chopper)
                       (character ?zombie)
                       (location ?location)
                       (axe ?axe)
                       (alive ?chopper)
                       (zombie ?zombie)
                       (at ?chopper ?location)
                       (at ?zombie ?location)
                       (has ?chopper ?axe))
    :effect       (not (zombie ?zombie)))

;; A character with an axe chops open a cabinet.
```

```
(:action chop-thing
    :parameters   (?chopper ?cabinet ?location ?axe)
    :precondition (and (character ?chopper)
                       (cabinet ?cabinet)
                       (location ?location)
                       (axe ?axe)
                       (alive ?chopper)
                       (at ?chopper ?location)
                       (at ?cabinet ?location)
                       (has ?chopper ?axe))
    :effect       (and (not (locked ?cabinet))
                       (not (closed ?cabinet))
                       (open ?cabinet)))

;; A zombie injures a character.
(:action scratch-person
    :parameters   (?zombie ?person ?location)
    :precondition (and (zombie ?zombie)
                       (location ?location)
                       (character ?person)
                       (alive ?person)
                       (at ?zombie ?location)
                       (at ?person ?location))
    :effect       (hurt ?person))

;; A character unlocks something with a key.
(:action unlock-thing
     :parameters   (?unlocker ?thing ?key ?room)
     :precondition (and (character ?unlocker)
                        (at ?unlocker ?room)
                        (at ?thing ?room)
                        (locked ?thing)
                        (alive ?unlocker)
                        (has ?unlocker ?key)
                        (key ?key)
                        (unlocks ?key ?thing))
    :effect        (not (locked ?thing)))

;; A character opens a closed, unlocked thing.
(:action open-thing
     :parameters   (?opener ?thing ?room)
     :precondition (and (character ?opener)
                        (at ?opener ?room)
                        (at ?thing ?room)
                        (not (open ?thing))
                        (alive ?opener)
                        (closed ?thing)
                        (not (locked ?thing)))
    :effect        (and (not (closed ?thing))
                        (open ?thing)))

;; A character moves from one location to another.
(:action move-location
    :parameters   (?mover ?location ?oldlocation)
    :precondition (and (character ?mover)
                       (location ?location)
                       (location ?oldlocation)
                       (at ?mover ?oldlocation)
                       (not (at ?mover ?location))
                       (alive ?mover)
                       (connected ?location ?oldlocation))
    :effect       (and (not (at ?mover ?oldlocation))
                       (at ?mover ?location)))

;; A zombie moves from one location to another.
(:action move-zombie
    :parameters   (?mover ?location ?oldlocation)
    :precondition (and (character ?mover)
                       (location ?location)
                       (location ?oldlocation)
                       (at ?mover ?oldlocation)
                       (not (at ?mover ?location))
                       (zombie ?mover)
                       (connected ?location ?oldlocation))
    :effect       (and (not (at ?mover ?oldlocation))
                       (at ?mover ?location)))

;; A character takes something from inside an open object.
(:action take-from
    :parameters   (?taker ?thing ?in ?place)
    :precondition (and (character ?taker)
                       (alive ?taker)
                       (open ?in)
                       (at ?taker ?place)
```

```
                      (in ?thing ?in)
                      (at ?in ?place))
    :effect       (and (not (in ?thing ?in))
                      (has ?taker ?thing)))

;; A character takes something from a location.
(:action take-thing
    :parameters   (?taker ?thing ?place)
    :precondition (and (character ?taker)
                      (alive ?taker)
                      (not (character ?thing))
                      (at ?taker ?place)
                      (at ?thing ?place)
                      (not (cabinet ?thing)))
    :effect       (and (not (at ?thing ?place))
                      (has ?taker ?thing))))
```

A.6.2 Zombie Problems

Problem 01

```
;;;
;;; A Zombie problem that serves as a text adventure.
