REPRESENTING AND REASONING ABOUT VIDEOGAME MECHANICS FOR AUTOMATED DESIGN SUPPORT

A Thesis
Presented to
The Academic Faculty

by

Mark J. Nelson

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Interactive Computing

Georgia Institute of Technology
August 2015

REPRESENTING AND REASONING ABOUT VIDEOGAME MECHANICS FOR AUTOMATED DESIGN SUPPORT

Approved by:

Dr. Charles Isbell, Advisor
School of Interactive Computing
Georgia Institute of Technology

Dr. Michael Mateas, Advisor
Department of Computational Media
University of California Santa Cruz

Dr. Ian Bogost
School of Interactive Computing
Georgia Institute of Technology

Dr. Noah Wardrip-Fruin
Department of Computational Media
University of California Santa Cruz

Dr. Ashok Goel
School of Interactive Computing
Georgia Institute of Technology

Date Approved: May 13, 2015

ACKNOWLEDGEMENTS

First, I'd like to acknowledge (and thank!) my advisors. Michael Mateas's influence is perhaps most pervasive in the work discussed here, since my work falls within the research tradition of *expressive intelligence* that he developed. And he has served as my primary research advisor, despite not being officially such due to having changed institutions (he was at Georgia Tech when I began, but is now at the University of California Santa Cruz). Thanks for providing a great intellectual environment in the Expressive Intelligence Studio (in both Atlanta and Santa Cruz), and years of hard-to-top mentorship and the kind of conversation one only gets from a widely read interdisciplinary expert. Charles Isbell served as my adoptive advisor, and deserves much thanks for helping shepherd me through the process of actually graduating, as well as contributing in a more normal advisory role, especially on the machine-learning side of things. In addition, thanks to him for fostering the intellectual environment of the *pfunk* research group I was fortunate to be a member of, which exceeded average funk levels of the Georgia Tech College of Computing by a large margin.

Secondly, I must thank many friends and colleagues, and I fear here I will inevitably omit a number who ought to be mentioned. At Georgia Tech, thanks to my officemates and discussion partners in the Expressive-AI and pfunk research groups and shared spaces: Sooraj Bhat, Andrew Cantino, Michael Holmes, Arya Irani, Chip Mappus, Nishant Mehta, Zach Pousman, David Roberts, Chris Simpkins, Christina Strong, Peng Zang, and Jichen Zhu. Thanks especially to Dave Roberts for a productive collaboration on a series of papers, and Arya Irani for much help navigating bureaucracy. Among other Georgia Tech friends, colleagues, staff, and professors, thanks go as well to: Mariam Asad, Calvin Ashmore, Ian Bogost, Randy Carpenter, Clara Fernández-Vara, Alexander Gray, Shan Shan Huang, Julie

Kientz, Dave Lillethun, Shwetak Patel, Chris Plaue, Ryan Riegel, Erika Shehan Poole, Cedric Stallworth, Chris Wojtan, Lana Yarosh, and José Zagal.

At my west-coast home of Santa Cruz, similar thanks goes out to my officemates and discussion partners in the Expressive Intelligence Studio's idyllic mountaintop headquarters: Sherol Chen, Mirjam Eladhari, Teale Fristoe, Ken Hullett, Martin Jennings-Teats, Arnav Jhala, Chris Lewis, Peter Mawhorter, Josh McCoy, David Olsen, Aaron Reed, Serdar Sali, Ben Samuel, James Skorupski, Adam Smith, Gillian Smith, Christina Strong, Anne Sullivan, Brandon Tearse, Mike Treanor, Noah Wardrip-Fruin, Ben Weber, and Jim Whitehead. Collaboration with Adam Smith has had a particularly large impact on this dissertation, and members of the #eis IRC channel were always a go-to source. Further Santa Cruzans deserving thanks include: John Dalessi, Kenn Knowles, Kathleen Kralowec, Josiah Pisciotta, and Alice Ye.

Finally, I'd like to warmly thank my parents for many years of support and encouragement. They deserve a longer thanks than this brief mention, but I'll deliver that in person.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Videogame designers hope to sculpt gameplay, but actually work in the concrete medium of computation. What they create is code, artwork, dialogue—everything that goes inside a videogame cartridge. In other materially constrained design domains, design-support tools help bridge this gap by automating portions of a design in some cases, and helping a designer understand the implications of their design decisions in others. I investigate AI-based videogame-design support, and do so from the perspective of putting knowledge-representation and reasoning (KRR) at the front. The KRR-centric approach starts by asking whether we can formalize an aspect of the game-design space in a way suitable for automated or semi-automated analysis, and if so, what can be done with the results. It begins with the question, "what could a computer possibly do here?", attempts to show that the computer actually can do so, and then looks at the implications of the computer doing so for design support.

To organize the space of game-design knowledge, I factor the broad notion of game mechanics mechanics into four categories: abstract mechanics, concrete audiovisual representations, thematic mappings, and input mappings. Concretely, I investigate KRR-centric formalizations in three domains, which probe into different portions of the four quadrants of game-design knowledge: 1. using story graphs and story-quality functions for writing interactive stories, 2. automatic game design focused on the "aboutness" of games, which auto-reskins videogames by formalizing generalized spaces of thematic references, and 3. enhancing mechanics-oriented videogame prototypes by encoding the game mechanics in temporal logic, so that they can be both played and queried.

CHAPTER I

INTRODUCTION

The topic of this thesis is the creation of knowledge representations and reasoning strategies that can support the game-design process. Key to making this work relevant and impactful, and to ensuring that the knowledge representations created actually capture significant and meaningful design spaces, is grounding the knowledge-representation (KR) formalisms and reasoning strategies in the game-design process *as actually understood by expert game designers*. I ground this work by building on a key insight from professional game-design practice, the Mechanics, Dynamics, Aesthetics (MDA) model [41]. This model views gameplay on three levels: the fundamental mechanics (the gameplay actions available to the players and how these actions evolve the game state), the dynamics (gameplay patterns) that emerge from the mechanics, and the aesthetic responses (interpretations) that players have to the dynamics. As a concrete example, consider the mechanics of card games, which can include shuffling, trick-taking and betting. These mechanics are fully specified by the rules of the game. From such card game mechanics, dynamics such as bluffing can arise, in which a player tricks another player into believing their hand is better than it actually is. In a game such as poker, bluffing is not an explicit rule; it arises as an emergent gameplay possibility as players explore the consequences of the rules. Finally, the aesthetics of a game arise from the dynamics. In a game such as poker, the aesthetics involve a psychological game of nerves in which one manipulates the beliefs and emotions of others while maintaining a cool head in one's own decision-making. In professional game-design circles, this has become a standard model, practiced by designers at frequent workshops held at the Game Developers Conference, and present in common game-design texts [1, 30, 105].

This perspective from expert game-design practice emphasizes the fundmental difficulty of game design. Game designers hope to sculpt gameplay: to design a game that, when played, will result in meaningful, aesthetic, interactive experiences. But these sculptors can't bang their chisels directly against gameplay. Instead, the designers' materials, through which they express their craft, are the mechanics: the systems of code, rules, and processes that the designer can actually put in the program for the game. How does one sculpt gameplay when working in the medium of mechanics? By definition the dynamics and aesthetics emerge in complex and unexpected ways from the mechanics. *Bridging this gap is a central challenge of game design*.

Can we, as AI researchers, help designers bridge the mechanics–gameplay gap? Other fields have long used *design-support* tools to help creators navigate the gap between their visions and their materials; the use of computer-assisted design (CAD) in architecture is a prominent example.[1] AI is commonly applied *in* videogames, for example for opponent AI, but is not yet commonly applied to helping designers craft the games themselves. The interest of this thesis is in enabling AI systems that help designers reason about the emergent gameplay properties of systems of mechanics.

1.1 Videogame knowledge representation

This thesis approaches videogame design support from the perspective of knowledge representation and reasoning (KRR) [20, 138]. Instead of starting with a specific tool or interface we'd like a designer to have, and then investigating how to build it, the KRR-centric approach starts by looking at the structure of game mechanics in particular slices of the game-design space, and investigating what a computational support system could plausibly do with them. Taking such a slice of the larger space of game design—level design of arcade-style games, say, or plot progression in story-driven mysteries—the KRR-centric

[1] Chapter 2 gives a brief history.

approach asks, what would a design-assistance tool need to "know" about the game mechanics, in order to be able to perform analyses of the mechanics–gameplay gap here? And then, how would we encode such knowledge? Finally, what can we build on top of the result?

There are two motivations for starting from knowledge-representation-and-reasoning when investigating videogame design support.

1.1.1 Knowledge representation as infrastructure

The first motivation is that representing knowledge declaratively provides a flexible infrastructure layer for videogame design support tools. Rather than each tool having to encode special-case information about game design each time it needs it, formalized declarative knowledge of (aspects of) a videogame domain provides a semantic resource that any tool can query, for very different applications.

Consider an analogy, in an easier to formalize domain, programming languages. Having a grammar for a language (a declarative representation) provides knowledge about the language used by many tools. Smart editors use the information to provide syntax-highlighting, code navigation, and visualization; analysis tools warn about likely errors; debuggers can trace a crash to its source location; and runtime systems can monitor execution.

Taking that analogy back to games, consider a declarative model of a puzzle game's mechanics and level design. If designers of front-end tools are given such a queryable model, this enables many tools: level editors that visualize reachable/unreachable areas; analysis tools that look for common design flaws; "debuggers" that help a designer find why something that "shouldn't have been possible" happened; and runtime systems that enforce constraints or adapt puzzles. In short, a huge range of design tools need to be able to *ask something* about a design, and formalizing aspects of a design gives them the backend infrastructure to do so, without strongly constraining how they should use it.

1.1.2 Starting from what the computer can do

The second motivation for starting from knowledge representation and reasoning is to ground our investigation solidly in what a computer can *actually do*, which in the videogame-design space is not yet at all clear. The strength of design-support systems is to amplify human designers' abilities by offloading analyses that machines are particularly good at, both giving the designer more information and freeing them to focus on other things (see Chapter 2 for a review of the literature). But this requires a realistic understanding of what the machine is in fact particularly good at.

An obvious reason to start from what the computer can actually do is to avoid starting with a pie-in-the-sky wishlist of analyses that would be great if we had them, but with no obvious way to implement them. To return to the programming languages analogy, the Common Lisp standard makes comments in various places that a "sufficiently smart compiler" could perform certain analyses, a turn of phrase that has become notorious for its theoretical optimism—and this in a domain that is already much more clearly formalized than videogame design.

A less obvious but equally important reason is that it's not actually clear what analyses *should* be on the wishlist. For one thing, existing videogame design processes are not that well understood. For another, designers understandably are often not entirely sure how they would use hypothetical design-support systems, given a wide-open field of possibilities. When I interviewed designers about what kinds of automatic analysis they might find useful (see Chapter 8), a common riposte was for them to ask instead: well, what kinds of automatic analysis can you do? Hence, this thesis investigates the latter question, in several domains.