;;;
(define (problem 01)
   (:domain ZOMBIE)
   (:objects ash
             linda
             evilspirit
             livingroom
             bedroom
             outside
             woodshed
             cellar
             cabinet
             key
             necronomicon
             boomstick
             axe)
   (:init (player ash) (character ash) (alive ash) (at ash livingroom)
          (character linda) (alive linda) (at linda livingroom)
          (character evilspirit) (spirit evilspirit)
          (location livingroom) (connected livingroom bedroom)
          (connected livingroom outside) (connected livingroom cellar)
          (location bedroom) (connected bedroom livingroom)
          (location outside) (connected outside livingroom)
          (connected outside woodshed)
          (location woodshed) (connected woodshed outside)
          (location cellar) (connected cellar livingroom)
          (cabinet cabinet) (closed cabinet) (locked cabinet) (at cabinet livingroom)
          (key key) (at key bedroom) (unlocks key cabinet) (old key)
          (book necronomicon) (evil necronomicon) (at necronomicon cellar)
          (gun boomstick) (in boomstick cabinet)
          (axe axe) (at axe woodshed))
   (:goal (and (not (zombie linda))
               (not (alive linda))
               (hurt ash))))
```

Appendix B

Evaluation Story Corpus

This appendix contains the interactive stories used in the evaluations of event and domain revision. The experiments using these stories are presented in Chapter 6, sections 6.4 and 6.5.

B.1 Domain Revision Stories

This section presents stories used in the domain revision evaluation. A description of the domain revision evaluation is given in Chapter 6, section 6.4. Seven stories were used in the experiment. Each story has six parts: an introduction, differentiating information, a choice frame, a choice, a choice outcome, and a comprehension question. The target sentences compared by the experiment are surrounded by brackets in the choice outcomes.

B.1.1 Lonely Miner

Introduction You have always wanted to be a miner. It was hard getting started but you found a mining job out west two years ago. It's exciting for you to enter the underground caverns each day and you couldn't be happier.

Differentiating Information You are extremely scared of heights and refuse to climb the ladder that leads out of the mine from where you work. You are also terrified of a pitfall near the entrance to the mine, spanned by a rope bridge. If you get close to either you nearly faint from fear. You show up thirty minutes early each day to avoid the ladder and take a long passage around the pitfall.

Choice Frame You always eat lunch inside the mine. You like to rest on the cool ground and eat your meal. Today you hear loud crashes as you eat and realize part of the mine is collapsing. You need to leave the mine as soon as possible. You grab your gear and head for the exit. You find that your usual route is blocked by boulders. You will have to find another way out of the mine.

Choice Climb the ladder. Cross the rope bridge.

Consistent 1 [You slowly approach the looming ladder but you are paralyzed by fear.] Eventually, a fellow miner arrives with a rope. She climbs to the top of the ladder and lifts you out using the rope. At the top, you lay down on solid ground and thank your co-worker for her help.

Consistent 2 [You slowly approach the swaying rope bridge but you are paralyzed by fear.] Eventually, a fellow miner arrives, picks you up, and carries you across the bridge and out into open air. Outside, you lay down on solid ground and thank your co-worker for their help.

Inconsistent 1 [You dash to the large ladder where you confidently and steadily climb.] You quickly reach the top and look around for other miners to help off the ladder. Eventually everyone gets out of the mine and thanks you for your help.

Inconsistent 2 [You dash to the long rope bridge where you confidently and steadily cross.] You quickly reach the other side and look around for other miners to help off the bridge. Eventually everyone gets out of the mine and thanks you for your help.

Question Did part of the mine collapse? Yes

B.1.2 Riverboat Ron

Introduction You love to spend time in your fishing boat on the river. You love to watch the water lap against the side of the small boat. You enjoy the sun and the cool breeze that sweeps across the water.

Differentiating Information You love the river but you are a poor swimmer. You avoid being in the river at all costs. You dread the feeling of being suspended in water with nothing below your feet. You become irrational and panic whenever you are in the river and cannot touch the riverbed.

Choice Frame Today you are fishing in your small boat on the river with your friend Ruth. The two of you talk about how long it has been since you have last fished. As you talk, you feel a strong tug at the end of your fishing line. You've caught a big fish! You are determined to stand your ground. You brace yourself against the edge of the boat and pull. Unfortunately, the fish is too strong for you. The fish drags you down into the water. You lose your grip on the fishing rod and it begins to move away from you.