1.2 Research Contributions

The primary research question driving this dissertation is:

What knowledge representation and reasoning (KRR) techniques can support a

designer in crossing the mechanics/gameplay divide, that is, in understanding the impact of mechanics design decisions on the dynamics and aesthetics of gameplay experiences?

Answering this question requires developing KRR techniques that effectively capture mechanics design spaces and from which useful conclusions regarding the gameplay implications of mechanics can be derived. But providing a single answer for "all games" is not possible: The diversity in the design space of games is enormous. The last decade has seen an enormous blossoming of genre innovation in the rise of the independent game scene, a proliferation of novel mobile, social media and embodied interaction platforms, and the mainstream acceptance of serious games as a vehicle for education, training, policy simulation, political communication, behavior change, and innumerable other applications. In the face of this diversity, the way forward in providing KRR-enabled design support is to focus on specific game design domains, grounding and validating specific KRR techniques in the concerns of of this domain.

This dissertation answers my research question in the context of three specific domains: 1. Multi-linear story progressions in storygames; 2. The relationship between the thematic assets of games and the mechanics; and 3. Representing and reasoning about discrete "board-game-like" mechanics to answer design questions that arise in early-stage rapid prototyping in the game design process.

Applying the approach outlined above to a particular game-design domain requires three steps: 1. Formalize an aspect of a design domain; 2. Develop ways of querying that formalization to answer design questions; and 3. Use the results in any of several settings, such as a design tool, an automated analysis tool, or a runtime system. Below I introduce each of my three domains and highlight my research contributions in each domain.

1.2.1 Declarative optimization-based drama management

Interactive storytelling systems aim to produce story-driven, narratively coherent experiences with a strong authorial voice, but nonetheless to allow the player freedom to interact with the storyworld, rather than putting them on linear "rails". This is typically done as multi-linear story design, in which the story of the game is revealed—perhaps through cutscenes, character dialog, or environmental elements such as notes, signs and architecture—as the player engages in gameplay action (such as movement, combat and resource management) to physically progress through the game world. Designers who wish to support multiple story paths must carefully orchestrate "lock and key" mechanisms to control the progression. This consists of bits of game state that are set by the player accomplishing specific actions, and which enable and disable potential paths through the storyworld (sometimes literaly by locking and unlocking doors, hence the common phrase "lock and key" for this logic). The difficulties of allowing players more than just a few branch points are well known. For example, the Computer Role Playing Game *Star Wars: Knights of the Old Republic*, which tries to provide many player-determined paths through the storyworld, has a number of quite severe story bugs, in which players experience inconsistent sequences of events, due to the designer not adequately reasoning about the space of possible interactions in the story progression logic [139, ch. 3]. This difficulty in the ability of the designer to reason about the emergent consequences of non-linear progression leads many designers in practice to enforce a linear sequence through the game. The popularity of the storygame design space, coupled with the known designer difficulties in reasoning about non-linear progression, make this a ripe target for KRR-based design support.

I start with a high-level formalization of the possible progressions through an interactive story, based on *plot points* and *ordering constraints*, proposed by Bates [4] and developed by Weyhrauch [141]. This formalism, *search-based drama management* (SBDM), provided designers with a means to declaratively specify the desired qualities of plot point progressions, using game-tree search to decide when to open-up, lock-down, or hint at

different possible paths through the storyworld. However, Weyhrauch only applied his formalism to a single small, relatively simple story world. Thus there was no validation of whether this KRR approach adequately captures the design space of multi-linear storyworld progression.

To validate this formalism, I apply it to the award-winning interactive story *Anchorhead*, which has the complexity of a real-world storygame. However, the orginal SBDM formalism doesn't scale to this case. First, the plot point and evaluation function model must be extended to adequately represent *Anchorhead* (for example, unlike Weyhrauch's simple test case, not all plot points take place in the same location). This dissertation presents the first declarative drama management formalism that handles the drama-management complexity of a real-world storygame. Second, game-tree search, Weyhrauch's method for querying the representation at runtime to produce optimal interventions into the story world, doesn't scale to a story of the complexity of *Anchorhead*. The original SBDM work depended on the evaluation function having symmetries to facilitate the manual construction of a memo table to speed up search. Such symmetries don't exist in general, and, even where they do, create additional authorial work for the designer who must find and exploit these symmetries in the search. To overcome this, I generalize Bates and Weyhrauch's search-based drama management to *declarative optimization-based drama management* (DODM), which may use any of several optimization methods (such as reinforcement learning).

Using these developments, I then demonstrate the first application of declarative drama management to a multi-linear story of real-world complexity, using reinforcement learning to solve the optimization problem. Furthermore, the generalization of drama management developed here (and the system implementing it) directly enabled a second parallel application, by colleagues who applied DODM in an entirely different genre, a graphical adventure game [124]. Finally, from a conceptual standpoint, I analyze how the notions of authorship and player freedom are accommodated by this formalism, and extend the evaluation

functions to better account for these goals.

To summarize, the contributions of my drama management work are:

1. The first declarative drama management formalism that scales to the complexity of a real-world storygame.

2. The first application of declarative drama management to a multi-linear story of real-world complexity, using reinforcement learning to solve the optimization problem.

3. An analysis of how the conflicting concepts of authorship and player agency can be adequately accounted for in this formalism.

1.2.2 Generating videogame themes

Automated game generation is the artificial intelligence problem of having a computer design a complete game. While there was a small amount of work in this area already, it focused almost entirely on generating *game rules*, mainly board-game rules. I argue in this thesis that automated videogame design requires tackling multiple areas besides game rules. One of the fuzzier ones, investigated in this second case study for the first time, is what I call *thematic mappings*, the real-world or semiotic references invoked by videogame elements, mechanics, and gameplay.

Although the inner logic of videogames is comprised of abstract computation, their surface-level presentation, which presents the first and often most salient level seen by the player, is full of concrete references to the real world (or an imagined one). While for purposes of beating a game the player could view themselves as simply an entity in a position in a space performing arbitrary actions, these are typically presented with a much richer set of semiotic references, which supply much of the feel and interpretation of the game. The player may be depicted as a sailor, wizard, soldier, or bartender; the space may be a desert, forest, bar, or racetrack; and the actions may be steering, serving beer, shooting, or engaging in commerce.

How arbitrary are these real-world references? And can an AI system understand them well enough to "reskin" a game to be about any arbitrary thing? This question has interest in itself from an AI perspective, but was also motivated by the goal of supporting the brainstorming of novice game designers who want to make small games about real-world subjects.

This second case study builds a formalism for reskinning classes of games based on constraints on their real-world references. A class of games is, for example, *a game in which something chases something else, which tries to get away*. A concrete game in this space may be, for example, *the player is a duck trying to avoid being shot by a hunter*. I build a prototype reskinning system for short, simple games in the style of Nintendo's *WarioWare* series, using the ConceptNet and WordNet databases of semantic relations to build editable constraint graphs specifing which combinations of real-world references make sense. Unlike with the drama-management case study, this system runs at design-time, not run-time; but like that case study, it is fully automatic, in the sense that the system selects a configuration according to the author-specified criteria.

The novel contributions of this part of the thesis therefore are:

1. The first analysis of the automated game design problem in generality, going beyond solely considering generation of *game rules*.

2. The first formalization of the *thematic mappings* of videogames, based on constraints over the semantic-relation knowledge bases ConceptNet and WordNet.

3. An interactive system for defining constraint-based *game spaces* that allow simple arcade-style games to be automatically reskinned to be "about" something else.

1.2.3 Mechanics-oriented prototype support

Game-design prototypes come in many forms, from those testing ideas about interfaces and art style, or how a focus group would react to a storyline idea, to those experimenting with

how rule systems interact. In a mechanics-oriented prototype, a designer is attempting to understand how their game operates, by implementing a simplified, stripped-down version of a set of game mechanics. These prototypes can take the form of physical tokens being moved around; or of an Excel spreadsheet in which the designer experiments with, *e.g.* an item system or combat system; or of lightweight computational experiments [30, 35].

Designers then playtest the prototypes (either themselves, or by recruiting players), and data (qualitative or quantitative) is collected about what happens during gameplay, which is used to revise the design. However, many things discovered during playtesting are not inherently empirical. If there was a solution to a puzzle and players simply couldn't find it, that would be a useful empirical playtesting result. But if there actually was no way out of a certain situation, due to a degenerate interaction of game mechanics, that is simply a fact about the game (an unwanted one), and shouldn't in principle require playtesting to discover.

This thesis's third case study therefore introduces the idea of *analytical metrics*, which are the automatically analyzed version of empirical, playtest-derived metrics. To compute them, it proposes encoding mechanics-oriented prototypes in formal logic, in order to be able to analytically query them for such properties. In addition, the same prototypes are made playable by playtesters, so empirical results can still be collected, and juxtaposed with the analytical results.

To formalize game mechanics, I propose using the *event calculus*, a temporal logic with a number of advantages for this domain (as I will argue). Briefly, it has a nice combination of usability as a modeling language and usability for computation. From a modeling perspective, it uses an ontology that aligns fairly well with the language of game mechanics: *events* happen at *times*, *state* holds at *times*, events can change state, and state conditions can trigger events. Unlike temporal logics originating in the verification community, it is also designed to be *elaboration tolerant*, meaning that statements can be added or removed incrementally without full-scale reformalization of a domain; this property is essential for

supporting the iterative nature of videogame prototyping. And important to its practical usability, off-the-shelf inference tools exist: in limited cases it can be translated directly to Prolog queries, and in a more general set of (discrete) cases, inference can be implemented via a SAT solver [68] or an answer-set solver [49].

I apply this formalism to build a "logical game engine", Ludocore, and a mechanics-oriented graphical prototyping tool that uses it, BIPED. These use Prolog to forward-simulate the models, enabling real-time play; and an answer-set programming solver to perform arbitrary design analysis on the formalized mechanics. Although the space of possible analyses one could pose in such a framework is quite large, I focus here on the idea of querying the formalized mechanics for sets of gameplay *traces* with certain properties, where a trace is a log of the game's progression, the analytically derived equivalent of what one gets when logging a playtester's play session. To demonstrate BIPED, I apply it to a prototype of a popular arcade game, *Motherload*.

The contributions of my mechanics-oriented prototyping work are, then:

1. Development of the event calculus as a general and elaboration-tolerant formalism for game mechanics.

2. Proposal of *analytical metrics* as a design-time alternative to playtesting, for those questions that can be answered by automated analysis of a game's mechanics.

3. A prototyping tool for early-stage, mechanics-oriented prototypes, BIPED, which allows the same prototype to be both played in real-time by human players, and automatically analyzed using design queries answered with logical inference.