Choice Swim back to the boat. Swim to your fishing rod.

Consistent 1 [You try to swim back to the boat but your arms and legs kick wildly in fear.] Ruth jumps into the river behind you and grabs you around the chest. Ruth asks you to be calm and you slowly relax. She helps you back onto the boat and you thank her for her help.

Consistent 2 [You try to swim to your fishing rod but your arms and legs kick fearfully.] Ruth jumps into the river behind you and grabs you around the chest. Ruth asks you to be calm and you slowly relax. She helps you back onto the boat and you thank her for her help.

Inconsistent 1 [You swim gracefully back to the boat, gliding along the river's surface.] You climb back on the boat and Ruth makes sure you're okay.

Inconsistent 2 [You swim gracefully to the fishing rod, gliding on the river's surface.] You grab the rod from the water but the fish has freed itself. You swim back to the boat and Ruth makes sure you're okay.

Question Were you on a boat in the ocean? No

B.1.3 House on the Prairie

Introduction You live with your ma and pa in a cabin on the prairie. Your pa built the cabin before you were born. It's the only home you've ever known.

Differentiating Information You have always been smaller and weaker than other children. When other kids go outside, run, and play you prefer to remain inside. You have trouble lifting things much heavier than a book. You can't run more than a couple feet without needing to sit down and rest.

Choice Frame Today your pa is in town buying supplies for the upcoming month. When he arrives with a wagon full of heavy crates the sky overhead is dark with clouds. Your pa is worried that a storm is about to begin. A storm will damage the goods in the crates. He asks you to help move the crates or bring a pile of heavy firewood into the house.

Choice Move the crates. Move the firewood.

Consistent 1 [You walk to the wagon and try to lift a crate but you are very weak.] Your mother comes outside and helps your father move the crates while you watch. Then they move the firewood. They get everything inside the house before the storm begins.

Consistent 2 [You walk to the firewood and try to lift a log but you are very weak.] Your mother comes outside and helps your father move the crates while you watch. Then they move the firewood. They get everything inside the house before the storm begins.

Inconsistent 1 [You rush to the wagon, lift the heavy crates, and run to the house.] Your father moves all the firewood inside. You get everything inside the house before the storm begins.

Inconsistent 2 [You rush to the firewood, lift the heavy logs, and carry them in the house.] Your father moves all the crates inside. You get everything inside the house before the storm begins.

Question Did your ma go into town for supplies? No

B.1.4 Train Conductor

Introduction You are an experienced conductor who runs a luxury passenger train from New York to Dallas. You enjoy your job and spend a lot of time with your passengers.

Differentiating Information You've always been fascinated by Europe and dream of visiting. You love trying to have conversations with people who speak other languages. Unfortunately, you only know how to speak English. No matter how hard you try you can never remember even a few words of other languages.

Choice Frame Today you are leaving the New York station and are excited to get underway. You always walk down to the dining car and get to know the new passengers after you leave the station. You notice that there are two European families on the train today, one from France and one from Germany. You can't wait to talk with them.

Choice Talk to the French family. Talk to the German family.

Consistent 1 [You approach the French family but have a hard time conversing with them.] You say goodbye after a while and resume your duties.

Consistent 2 [You approach the German family but have a hard time talking with them.] You say goodbye after a while and resume your duties.

Inconsistent 1 [You approach the French family and fluently converse with them in French.] You ask them all about their home country and their journey to the United States. You always love getting to know your passengers.

Inconsistent 2 [You walk up to the German family and speak fluent German with them.] You ask them all about their home country and their journey to the United States. You always love getting to know your passengers.

Question Was your train travelling to Dallas? Yes

B.1.5 Hotel Manager

Introduction You work as a manager at an upscale hotel on the edge of the frontier. You make sure your guests have everything they need and coordinate efforts among your hotel staff.