1.3 Thesis organization

Chapter 2 provides a overview of the background that informs the AI representation-and-reasoning work carried out here, primarily through a survey of design-support and creativity-support systems. In addition, I develop a factoring of videogame knowledge

into four parts: abstract mechanics, concrete representations, thematic mappings, and input mappings. This gives a vocabulary for discussing the aspects of the problem that different formalizations may choose to focus on or sidestep—rather than attempting to formalize everything to do with videogame design as one large mess.

Chapters 3 and 4 investigate the first case study, declarative optimization-based drama management. Chapter 3 applies search-based drama management to a more complex story than it has previously been applied to, developing new representational idioms in order to do so. It also finds that search does not perform nearly as well in this more complex setting, compared to previously reported results. Therefore it decouples the *representational* aspect (plot points, drama manager actions, etc.) from the *optimization* aspect (search), recasting the formalism as declarative optimization-based drama management (DODM), and applying reinforcement learning as an alternative optimization method. Chapter 4 considers the implications for authoring and player agency of this formalism, defending it against concerns that have been raised in the literature.

Chapter 5 investigates the second case study, auto-skinning videogames. It proposes the notion of spaces of auto-reskinnable games, where specific games such as *a hunter chases a duck* are generalized into classes of games, such as *an attacking entity pursues an avoiding entity*. These classes are defined by a set of constraints specified with reference to the ConceptNet and WordNet databases of commonsense knowledge. Legal reskinnings matching the constraints are then layered onto a set of simple stock mechanics derived from Nintendo's *WarioWare* series of games.

Chapters 6, 7, and 8 investigate the third case study, mechanics-oriented prototype support. Chapter 6 argues for *analytical metrics*, based on design-time analysis of a game's mechanics, to join designers' arsenal alongside the more established empirical metrics derived from playtesting. It proposes an analysis of games as possibility spaces, viewed extensionally as bags of gameplay traces that could arise in them, and describes seven strategies for analytically characterizing these spaces. Chapter 7 proposes modeling games

specifically in the event calculus, a temporal logic based on events and time-varying state, and which accounts for the property of elaboration tolerance—being able to add or remove mechanics. Chapter 8 applies the preceding two chapters' formalism via a prototype. First, I use a "backend-only" prototype in interviews with designers to gauge whether mechanics-oriented prototype support would prove applicable and helpful to any of their existing design problems. Then, I demonstrate the feasibility of using the formalized mechanics in realtime for playtesting as well, collecting both analytical and empirical metrics using the same event-calculus-based prototype.

Finally, Chapter 9 concludes and contextualizes the thesis. I summarize the thesis's contributions and systems built, and discuss the broader impacts this work has had in the field, especially in enabling research by others on follow-up or divergent lines of investigation in the areas of automated game design and game-dynamics formalization. I then suggest three main areas for future work: improving the modularity and reusability of game formalizations, pursing a closer connection with game-studies research, and investigating the HCI side of game-design assistance tools.

CHAPTER II

DESIGN SUPPORT AND GAME-DESIGN KNOWLEDGE[1]

In order to develop knowledge-representation-based design-support systems for videogames, we need both to understand the landscape of design-support systems, and the kinds of knowledge we'll need to represent to build such systems for videogames. This chapter builds that requisite background by surveying the history of design-support systems, and proposing a factoring of game-design knowledge into four sub-areas.

2.1 Design-support systems

Ideas on how to integrate computers with the design process are nearly as old as practical electronic computers. Two articles from 1956—G.R. Price's piece "How to speed up invention" [92] for *Fortune Magazine* (at the time publishing serious long-form pieces) and D.T. Ross's conference paper "Gestalt programming: A new concept in automatic programming" [101]—conceived of an asymmetric conversational process between designers and machines, in which the machine carries out tedious calculations involving material properties, while the designer makes high-level design decisions. In the terminology of Donald Schön's influential view of design as a reflective conversation with the design situation [107], this is in retrospect a proposal for the machine to participate in the design conversation by increasing the "backtalk" of a situation, crunching numbers to illuminate current constraints and implications.

2.1.1 Initial development of CAD

The first serious requirements analysis for a design-support system (chronicled in J. Francis Reintjes's excellent history, *Numerical Control* [93]), determined that it should have a

[1] Parts of this chapter have been previously published as [74, 76, 78].

graphical input method that would allow designers to make and modify sketches; the system would both display refined sketches back to the designer, and simultaneously convert them into internal representations on which automated numerical analyses (such as stress analysis) could be performed. The result would be a system that should function in two roles: "at some times, it would be the designer's slave, at others it would alert the designer to impossible requirements or constraints being imposed" [93, p. 96]. This dual view of a backend automated reasoning system coupled with a front-end interactive modeling tool has gone through a series of evolutions, with parts variously emphasized or de-emphasized.

Early work on backend reasoning showed that designers were willing to try out more modifications when automated stress analysis was available, and also began exploring giving designers computerized parts catalogs, both so they could simulate the physical properties of a known part, and quickly retrieve parts with specific desired properties [93, p. 104–105].

Ivan Sutherland's now-famous Sketchpad system provided a front end, with a light pen and real-time graphics display that were at the time quite novel [125]. He showed some advantages to computer sketching over paper sketching, such as being able to precisely draw diagrams with large numbers of components (especially repetitive ones). He nonetheless concluded, "it is only worthwhile to make drawings on the computer if you get something more out of the drawing than just a drawing", so rather than positioning Sketchpad primarily as a computer drawing tool, he positioned it as a "man-machine graphical communication system", with sketching as the input method by which a designer communicated design information to the backend reasoning systems. To that end, it supported semantic annotations about the meanings of lines in the sketch and their relationships to each other, allowing, for example, force-distribution analysis on a sketch of a truss bridge, or simulation of sketches of electronic circuits (additional discussion on pp. 137–138 of his dissertation, which continues the conversational metaphor with the subtitle, *A Man-Machine Graphical Communication System*).

In contrast, some later systems did see interactive modeling and production of design diagrams as a major use of CAD (see, e.g., David Weisberg's recounting of the first commercial CAD system [140, chapter 6]). An influential line of work in that direction developed a set of graphically editable three-dimensional surface primitives that could be combined to produce arbitrary shapes [17].

2.1.2 Design research

A conceptual landmark on the "backend" side came in 1969 with Herb Simon's *Sciences of the Artificial* [111], which both developed a theory of design as a problem-solving activity, and connected it to concepts in the burgeoning artificial-intelligence literature.

By the 1970s, work had multiplied to such an extent, especially in the architecture community, that there existed a whole range of design tools, assistants, critiquers, automators, each varying in both technical properties as well as in ideas about how they should interact in and perhaps change the design process. One cluster of work was spurred by the 1971 introduction of shape grammars by George Stiny and James Gips [123]; they were taken up in architecture, tying CAD to the generative-content work that had already begun flourishing in areas like procedural art. By 1977, there was enough work for Nigel Cross to write a book-length survey taking stock of the field, *The Automated Architect* [18], consisting in part of a survey of the many extant systems, and in part of a deeper analysis of what automation was supposed to bring to architecture, both technically and socially. A blurb in 1977[2] billed it as "an anticipatory assessment of the impact of computer-aided design systems... written from a social and human orientation, in contrast to the machine orientation of almost all other work in the field", marking somewhat of a turn away from a pure engineering orientation.[3]

A subsequent wave of systems, coinciding with a stronger shift from the engineering to

[2]Advertising sidebar for the Pion Limited "Research in Planning and Design" book series, on p. 211 of the April 28, 1977 edition of *New Scientist*.

[3]Cross would, 30 years later, return to an analysis of what design research is exactly, concluding that it's a third field, neither pure science nor humanities, in his book *Designerly Ways of Knowing* [19].

the design community, took design-support systems in several different directions, mostly focusing on the importance of knowledge in specific design domains.

In many domains, vocabularies and representations had evolved over significant periods of time to encode useful ways of thinking about problems. Starting in the early 1980s, Bryan Lawson and his collaborators criticized the use of a generic set of geometric surfaces as the representation for all architectural design problems, arguing that doing so lost domain-specific knowledge, at worst even encouraging a design style that leads to visually impressive but poor designs, akin to using a lot of fonts and visual effects in desktop-publishing software (a retrospective appears in his 2002 *Leonardo* article, "CAD and creativity: does the computer really help?" [51]). Even when CAD didn't have outright negative effects, Lawson argues, the focus on visual modeling led to CAD failing to fulfill its original vision as a design assistant, and instead serving a narrower role as computerized draughtsman [52]. His own early attempt (1982) to improve that situation built a domain-specific tool for roof design, replacing the generic geometric primitives with a traditional architects' vocabulary of ridges, verges, valleys, eaves, hips, and so on—representations that bring relevant design questions to the fore, such as the relationship between structural support and space enclosure, and interior and exterior surfaces [95].[4]

The development of knowledge-based AI systems in the 1980s provided an opportunity to bring automated reasoning to more semantic representations (rather than numerical simulations like stress analysis, or purely structural inferences). For example, if a building were designed using terminology from municipal codes (windows, floors, hallways, etc.), and the municipal codes themselves were encoded in a design-support system, the system could determine which parts of the fire code applied, and whether a design met them [31].

[4]Lawson much later expanded on this theory of space in architecture in the 2001 book *Language of Space* [50], which provides an example of the kinds of domain-specific representational theories we might hope to one day have for videogames.

In the late 1980s and early 1990s, Gerhard Fischer introduced the idea of domain-oriented design environments (DODEs) [28], to combine and extend several of these approaches. DODEs start with building blocks meaningful in a particular domain (e.g. sinks, counters, ovens, and windows for kitchen design) and allow designers to compose them into higher-level representations. They extend the idea of domain-specific knowledge to include not only factual knowledge (such as structural soundness or building codes), but also *design* knowledge, such as best practices and common solutions. This knowledge can be employed to do things like critique a proposed design (the sink isn't in front of a window), or to provide design suggestions and the reasons for them (the sink should be placed near the range, due to common workflow), shifting the computer's role in the design conversation from providing backtalk to actively participating on the design side as well [27].

In addition to domain-specific environments, the 1980s and 1990s also saw, in echoes of Simon's approach, considerable renewed effort in tying design and artificial intelligence concepts more closely. One fairly direct way of doing so looked at, if not exactly equating, then at least pursing a close relationship between "design spaces" and "search spaces". This naturally points to the role of techniques such as heuristic search in design problem-solving. A 1991 paper on "searching for designs" by Robert Woodbury surveys both the conceptual and practical angles of the design-as-search view, as of that time [144]. Especially for problems where it is possible to specify a reasonably compact design space along with an easily evaluated design goal, this kind of search-oriented design has become a major field on its own, e.g. in the form of evolutionary design [5].