Differentiating Information You are not very good at fixing anything mechanical, especially if it requires pipes, wires, or gears. Whenever your guests have a mechanical problem, your strategy is to stall them until your handyman Owen can arrive.

Choice Frame Today has been a nightmare. The temperature has plumetted below freezing and a lot of travelers are passing through town. You have two complaints from Mister Smith, a very wealthy and grumpy man. One complaint is about a broken steam radiator and the second is about a busted plumbing pipe. It will be fifteen minutes before Owen arrives. You walk with Mister Smith to his room and reassure him that you will work on his problems until then.

Choice Work on the radiator. Work on the plumbing.

Consistent 1 [You look busy with the radiator but have no clue what to do.] Owen arrives before too long and takes over the rest of the job. You leave the room and return to your other duties.

Consistent 2 [You appear busy with the plumbing but have no idea what you're doing.] Owen arrives before too long and takes over the rest of the job. You leave the room and return to your other duties.

Inconsistent 1 [You get to work repairing the radiator and soon have it working.] Owen arrives before too long and takes over the rest of the job. You leave the room and return to your other duties.

Inconsistent 2 [You quickly get to work repairing the plumbing and soon have it working.] Owen arrives before too long and takes over the rest of the job. You leave the room and return to your other duties.

Question Was the temperature below freezing? Yes

B.1.6 Snake Oil Salesman

Introduction You sell snake oil liniment to the men and women of the pioneer. You travel near and far in your wagon selling what you pitch as a magical, cure-all elixir.

Differentiating Information Of course, your elixirs don't actually cure anything. You make the concoction from mineral oil, red pepper, and turpentine. No matter what you claim, your mixtures can't heal anything. In fact, they usually make people sick.

Choice Frame Today you arrive at a small town called Slate. You hear gunshots as you pull in to town. A young woman runs over to your wagon and implores you to bring elixir to the local tavern. As you enter the tavern with a bottle of elixir, you see the local sheriff and a young man laying on the floor. Each man has a fatal bullet wound in their stomach. The young woman asks you to save them with your elixir.

Choice Save the sheriff. Save the young man.

Consistent 1 [You help the sheriff drink the elixir but he dies from the fatal wound.] The young man also succumbs to his bullet wound. You tell the town people that the elixir did not have time to fully restore their wounds. You sell several crates of elixir to protect the people from possible bullet wounds.

Consistent 2 [You help the young man drink the elixir but he dies from the fatal wound.] The sheriff also succumbs to his bullet wound. You tell the town people that the elixir did not have time to fully restore their wounds. You sell several crates of elixir to protect the people from possible bullet wounds.

Inconsistent 1 [You help the sheriff drink the elixir and it soon heals the fatal wound.] You run out to your wagon and bring back another bottle for the young man. He is fully healed as well. You sell several crates of elixir to protect the people from any additional bullet wounds.

Inconsistent 2 [You help the young man drink the elixir and it soon heals the fatal wound.] You run out to your wagon and bring back another bottle for the sheriff. He is fully healed as well. You sell several crates of elixir to protect the people from any additional bullet wounds.

Question Did you visit a small town called Slate? Yes

B.1.7 Deputy Andy

Introduction You are a sheriff deputy in a small town. Your main duty is to tend the town jail each night after the sheriff goes home. You spend the long nights reading at a desk next to the cells.

Differentiating Information You are authorized to carry a rifle but you are a terrible shot. No matter how much you practice, you can't hit any targets. The sheriff is always amazed at how you consistently miss everything you aim at.

Choice Frame Tonight you are sitting alone in the town jail when you hear gunshots from the bank. You rush outside and see a wagon pull off into the night. The wagon has a heavy bag of cash on its top. You see its wheels spin as it retreats into the distance. You have to stop them!

Choice Shoot the wagon's wheel. Shoot the bag of cash.

Consistent 1 [You take aim, then fire your rifle and completely miss the wide wagon wheel.] You watch as the wagon disappears in the distance. After the shock has worn off, you rush over to the sheriff's house and tell him what happened. The sheriff thanks you and says he will handle everything.