A different, contemporaneous line of research connected the design task to a more heterogeneous set of AI techniques, rather than viewing it as a "flat" problem of finding a good design in a design space, to be explored with black-box heuristic search. Many of these approaches can be grouped into a "propose–critique–modify" cycle [11], in which each aspect of the cycle provides different challenges to AI-based design. Among other

points of connection, the critique portion ties in closely with the idea of *verification* and *diagnosis* already explored in other areas of AI: a design critique can be thought of as verifying whether some design desiderata hold, or if not, then diagnosing the design problem [11, 38]. Chapter 6 especially takes up this connection to verification of design properties and diagnosis of design flaws. Treating the design problem, and its proposed solutions, as having richer structure than point solutions in a search space furthermore provides a basis for *redesign*, in which a targeted change to a design is desired without starting over from scratch [36, 122]. (In game design, this need is quite frequent.)

Knowledge-based systems do run into the problem that few domains are well-defined and static enough to effectively capture domain knowledge in a tool that can be built and deployed to users, leading to the necessity of *open systems* that can be evolved and extended [39]. Applying that principle to DODEs, they've been extended to support designers evolving (and sharing amongst each other) their representations and design knowledge [26], which has developed into a concept of *metadesign* systems that support the designer not only in evolving their design within a specific design domain, but also in the process of specifying and evolving the design domains themselves [32].

A different line of work at around the same time argued that CAD had fundamentally erred in being based around graphically interacting with a drawing—that of the two things that can be found in any design office, namely conversation and drawings, the conversation was where the design took place, with the drawings being secondary, and mainly representing the *end result* of design. In particular, Lawson and Loke [53] argued that, early in the design process, there is rarely a single design in progress for which there could be a drawing, but instead many, often disconnected, bits and pieces of design goals, tentative conclusions, design decisions, and ideas being pursued in parallel. Although admitting that drawing does play a role in this process, their work instead built a prototype system that converses textually with the designer, learning about his or her design goals, bringing them up later as reminders, making suggestions, critiquing ideas, answering questions, and so

on—a pretty solid return to the original conversation-with-machines metaphor.

2.1.3 Creativity support

As a result of this history of development, several commentators have abstracted general principles for how design-support systems might help specifically with the creative aspects of design.

Schön proposes [108] four main uses of a design system: enhance the seeing-drawing-seeing loop, allow construction and exploration of microworlds, help manage a repertoire of prototypes and apply them in specific design situations, and allow the designer to discover and reflect on their design knowledge.

Lawson and Loke propose [53] five roles for a system in the design conversation: learner, informer, critic, collaborator, and initiator. As a learner, the system makes note of the design goals and preferences of the designer, facts about the current design situation, proposed ideas or design decisions, and justifications for decisions or preferences. As an informer, it answers questions based on what it knows so far. As a critic, it checks the validity of comments the designer makes, and warns if there are problems with proposed design decisions. As a collaborator, it tries to elaborate on the designer's comments or proposals. As an initiator, it jumpstarts dead-ends by starting new lines of discussion or suggesting new perspectives on a problem.

Giaccardi and Fischer propose [32] that a creativity-support system needs to help designers cope with ill-defined problems by integrating problem *framing* with problem *solving*; to support reflective conversation with the design situation; and to support sharing of knowledge among people with different perspectives and backgrounds. To get at this, they propose (among other things) that design-support systems need to have embedded critics; need to support reuse and sharing of design representations and solution; and need to support collaboration among designers.

2.2 Factoring game-design knowledge

To develop a knowledge-representation-and-reasoning oriented design-support system for game mechanics, we need a more specific understanding of what kinds of design knowledge is involved in videogame design. An influential view of how game design is formally structured describes it as a mechanics–dynamics–aesthetics hierarchy: *mechanics* define the rules of the game, which interact with each other and the player to produce the *dynamics* of gameplay, which in turn interact with the game's art, cultural context, and the player's preferences to produce the *aesthetics* of the game [41].

The mechanics–dynamics–aesthetics pipeline describes how we get from the videogame itself—the code and content sitting on a computer or in a cartridge somewhere—to the dynamic results of the game actually being played, and ultimately to the aesthetic responses players have to that gameplay. But rather than treating the first term of the equation, "mechanics", as an amorphous sea, it's useful to factor it into several parts, since the representational questions are quite different. I propose four components: *abstract mechanics*, *concrete representation*, *thematic elements*, and *input mappings*.

I'll use the 1983 arcade game *Tapper* (Figure 1) as a running example, because it's fairly simple, yet clearly exhibits all four kinds of mechanics. In *Tapper*, the player is a bartender who fills up mugs of beer, and serves customers by sliding the beers down one of several bars. The customers move along the bars towards the bartender; serving a customer pushes him back towards the door. The goal is to push the customers out of the bar without letting any reach the bartender.

2.2.0.1 Abstract mechanics

A game's *abstract mechanics* specify abstract game state and how that state evolves over time, both autonomously and in response to player interaction. In a typical arcade game, abstract state includes things such as time limits, health levels, powerup status, and so on: anything that is conceptually part of game state, but doesn't make concrete commitments to

Figure 1: The arcade game *Tapper* (1983).

things like graphics engines, physics engines, on-screen display, etc. One way of thinking about how to strip a game down to its abstract mechanics is to think about what elements you'd put into a board-game or card-game version of the game.

In *Tapper*, the abstract mechanics are those of an order-fulfillment game: There are requesters (here, bar customers) who want certain items (beers) within certain time limits. There are sources for the player to get those items from (beer taps), and a way for the player to ferry the items between the sources and the requesters (sliding them down the bar). The abstract mechanics of an order-fulfillment game like *Tapper* are rules about requesters, sources, requested items, the progression of time, and the relationships between each. Changing the game at the abstract-mechanics level involves changing one of these aspects or their relationships; for example, we could make a *Tapper* variant with no time limits, or a variant where the sources run out of items after a certain number of retrievals, requiring the player to do something to restore the item source (tap runs out of beer, and player has to change the keg to get more beer).

2.2.0.2 Concrete representation

A game's *concrete representation* is how the game state actually is shown to the player, usually with audiovisual elements. A time limit, for example, might be represented with a literal clock on screen, or by the position of an object on screen, or even by the tempo of the music. The player's abstract position in a game world might be represented by telling them textually where they are (as in a text-adventure game); or by a sprite on a 2d, top-down-view map; or indirectly by showing the player their first-person view in a 3d game engine.

In *Tapper*, the abstract time limit within which an order request must be serviced is represented concretely by the customer's position along the bar: their distance from the player's end of the bar represents the time left to serve them. Serving a customer is represented by sliding a beer down the bar.

Concrete representations can have their own associated concrete mechanics: things like collision detection, movement trajectories, and so on. In *Tapper*, these are fairly minimal, but in a 3d first-person shooter the concrete mechanics specified by the physics engine might make up a fairly large part of the gameplay. Even in a game like *Zelda*, the concrete mechanics enforce things like which tiles the player can step onto, when something counts as having hit the player, and so on. The concrete and abstract mechanics interact: collisions in a concrete physics engine might cause the player's abstract health state to decrease; and abstract state like "can walk through walls" would in turn modify whether the physics engine enforces certain kinds of collision detection.

Finally, the *style* of the audiovisual elements themselves can be a signature part of the game design, which a game generator would need to either work within or innovate in. A useful point of reference on that subject is a study by Järvinen [44], which identifies three main audiovisual styles in vidoegames—photorealism, caricaturism, and abstractionism—and traces their development.

2.2.0.3 Thematic elements

Thematic elements are the real-world references a game makes. In *Tapper*, these are all clearly about being a bartender, filling up beer mugs from taps, and serving them to patrons sitting at bars. These elements have an effect because a player recognizes them as references to something they know from outside videogames. One way to think about what role thematic elements play is to imagine the same abstract mechanics, represented using the same concrete representation strategies, but with generic placeholder graphics: a version of *Tapper* where the player is a large oval, slinging triangles towards waiting squares that slowly move towards the player as their time limits count down.

Replacing a game's thematic elements without making significant changes to either its abstract mechanics or their concrete representations is called *reskinning*, and is a common way of modifying games, especially for user-made modifications. Appropriate use of thematic references is also a common element of *newsgames*, games that reference and comment on current events [134, 9].

2.2.0.4 Input mappings

Finally, for a game to be a game, it needs player input. *Input mappings* (or control mappings) describe the relationships between the physical player input, such as button presses and joystick movement, and modification of game state.

In *Tapper*, pressing a button at the tap begins filling a mug; releasing the button stops filling it and, if the mug is full, automatically slides it down the bar. The game could be significantly changed just by making small changes to these mappings, without changing any of the other mechanics of the game. For example, if sliding the beer down the bar was initiated with a separate button press, instead of automatically happening when the player released the button after filling, the player could fill up a beer at one tap, then notice that another patron was getting too close, and walk over to serve them instead of the customer they were initially planning to serve. Nothing in the mechanics makes this impossible, but

the input mappings make it so the player can't actually make that happen.

In simulation games, to choose another example, input mappings often fundamentally define what kind of gameplay is supported. A simulation with abstract mechanics, concrete representations, and thematic elements, but *no* input mappings (as is the case with some scientific and engineering simulations), could not possibly be a game. How exactly the player gets to intervene in the simulation strongly influences what kind of game it is. Can they start clicking on elements of the simulation world arbitrarily and directly change them, *SimCity* style? Or do they click on and control characters within the simulation, which in turn change other elements, like *Dungeon Keeper*? Or does the player only control one character in the world and have to influence the simulation that way, as is common in sports games?

2.2.0.5 Relative emphasis of factors

Of course, it is not a requirement that all games equally emphasize all four of these factors. In chess, the thematic content and concrete representation are of minimal importance; designing interesting chess variants would focus on the abstract game mechanics. A first-person shooter, on the other hand, puts large emphasis on the game's concrete representation and concrete mechanics (3d graphics and physics). A newsgame commenting on a recent event, or a political game designed to persuade you of a particular position, puts a large focus on the thematic elements. And, some arcade games put strong emphasis on the precise timing and cadence of the input mappings.

In the case studies that follow I will focus on formalizing specific subsets of this factoring, since separating the kinds of knowledge needed for game design was the motivation for developing it.

However the first case study, drama management, is the least clearly separated. Interactive storytelling deliberately aims at close coupling between a game's abstract mechanics (traversal through a game world's "story space") with the thematic mappings, since the

specific content of the story is what makes navigating "story space" story-related at all. Therefore a particular *story graph*, the formalism I investigate, will impact both aspects.

The second case study, auto-skinning games, is clearly focused on thematic mappings. In this case stuy I deliberately choose a domain with very simple mechanics—simple enough to be hard-coded with templated code—so the majority of the interesting aspects to formalize come from the real-world references layered on top of those mechanics.

Finally, the third case study, mechanics-oriented prototypes, is solidly focused on modeling abstract mechanics. Some concrete mechanics do creep in where difficult to disentangle—for example, graphical-logic games based on movement inevitably have references to concrete mechanics such as collision in their models.