Consistent 2 [You take aim, then fire your rifle and completely miss the large bag of cash.] You watch as the wagon disappears in the distance. After the shock has worn off, you rush over to the sheriff's house and tell him what happened. The sheriff thanks you and says he will handle everything.

Inconsistent 1 [You take aim, then fire your rifle and hit the wheel, disabling the wagon.] The wagon skids to a halt and two men jump off the top. The sheriff rushes up to you and assesses the situation. The two of you capture the bandits and lock them in jail. The sheriff thanks you for your work.

Inconsistent 2 [You fire your rifle and hit the bag of cash, which flies off from the wagon.] The wagon skids to a halt and two men jump off the top. The sheriff rushes up to you and assesses the situation. The two of you capture the bandits and lock them in jail. The sheriff thanks you for your work.

Question Did you hear gunshots from the general store? No

B.2 Event Revision Stories

This section presents stories used in the event revision evaluation. A description of the event revision evaluation is given in Chapter 6, section 6.5. Seven stories were used in the experiment. Each story has four parts: a choice frame, a choice, a choice outcome, and a comprehension question. The target sentences compared by the experiment are surrounded by brackets in the choice outcomes.

B.2.1 Traveling Banker

Choice Frame You are a banker from New York visiting a small pioneer town named Slate. You are visiting Slate to inspect a new branch of your bank that has recently opened in the town. Your train arrives at the Slate station before noon and you spend some time checking into the town's hotel and storing your belongings. As you unpack, you try to decide whether you should get something to eat for lunch or go straight to the bank and introduce yourself. As you step outside the hotel you see the saloon in front of you and the bank next door.

Choice Choice Go to the saloon. Go to the bank.

Consistent 1 You walk over to the saloon and step through its swinging doors. [Inside the saloon you find a table, take a seat, and order a meal.] You relax and enjoy yourself for an hour until you feel recuperated from the long train ride. After your meal, you walk over to the bank, introduce yourself, and get to work.

Consistent 2 You walk over to the bank and open its large wooden door. [Inside the bank you are greeted by a clerk sitting alone at a desk.] The clerk introduces himself and shows you around the bank's offices and vault. After your tour, the clerk takes you to the saloon and buys you lunch.

Inconsistent 1 You walk over to the saloon and step through its swinging doors. [Inside the bank you are greeted by a clerk sitting alone at a desk.] The clerk introduces himself and shows you around the bank's offices and vault. After your tour, the clerk takes you to the saloon and buys you lunch.

Inconsistent 2 You walk over to the bank and open its large wooden door. [Inside the saloon you find a table, take a seat, and order a meal.] You relax and enjoy yourself for an hour until you feel recuperated from the long train ride. After your meal, you walk over to the bank, introduce yourself, and get to work.

Question Were you a banker? Yes

B.2.2 Jail Duty

Choice Frame You are a sheriff deputy in a small town. Your main duty is to tend the jail each night after the sheriff goes home. You spend the long nights reading in an office next to the cells. Tonight, the town troublemaker is locked up after starting a fight inside the saloon. He is not happy and has been rattling his tin drinking cup against the cell bars for the last hour. You have had enough of the noise and can't concentrate on your book. You think about entering the cell and taking the cup away from your prisoner, but you could always just step outside the jail for a few minutes.

Choice Enter the cell. Go outside the jail.

Consistent 1 You open the cell door and step inside. [As you enter the cell, your prisoner ducks beneath you and runs outside.] You chase him out into town and tackle him before he can get away. You escort the troublemaker back to his cell, confiscate his tin can, and get back to your book.

Consistent 2 You open the jail's front door and step outside. [As you exit the jail, you see a man leave the bank with a bag of cash.] It is well past business hours, so you chase the man down and tackle him before he can get away. You escort the bandit back to the jail, lock him up, and confiscate the tin can as you leave the cell.

Inconsistent 1 You open the cell door and step inside. [As you exit the jail, you see a man leave the bank with a bag of cash.] It is well past business hours, so you chase the man down and tackle him before he can get away. You escort the bandit back to the jail, lock him up, and confiscate the tin can as you leave the cell.