CHAPTER III

DECLARATIVE OPTIMIZATION-BASED DRAMA MANAGEMENT[1]

Interactive drama is a type of interactive experience in which a player interacts with a rich story world and experiences both a strong feeling of interactive agency and a dramatic, interesting, and coherent narrative. Giving a player a large degree of freedom in a story world and populating it with believable agents will not necessarily emergently create interactive drama, since interactions must not only be believable but also combine to form a globally coherent and interesting narrative.

A drama manager (DM) coordinates and adapts the agents and other contents of a story world as an experience unfolds, in order to maintain global narrative coherence without removing the player's interactive agency. In a formalism proposed by Bates [4] and developed by Weyhrauch [141], termed search-based drama management (SBDM), the author specifies the narratively important events, called *plot points*, that can occur in an experience. Examples of plot points include a player gaining story information, changing their relationship with a non-player character, acquiring an important object, and so on. The plot points are annotated with ordering constraints that capture the physical possibilities of the story world. For example, events in a locked room are not possible until the player gets the key. Plot points are also annotated with bits of information that may be useful to the DM, such as where the plot point happens, what subplot it's part of, and so on (the exact set of annotations is flexible and depends on the story). Figure 3 shows the set of plot points and ordering constraints I developed to model a portion of Michael S. Gentry's interactive fiction *Anchorhead*.

[1] Parts of this chapter have been previously published as [73, 79, 80, 75].

The author also specifies a set of *DM actions* that the DM can take to intervene in the unfolding experience. Actions can *cause* specific plot points to happen, *hint* in a way that makes it more likely a plot point will happen, *deny* a plot point so it cannot happen, or *undeny* a previously denied plot point. For example, the DM might tell a non-player character to proactively approach the player and reveal some information, thereby causing the plot point associated with the player gaining that information. The set of plot points and DM actions, when combined with a player model, provide an abstract, high-level model of the unfolding experience.

Search-based drama management guides the player by projecting possible future stories and reconfiguring the story world based on those projections. An author-specified evaluation function rates the quality of a particular sequence of plot points. Search over possible choices of DM actions and the possible resultant sequences of plot points (subject to a model of what the user is likely to do) is run at each step to choose the DM action that maximizes expected plot quality. This all takes place in an abstract model, connected to the real game by passing messages back and forth, as illustrated in Figure 2: The game tells the drama manager when plot points have happened, and the drama manager tells the game when it wishes to take a DM action.[2]

SBDM rests on two fundamental assumptions: That an evaluation function can encode an author's aesthetic, and that search can be used to effectively guide a game's plot progression. Weyhrauch demonstrated a proof of concept for both assumptions in a small interactive fiction story, *Tea for Three*, but to what extent these results can be generalized, scaled, and extended isn't clear.

The work here investigates generalizing Weyhrauch's in two ways. First, it applies

[2]Weyhrauch uses a game-tree-search metaphor, describing the progression of a drama-managed story as a trading back and forth of "moves" caused by the player and the drama manager. Accordingly, what I call *plot points* and *drama manager actions*, he describes as *user moves* and *MOE moves*, respectively (MOE being the name of his drama manager). Since the player and drama manager are not truly opponents, I prefer to describe the search as forward projection.

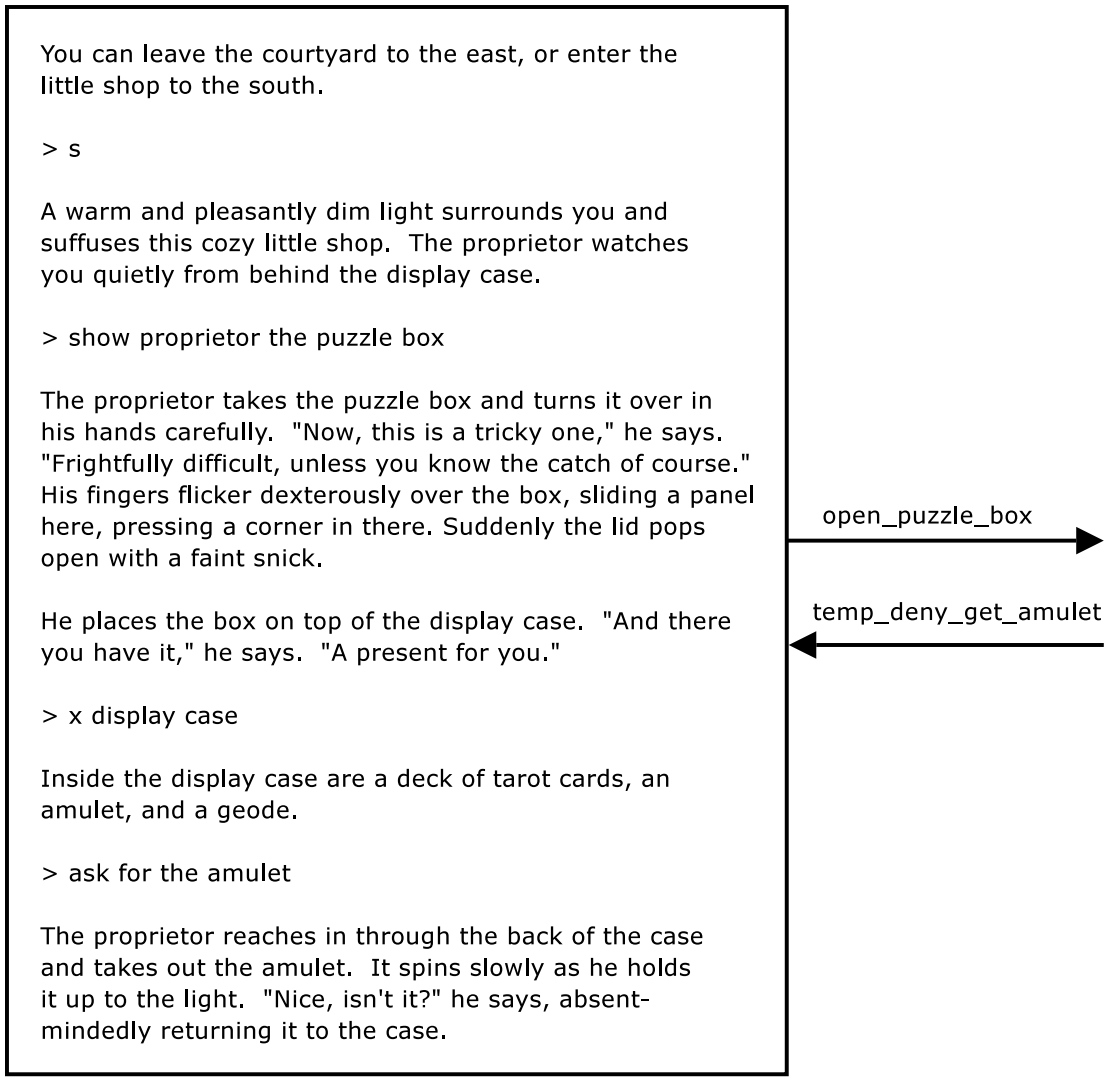


Figure 2: An excerpt from *Anchorhead*, showing the relationships between concrete game-world actions and the abstract plot points and DM actions. When the proprietor opens the puzzle box, the game recognizes this as the plot point `open_puzzle_box` and tells the drama manager. The drama manager decides on the DM action `temp_deny_get_amulet` and sends it to the game, which implements it by not allowing the user to get the amulet.

SBDM to the interactive fiction piece *Anchorhead*, in order to further investigate the algorithmic and authorship issues involved. This experiment concludes that, while SBDM remains promising, Weyhrauch's results were too optimistic, and are not easily generalizable. In particular, search scales poorly to large stories, and the effectiveness of tractable sampling search depends heavily on the nature of the particular story. Secondly, it generalizes SBDM to declarative, optimization-based drama management (DODM). Although SBDM presents itself as an integrated framework, it in fact consists of a declarative knowledge representation—the story graph, annotations, author-suppied evaluation function, and potential opportunities for DM intervention—and a particular way of querying that representation, namely sampling search. The latter is not essential, and other kinds of methods of optimizing this model are possible; I experiment here with reinforcement learning.

3.1 Modeling and search in Anchorhead

Anchorhead[3] is an interactive fiction piece by Michael S. Gentry in the style of H. P. Lovecraft. As compared to the *Tea for Three* story that Weyhrauch investigated, *Anchorhead* has a much larger world, both in terms of the number of plot points and in terms of the physical size of the world itself, such as the numbers of locations and objects available to the player.

When the story starts, the player has just arrived in the town of Anchorhead, where her husband, Michael Verlac, recently inherited the Verlac mansion from a branch of the family he hadn't been in contact with. The player begins to find out strange things about the town and the Verlac family: Edward Verlac, Michael's brother and previous occupier of the mansion, killed his family and later committed suicide in a mental institution; the townspeople are aloof and secretive; the real-estate agent who had overseen the inheritance is nowhere to be found; and so on.

The full *Anchorhead* story is quite large, consisting of well over a hundred significant

[3] Z-machine executable in the Interactive Fiction Archive: `http://www.ifarchive.org/if-archive/games/zcode/anchor.z8`

plot points, making it somewhat unwieldy for initial experiments. Fortunately, it's broken into seven relatively separate days of action, and I've chosen to focus on the second day. I modified the original second day to make it stand on its own, by removing some subplots that only make sense in light of the subsequent days and moving some events from later days forwards. The end result was a story with two main subplots, each potentially leading to an ending.

In one subplot, the player discovers a safe in which a puzzle box she's unable to open is hidden. The owner of the town's Magical Shoppe will helpfully open it, revealing an odd lens. When the lens is inserted into a telescope in the Verlac mansion's hidden observatory, the player sees an evil god approaching Earth on a comet, reaching the climax of the subplot and a possible ending.

In the other subplot, the player discovers that giving a bum a flask of liquor makes him talkative. Through questioning, the player discovers the bum knows quite a bit about the Verlac family, including a terrible secret about a deformed child, William, who supposedly was killed soon after birth. The bum grows anxious and refuses to give more information until the player finds that William's coffin contains an animal skeleton. Upon being shown the animal skull, the bum confesses that William is still alive, and confesses his role in the matter. The bum reveals who William is and some of the background of the Verlac family. Parallel to this progression, the bum is afraid for his life and desires a protective amulet the player can get from the shopkeeper; if the player gives it to him, he'll in return give the player a key to the sewers, in which will be a book revealing the full background, and forming the other possible ending.

3.1.1 Modeling a story with plot points

The first authorial task when applying SBDM (or DODM) is to abstract the contents of the story into discrete plot points, each of which represents some event in the story that the drama manager should know about. Sequences of these plot points form the abstract plot

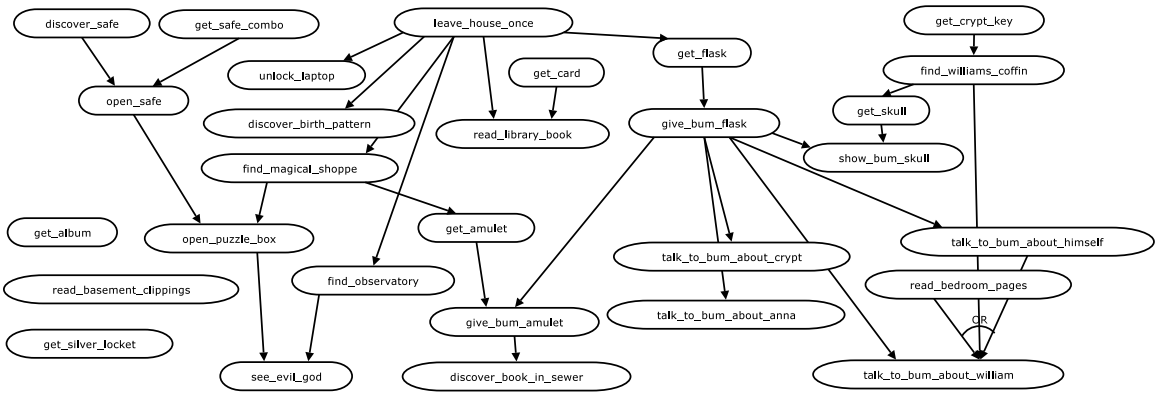


Figure 3: Plot points modeling *Anchorhead*'s Day 2, with ordering constraints. A directed edge from *a* to *b* indicates that *a* must happen before *b*, unless multiple edges are joined with an OR arc.

space through which search will take place.

The plot points are assigned ordering constraints, so that the drama manager only considers possible sequences that could actually happen; for example, the plot point `open_safe` can only happen after both the plot points `discover_safe` and `get_safe_combo` have already happened. (These ordering constraints specify only what *must* happen based on the actual mechanics of the game world—sequences that are undesirable but possible are another matter.)

Weyhrauch specifies these ordering constraints by placing all the plot points in a directed acyclic graph (DAG), with the edges specifying ordering. The possible sequences of plot points are then just the topological orderings of the DAG. We extend this representation by using an AND-OR graph instead of a DAG, allowing for disjunctive constraints in addition to the conjunctive ones a DAG allows; for example, the plot point `talk_to_bum_about_william` in *Anchorhead* can only happen once the player has been told of William's existence, but there are three different plot points that can satisfy this requirement. (An obvious further extension is to allow full boolean constraints, but that hasn't proven necessary for this particular story.)

Figure 3 shows the AND-OR graph I used to model *Anchorhead*'s Day 2.

3.1.1.1 Level of detail

A major issue in choosing a set of plot points is the level of detail at which the story should be abstracted. For example, a conversation could be a single plot point, `conversation_happens`, or it could be a set of plot points for the major possible conversation topics, or in the extreme case there could be a plot point for every possible line of dialog.

As might be expected, there are tradeoffs between fine and coarse modeling. The drama manager cannot make decisions about plot components not represented as plot points, so the more plot points, the more decisions the drama manager can make. However, each added plot point increases search time, and so in a time-limited environment, decreases search accuracy. In addition, including many relatively unimportant plot points tends to make evaluating plot sequences more error- and noise-prone, as the important plot points are obscured amongst the rest (barring a perfect evaluation function). This leads to the heuristic that any plot point you might conceivably want to change (*i.e.* cause to happen, prevent, or otherwise modify) with a DM action should be represented, as should any plot point that will have a significant impact on the quality of the story (and so should be visible to the evaluation function); all others should be omitted. This is of course a subjective judgment, and some experimentation is likely the best way to arrive at a reasonable level of detail.

An additional consideration for the present is that, following Weyhrauch's model, all plot points are considered equally important and equally likely. Thus maintaining a fairly uniform level of detail in modeling will tend to lead to better results: If, for example, one conversation is modeled as a single plot point and another conversation is modeled as ten, each individual plot point in the second conversation will be seen by the evaluation function as being as important as the entire first conversation, and similarly will be modeled as equally likely to take place (of course, this might sometimes be the desired behavior). Future extensions to address this problem could include either weighting plot points with importance values, or modeling stories with a hierarchical space of plot points.

3.1.2 Choosing a set of DM actions

The next authorial issue is choosing a set of DM actions, the "tools" the drama manager will have to work with. There are various types of conceivable actions: They could prevent things from happening; cause them to happen; give hints; and so on. Of course, an action should not simply make strange things happen in front of the user's eyes. If the user hasn't yet found the safe, for example, we can just make it disappear so they'll never find it, but if they've already seen it, we need to be more careful. How to design unintrusive DM actions depends a lot on the story world; one generalization is that it's much easier to do with plot points involving characters, since they can often be plausibly made to start conversations, perform actions, and so on.

3.1.2.1 Types of DM actions

For modeling *Anchorhead*, I investigated five types of DM actions:

Permanent deniers change the world so that a particular plot point becomes simply impossible for the duration of the game. For example, if `find_safe` hasn't happened yet, we can prevent it from ever happening by changing the bookcase with a loose book (behind which the safe is hidden) into just a normal bookcase.

Temporary deniers also change the world so a particular plot point becomes impossible, but only temporarily: Each comes with a paired *undenier* (or *reenabler*) DM action that makes the plot point possible again. For example, `find_safe` might be reenabled by hiding the safe in some other location the user hasn't yet been to.

Causers simply make a plot point happen. For example, the bum in *Anchorhead* could volunteer information about his past, thereby causing `talk_to_bum_about_himself` to happen.

Hints make a plot point more likely to happen, with an associated multiplier and duration. For example, if the bum tells the player that the crypt key is hidden in the basement of the house, it increases the chances that one of the next few plot points will be

`get_crypt_key`.

Game endings are a special type of DM action that ends the game. These are included so that stories can have multiple endings, which are triggered by the drama manager using the same criteria it uses for its other decisions.

3.1.2.2 Issues in specifying DM actions

The first issue encountered was that, in a large world like *Anchorhead*, not every DM action is appropriate at any given time. The *Tea for Three* world is fairly small, so this was a reasonable assumption, but in *Anchorhead*, it hardly makes sense for the DM to request, for example, that the bum bring up a particular topic in conversation when the player is not even remotely near the bum in the world. As a first step in remedying this, we've added two possible constraints on DM actions: *must-follow* and *must-follow-location*. A *must-follow* constraint allows a DM action to be chosen only immediately after a particular plot point; this is particularly convenient for endings, which usually only make sense to trigger at a specific point. A *must-follow-location* constraint allows a DM action to be chosen only immediately after a plot point that happens in a particular location; for example, we can constrain any DM actions that cause the bum to do something to be legal only following plot points that occur in the bum's vicinity.

An additional issue is that making DM actions too powerful can have negative consequences. This is particularly an issue with permanent deniers, since they force story choices of potentially major consequence that cannot then be undone. If a particular plot-point denial maximizes outcome in, say, 90% of cases, but the user's playing causes the story to unfold into one of the other 10%, then there is little to do to recover and still push the story towards a reasonable conclusion. Therefore, temporary deniers are preferable, since they can always be undone if necessary; however, permanent deniers are still worth considering, as some potentially useful deniers are very difficult to make undoable (short of an undesirable *deus ex machina* style of drama management).

3.1.3 Specifying an evaluation function

An evaluation function encodes the author's story aesthetic declaratively, which is one of the main attractions of a declarative approach to drama management. The author simply specifies the criteria used to evaluate a given story, and the drama manager tries to guide the story towards one that scores well according to that function. In the process of doing so, it makes complex tradeoffs—difficult for an author to manually specify in advance—between possibly conflicting authorial goals (as specified by components of the evaluation function), while taking into account the player's actions and incorporating them into the developing story.

3.1.3.1 Toolbox of features

In order to ease authoring, an author can choose from a toolbox of features representing common authorial goals. To make weighting goals straightforward, all features range from 0.0 to 1.0, so an author can specify an overall evaluation function as a weighted combination of the features.

For *Anchorhead*, I used seven features to construct the evaluation function, all of which are designed to be applicable to any story where the goal the feature encodes would be desirable.

General features

Three features specify general properties we'd like our stories to have.

Location flow is a measure of spatial locality of action: The more pairs of plot points that occur in the same location, the higher the score. This feature is based on a judgment that wandering constantly around the world is undesirable.

Thought flow is calculated similarly to location flow, but measures continuity of the user's thoughts, as specified by an optional *thought* annotation on plot points. This feature can be seen as preferring coherent "sub-subplots"; for example, `get_safe_combo` and `discover_safe` both have a thought of `safe`, so the thought flow feature would prefer

plots in which the user finds the safe and then looks for the combination (or vice versa), rather than finding the safe, getting distracted by something else, and then finding the combination later.

Motivation is a measure of whether plot points simply happened out of nowhere, or happened after other plots points that motivated them in the player's mind. This is a subjective determination of the author; for example, finding the observatory (`find_observatory`) and noticing that the telescope is missing a lens would make opening the puzzle box to find a lens inside (`open_puzzle_box`) motivated, while finding an unexpected and unexplained lens wouldn't be a motivated plot point.

Features for stories with multiple endings

With multiple subplots leading to multiple potential endings, two additional features can evaluate interactions between the plots.

Plot mixing measures to what extent the initial part of the story includes plot points from multiple subplots. We'd like the user to explore the world in the beginning, rather than finding one of the plot sequences and going straight to one of the endings.

Plot homing measures to what extent the latter part of the story includes plot points uniformly from the same plot. This is a counterpart to the plot mixing feature: While we don't want the user to go straight to one plot and finish the game right away, we do want them to do so eventually, rather than continually oscillating between plots and then stumbling upon one of the endings.

Meta-features

The final two features are intended to rate the impact of drama management on a story rather than rating the story itself.

Choices is a measure of how much freedom the user has to affect what the next plot point will be. The goal is to allow the user as many choices of action at any given time as possible, rather than *e.g.* using a lot of deniers to prevent the user from doing anything undesirable. (Without this feature, a drama manager with access to a lot of deniers would

basically linearize the story, making the best story as judged by the other features the *only* possible story, which would defeat the entire purpose of an interactive experience.)

Manipulativity is a measure of how manipulative the drama manager's changes in the world are. The author specifies a manipulativity score for each DM action, encoding a judgment of how likely it is to be noticed by the user as something suspicious going on (subtle hints might be judged to be less manipulative than outright causers, for example).

3.1.4 Searching and results

Since the goal of drama management is to improve the quality of experiences over a whole range of possible player behavior, not to improve any one particular run, success would mean that the distribution of story scores under drama-managed play is shifted upwards as compared to the distribution with no drama management. This can be tested by generating and scoring random plots to construct the unmanaged distribution, and by running drama management with simulated users to construct the managed distribution. In the experiments reported here, non-drama-managed distributions were constructed from 10,000 samples each and drama-managed distributions from 100 simulated runs each; all histograms use a bin width of 0.02.