Inconsistent 2 You open the jail's front door and step outside. [As you enter the cell, your prisoner ducks beneath you and runs outside.] You chase him out into town and tackle him before he can get away. You escort the troublemaker back to his cell, confiscate his tin can, and get back to your book.

Question Were you in a fight at the saloon? No

B.2.3 Farm Hand

Choice Frame You live with your ma and pa on a family farm. Your parents rely on you to take care of your family's two milk cows and help complete tasks around the farm. Today your pa is in town buying supplies for the upcoming month and the cows are grazing in a nearby field. You are outside fetching water when your pa arrives home with a wagon full of supplies from town. He asks you to either take the supplies inside or go out to the field and bring the cows in.

Choice Take the supplies inside. Go out to the field.

Consistent 1 You take a large bag of supplies and head inside. [As you enter the house you turn and unload the supplies in the pantry.] You continue transporting supplies and unloading them in the pantry until the wagon is empty. Your father soon returns with the cows from the pasture and loads them into the barn. Once finished, you head inside for the night.

Consistent 2 You head out to the nearby field where the cows are grazing. [As you walk the field over to the cows you begin calling out to them.] They slowly stop grazing and begin to head your way. You lead them back to the barn and make sure they are locked up. Once finished, you head inside for the night.

Inconsistent 1 You take a large bag of supplies and head inside. [As you walk the field over to the cows you begin calling out to them.] They slowly stop grazing and begin to head your way. You lead them back to the barn and make sure they are locked up. Once finished, you head inside for the night.

Inconsistent 2 You head out to the nearby field where the cows are grazing. [As you enter the house you turn and unload the supplies in the pantry.] You continue transporting supplies and unloading them in the pantry until the wagon is empty. Your father soon returns with the cows from the pasture and loads them into the barn. Once finished, you head inside for the night.

Question Does your family own two milk cows? Yes

B.2.4 Equestrian

Choice Frame Your family has owned horses ever since you were little. Each day you ride your favorite horse in one of many nearby pastures. As you leave today your mother tells you to be careful of a black bear seen near your house. As you are riding, you are distracted by a loud cry that sounds like an animal in pain. You look around to see where the cry has come from and see a cave entrance nearby. You wonder whether you should investigate the sound or head home.

Choice Enter the cave. Go back home.

Consistent 1 You approach the cave and step inside. [As you enter the cave you see a small deer fawn struggling with a large rock.] The rock has trapped the fawn's hoof so it cannot escape. You slowly approach the fawn and move the rock off its hoof. The fawn leaps out of the cave and you return home, happy to have helped the little creature out.

Consistent 2 You turn and gallop back towards your house. [As you approach the house you see your mother quickly run out to meet you.] Your mother tells you the bear has been spotted again not far from the house. She implores you to stable the horse and get inside the house as soon as possible. You quickly do as your mother tells you and make it inside without seeing the bear.

Inconsistent 1 You approach the cave and step inside. [As you approach the house you see your mother quickly run out to meet you.] Your mother tells you the bear has been spotted again not far from the house. She implores you to stable the horse and get inside the house as soon as possible. You quickly do as your mother tells you and make it inside without seeing the bear.

Inconsistent 2 You turn and gallop back towards your house. [As you enter the cave you see a small deer fawn struggling with a large rock.] The rock has trapped the fawn's hoof so it cannot escape. You slowly approach the fawn and move the rock off its hoof. The fawn leaps out of the cave and you return home, happy to have helped the little creature out.

Question Did you encounter a pack of wolves? No

B.2.5 Bandit

Choice Frame You are a bandit who robs the rich cattle barons of the west. You always work alone and make sure you are never discovered. Tonight you are taking a precious jewel from an especially rich man. You climb to and enter a window at the top of his large house and find a bedroom. You see a bed, a vanity, an open wardrobe, and a safe in the corner of the room. Suddenly you hear voices nearing the bedroom door that leads out into the hallway. You need to either hide yourself in the wardrobe or leave the room through the window.