Once plot points, DM actions, and an evaluation function are specified, search through this abstract plot-point space can yield the optimal DM action for any given situation. The problem that immediately arises is that actually performing a complete search over all possible future combinations of DM actions and plot points is computationally infeasible, since the search space's size grows exponentially with the size of the story.

In *Tea for Three*, Weyhrauch implemented a memoized full-depth search, taking advantage of symmetries in the search space to collapse the entire search tree into a table of approximately 2.7 million nodes. However, the memoized search is relatively difficult to author for, since the way to construct the table depends on the particular combination of evaluation features used, and would have to be recoded each time features changed (a

process less appealing than specifying an declarative evaluation function to be used by an unchanging search process). In any case, as Weyhrauch notes, even the memoized search doesn't scale well: In our model of *Anchorhead*'s day 2, the number of table entries would be in at least the hundreds of millions, requiring a table gigabytes in size.

More promisingly, Weyhrauch reported surprisingly good results with SAS, a sampling search. SAS performs search to a specified depth (adapted here to a version that performs iteratively deepening until a time limit), and then obtains a score by averaging together a fixed number of samples of complete plots that could follow the cutoff point. SAS+ is a variant that allows temporarily denied plot points to appear in the samples, under the assumption that they could be reenabled at some point in the future (necessary in order to prevent the possibility of stories in which no ending is reachable). In *Tea for Three*, the mean quality of stories produced through SAS+ with a depth limit of 1 was at the 97th percentile of the unmanaged distribution, nearly equalling the 99th percentile obtained by the memoized full-depth search.[4]

The performance of SAS+ on this model of *Anchorhead*, on the other hand, is much less impressive. Figure 4 shows the distribution of plot scores in: an unmanaged story; a story managed by SAS+ with a simulated user ignoring hints; and a story managed by SAS+ with a simulated user probabilistically following hints as the drama manager expects. With the user ignoring hints, the mean score is at the 64th percentile and the median at the 59th; when the user follows hints probabilistically as expected, the mean is still at the 64th percentile, and the median at the 63rd. This is still successful (the overall curve is shifted upwards), but less impressively so than in *Tea for Three*, indicating that the SAS+ results from that story aren't generalizable.

The distributions of scores is interesting to note. The worst stories are reduced in frequency, as we'd like, but not avoided entirely, and the main peak is roughly where the main

[4]Since the raw scores are arbitrary numbers, percentiles are reported as a useful relative measure.

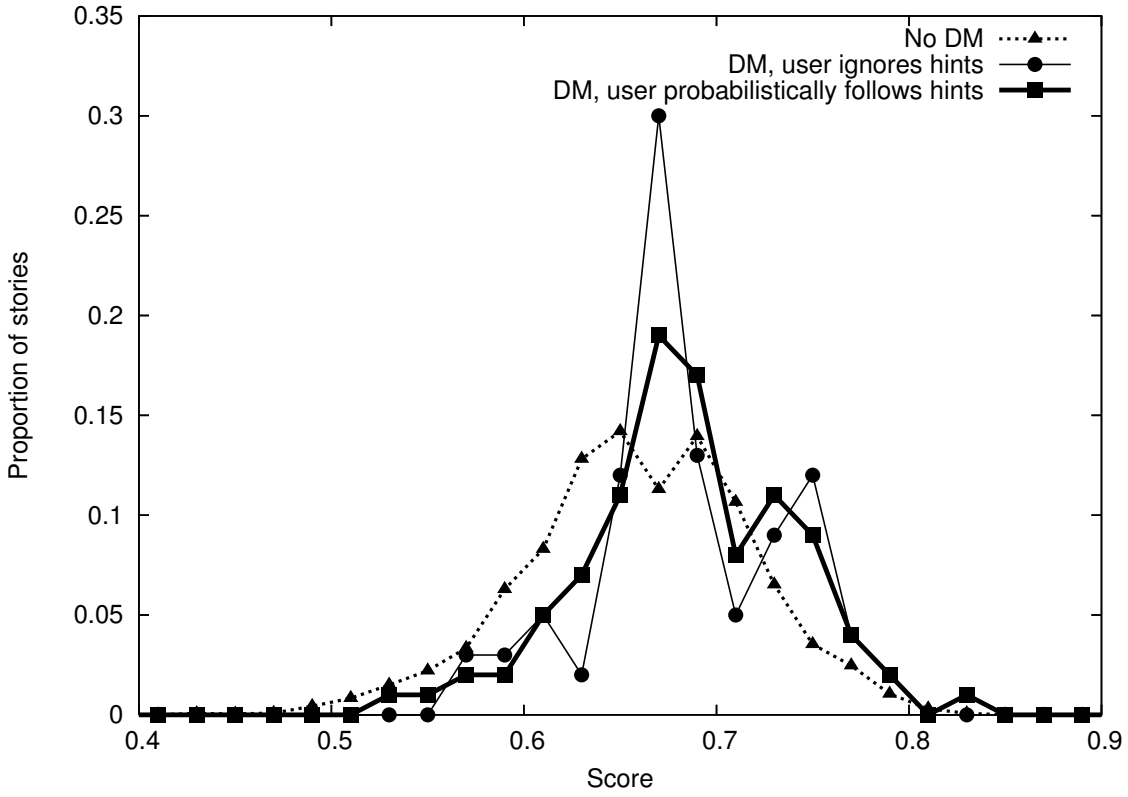


Figure 4: Distribution of plot qualities with and without drama management. The drama-managed runs were done with SAS+ limited to 2 seconds per decision.

peak in the unmanaged distribution is, shifted slightly to the right. A new peak also appears at around 0.73—the 90th percentile—indicating that in many (but not most) cases the drama manager is capable of bringing about one of the best stories. (The *very* best stories, however, are not noticeably increased in frequency.)

Indeed, we wouldn't expect SAS+ to achieve results anywhere near the 97th percentile reported by Weyhrauch in general: With shallow search depth, a sampling search of this sort is essentially doing local, "greedy" search, at each point choosing the DM action that maximizes the average future plot score under the assumption that no further DM actions will be taken. Since the entire point of SBDM is to maximize score taking into account the possibility of future DM actions, this is a significant handicap.

To take a simplified example: Say that plot point *a* has already happened, and the DM is choosing between a DM action that causes *b* to happen, and one that causes *c* to happen. If all plots starting with *a b* end up with a score of 0.6, while of the plots starting with *a c*, half are scored 0.0 and half 1.0, 1-level SAS+ will choose the DM action that causes *b* to happen, since the average of 0.6 is better than the average of 0.5. It will make that choice even if the 0.0-score stories starting with *a c* could be easily avoided with future DM actions, since deeper search would be required to discover that fact.

The limitation is particularly problematic for DM actions that need to be paired to be effective, such as temporary deniers and their corresponding reenablers.

To determine whether the results here are worse than Weyhrauch's because of the set of DM actions, I ran an experiment with causers, temporary deniers, and reenablers for each plot point. These "synthetic" DM actions (synthetic because only a subset are plausibly implementable in the real story world) ought to give the drama manager as complete control as possible over the story. However, as Figure 5 shows, the performance with this set of DM actions is actually *worse* than not using drama management at all! This would be impossible with a search reasonably approximating full-depth search, because even in the worst case the drama manager could avoid actually worsening a story by simply choosing

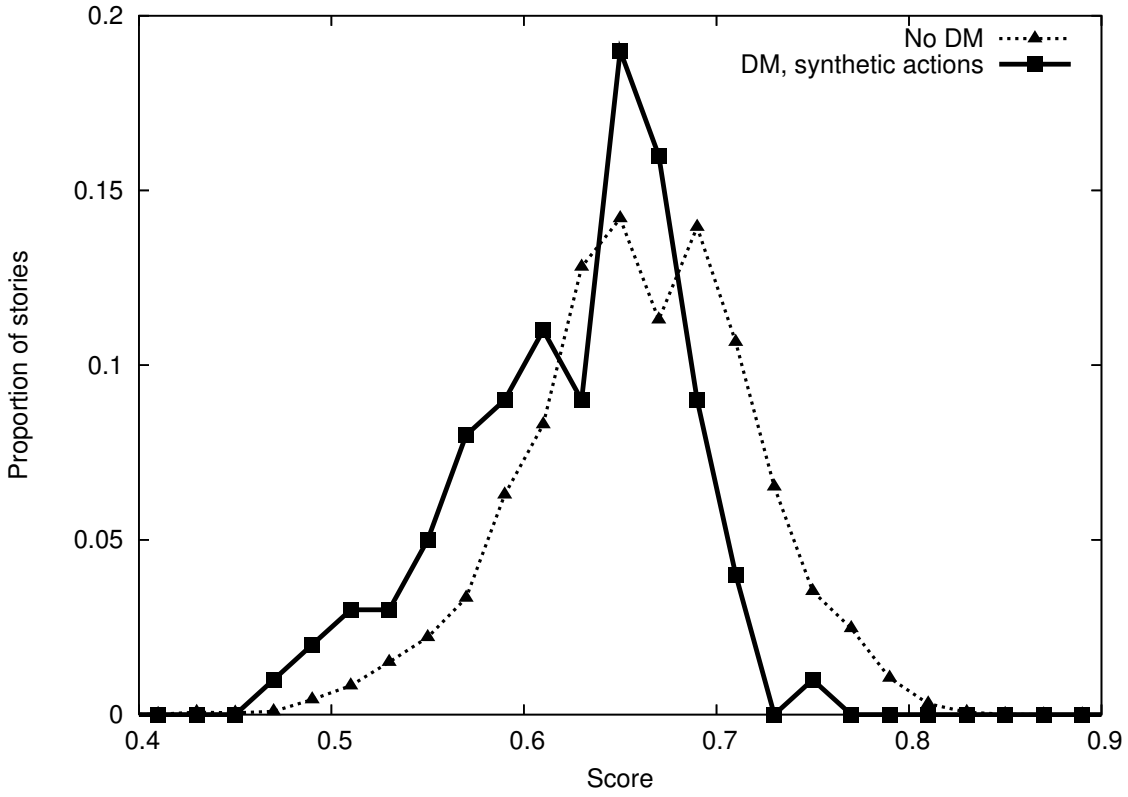
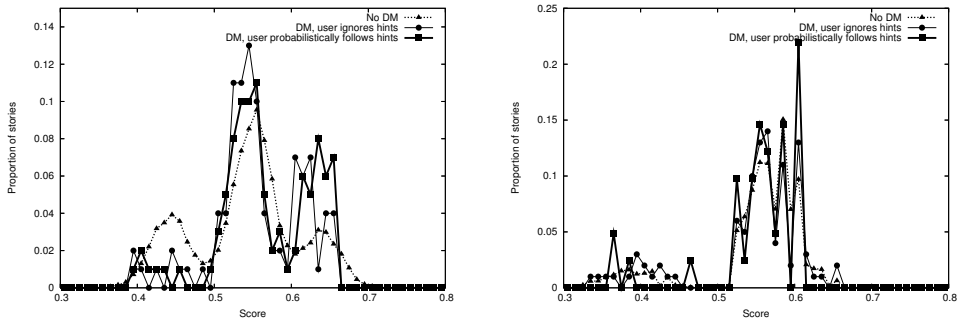


Figure 5: Distribution of plot qualities with and without drama management, using synthetic DM actions for the drama-managed runs (see text). The drama-managed runs were done with SAS+ limited to 2 seconds per decision.