Choice Enter the wardrobe. Exit through the window.

Consistent 1 You approach the wardrobe and step inside. [As you enter the wardrobe you silently shut the large door behind you.] After the voices leave the room, you exit the wardrobe and get to work. In no time at all you crack the safe's lock, take the precious jewel, and escape the house.

Consistent 2 You turn and climb out the window. [As you climb outside the window you silently latch it shut behind you.] You lower yourself to the ground and return to town. Once in town, you visit the saloon and replay the night's events in your head. The precious jewel in the safe will have to wait for another day.

Inconsistent 1 You approach the wardrobe and step inside. [As you climb outside the window you silently latch it shut behind you.] You lower yourself to the ground and return to town. Once in town, you visit the saloon and replay the night's events in your head. The precious jewel in the safe will have to wait for another day.

Inconsistent 2 You turn and climb out the window. [As you enter the wardrobe you silently shut the large door behind you.] After the voices leave the room, you exit the wardrobe and get to work. In no time at all you crack the safe's lock, take the precious jewel, and escape the house.

Question Were you trying to steal a jewel? Yes

B.2.6 Friendly Miner

Choice Frame You found a mining job out west two years ago. It's exciting for you to enter the underground caverns early each day and you couldn't be happier. Today as you enter the caverns you hear loud crashes from farther in and realize part of the mine has collapsed. Several of your friends arrived earlier than you and were working in the area of the cave-in, but most of your fellow miners are still on the surface. You want to go help your friends immediately, but wonder if you should head back and let everyone know what happened.

Choice Go deeper into the cave. Return to the surface.

Consistent 1 You hurry down into the mine to help your friends. [As you descend deeper down into the mine you find your friends are unharmed.] For the rest of the week you all work together to clear the rocks from the mine. Mining is hard work and there are many risks involved, but you remain optimistic through the week.

Consistent 2 You hurry to the surface to warn the other miners. [As you arrive up at the surface you find workers entering the cave.] You tell your coworkers about the cave-in and ask them to follow you down. The miners follow you to the cave-in area and together you make sure all of your friends are okay.

Inconsistent 1 You hurry down into the mine to help your friends. [As you arrive up at the surface you find workers entering the cave.] You tell your coworkers about the cave-in and ask them to follow you down. The miners follow you to the cave-in area and together you make sure all of your friends are okay.

Inconsistent 2 You hurry to the surface to warn the other miners. [As you descend deeper down into the mine you find your friends are unharmed.] For the rest of the week you all work together to clear the rocks from the mine. Mining is hard work and there are many risks involved, but you remain optimistic through the week.

Question Did you work as a blacksmith? No

B.2.7 Bank Robber

Choice Frame Two days ago you robbed a bank in the small town of Granite. Ever since, a posse led by the town's sheriff has been pursuing you through the wilderness outside of town. Your plan is to reach the town of Slate and catch a train back east before the posse catches you. The only thing between you and Slate is a large snowy mountain. You have to either go over the mountain or go around it through a forested valley which will be easier but take more time.

Choice Go over the mountain. Go through the valley.

Consistent 1 You ride up the slope of the mountain to save time. [As you approach the mountain's peak your horse has a hard time gaining footing.] You eventually make it over the peak and down the other side of the mountain. You reach the Slate station in time to catch the train back east with your new riches.

Consistent 2 You ride down through the valley to be safe. [As you ride through the valley's dense trees you find navigating difficult.] You eventually make it out of the woods and up the other side of the valley. You reach the Slate station in time to catch the train back east with your new riches.

Inconsistent 1 You ride up the slope of the mountain to save time. [As you ride through the valley's dense trees you find navigating difficult.] You eventually make it out of the woods and up the other side of the valley. You reach the Slate station in time to catch the train back east with your new riches.

Inconsistent 2 You ride down through the valley to be safe. [As you approach the mountain's peak your horse has a hard time gaining footing.] You eventually make it over the peak and down the other side of the mountain. You reach the Slate station in time to catch the train back east with your new riches.

Question Did you rob a bank? Yes