(a) `discover_book_in_sewer` ending. (b) `see_evil_god` ending.

Figure 6: Distribution of plot qualities with and without drama management on the subplots considered separately. The drama-managed runs were done with SAS+ limited to 2 seconds per decision. Histogram bin width is 0.02.

to never take any action. Clearly, then, the difference in distribution quality is due to SAS+ being ineffective on this model of *Anchorhead*.

In order to untangle the effects of multiple endings, I also tried each plot separately, turning off the plot homing and plot mixing features and keeping only the plot points and DM actions relevant to each. The results in figure 6 show that the drama manager is much more successful at improving the quality of the stories with the `discover_book_in_sewer` ending than those with the `see_evil_god` ending. One possible reason is that the storyline ending with `see_evil_god` mostly includes temporary deniers and reenablers as its DM moves, which local search is bad at using: Much like a chess program that searches only 3 moves into the future can't plan 5-move combinations, SAS-style search can't effectively plan using spaced-out pairs of temporary deniers and reenablers to delay plot points.

3.1.5 Conclusions for search

Search-based drama management is a conceptually appealing way of casting the drama-management problem. However, the previously-reported results for shallow sampling search were too optimistic. Exponential explosion in the size of the search-space makes brute-force full-depth search infeasible. Unfortunately, more tractable shallow sampling searches

don't always perform well, and in some cases perform particularly badly. SAS+ does positively impact the quality of *Anchorhead* story, but not as effectively as in Weyhrauch's story, indicating that his positive SAS+ results don't generalize to arbitrary stories. In many ways this is to be expected: The whole point of SBDM is force the drama manager rather than the author to perform complex tradeoffs among story evaluation features.

3.2 Optimization by reinforcement learning [5]

As seen in the *Anchorhead* experiments, deep search will be required to perform nontrivial tradeoffs, especially those that involve dependencies among longterm-term story elements, which are precisely the interesting cases for drama management. Framing drama management as search does have a certain appeal, as it makes the large body of search techniques and optimizations developed over decades of research available for drama management. However, once we recognize that search-based drama management is two separate pieces—a formal modeling framework and a method for querying that model—we need not try only search.

If one were to approach the formal model from a statistical machine learning perspective, the obvious tool to try would be reinforcement learning (RL). RL algorithms attempt to learn a policy that, given a world modeled by states and a (possibly unknown) transition function between states, specifies an action to take in each state in a manner that maximizes expected evaluation according to some function. In this case, state is specified by the sequence of plot points and drama-manager actions that have happened so far (order is important); actions are the drama-manager actions plus the null action (do nothing); transitions are the likelihood of plot points occurring in any given state, and are given by the combination of prerequisites for a plot point, the player, and any influence from drama-manager actions; and the evaluation function is the one provided by the author.

Although nicely defined, this is a complicated reinforcement-learning problem: the

[5] The experiments reported in this section are joint work with David L. Roberts.

player is difficult to model and highly stochastic, and the state space is extremely large. This necessitated developing a modified training regime, self-adversarial / self-cooperative exploration (SASCE), discussed below. I compare three approaches to optimizing the selection of DM actions: the limited-depth sampling search (SAS+) proposed by Weyhrauch; standard temporal-difference (TD) learning; and TD learning with a modified training regimen (SASCE). I evaluate the results on two different stories: one a variant of the *Tea for Three* story used by Weyhrauch, and the other the *Anchorhead* story used above, and a more detailed analysis on a subset of *Anchorhead*.

3.2.1 TD learning

Temporal-difference learning [126] is a reinforcement-learning technique that, over many thousands of simulated games, estimates the values of story states (a story state is either a partly- or fully-completed story). For completed stories, the state value is simply what the evaluation function says it is; for partial stories, it is the expected value of the complete story (*i.e.* the average of all stories that are continuations of that partial story, weighted according to likelihood). The drama manager's policy is then to simply pick whichever DM action leads to the highest-valued successor state.

TD learning builds its value estimates for each state based on the current estimate of its successor, each time the state is encoutnered in a simulation run. Since at the end of each story we can use the evaluation function to get the real value for the last state, the real values propagate backwards, and all states' estimates should eventually converge to the true values. Since there are far too many possible story states to store the value estimates in a lookup table, I follow Tesauro [127] in training a neural network as a function approximator to estimate a function from states to their values.

In story-based games, the order of plot points is of course important, so the state cannot simply include what has happened, but also the order in which it happened. Instead, state is encoded as an $n \times n$ precedence matrix, M, where n is the number of plot points and $M_{i,j}$

indicates whether plot point i happened after plot point j.[6] For example, in a story where there are four possible plot points, and in which plot points 3, 4, and 2 have occurred so far (in that order), the precedence matrix would look like:

$$M = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

3.2.2 SASCE

Self-adversarial / self-cooperative exploration (SASCE) is a novel modified training regimen for TD learning. The experiments reported below using TD learning both with and without SASCE were identical except for the player model used for training. Typically in machine learning we assume that training data is drawn from the same distribution as the test data. In this domain the TD learner would therefore be trained on a player similar to the one it is expected to encounter when deployed. In the present experiments, this is the player exploring the world.

Instead, the SASCE player model uses the drama manager's current value function to select its actions. Because the SASCE player model uses the agent's state-value estimates, it can choose to act cooperatively or adversarially. A cooperative player model would select actions that put the player in the highest-rated subsequent state, while the adversarial player would move to lower-rated states. The basic idea behind SASCE is to convert the asymmetric non-adversarial drama management problem into a *pseudo-adversarial* or *pseudo-cooperative* problem. This feedback loop is intended to force meaningful exploration of the state space.

[6] This bears some resemblance to the more common bigram encoding, which specifies for each pair of events i, j whether that *sequence* appeared in the history. The precedence matrix trades off some information about exact sequences for a gain in information about global ordering.

The player model selects states based on an exponential distribution:

$$p(s_i) = \frac{e^{\alpha\beta(s_i)V(s_i)}}{\sum_{s_j \in S} e^{\alpha\beta(s_j)V(s_j)}}.$$

Here s is a state; $V(s)$ is the agent's current state-value estimate; α is a parameter controlling the "desire" of the model to cooperate (large positive values are more strongly cooperative and large negative more adversarial); and $\beta(s)$ is function that responds to the "desires" of the drama manager. In the experiments here, $\beta(s) = 1.0$ for all states except for those the manager has explicitly attempted to increase the likelihood of (*e.g.* by providing a hint). In those cases, $\beta(s)$ is some constant $c_a > 1$ associated with action a.

While α is a constant parameter, it can be adjusted to achieve different player-model behaviors. For example, for large negative values of α, the player will be very adversarial, selecting the lowest-valued states (according to the state-value estimates) almost exclusively. Similarly, for large positive values of α, the player will select the higher-valued states almost exclusively. With $\alpha = 0$, the player will be uniformly random.

In practice, a *mixed strategy* is selected. The mixed-strategy player has three parameters. It has both cooperative and adversarial values for α and a probability (q) that the player will use the adversarial value of α. The goal of the mixed strategy is to keep from learning a worst-case policy: Since we don't expect the player to *actually* be acting adversarially, we don't want the policy to optimize only for that case.

3.2.3 Results

The experiments are as follows. First, results for *Tea for Three* and a subset[7] of *Anchorhead*, demonstrate that TD learning outperforms SAS+ on complex domains, and further that using SASCE improves TD learning's performance on both relatively simple and complex domains. Analysis of this subset of *Anchorhead* gives some clues as to why the policies behave as they do. Then, experiments on the full *Anchorhead* model in Figure 3 show that

[7]The subset includes all the plot points from one major subplot, and all DM actions relevant to those plot points.

Table 1: Summary of results using SAS+ search and TD learning with and without SASCE. Numbers are in percentiles relative to the non-drama-managed distribution of story qualities (see text).

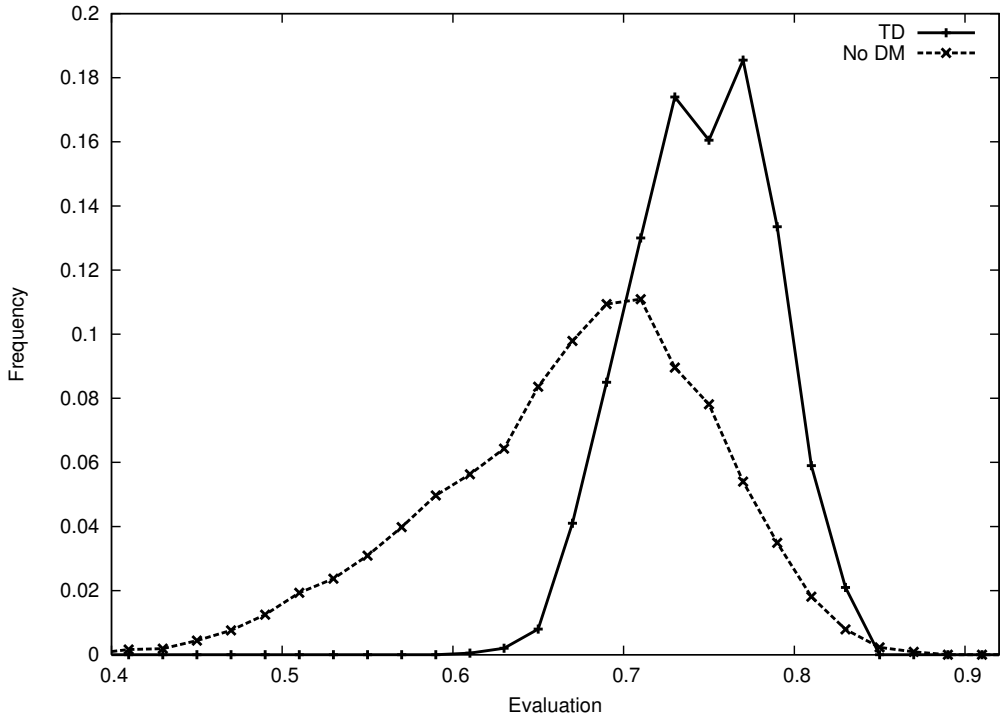
	Tea for Three		***Anchorhead* subset**	
Method	Mean	Median	Mean	Median
SAS+	**96.3%**	**96.7%**	42.5%	50.5%
TD	81.2%	81.8%	80.4%	70.5%
TD/SASCE	91.3%	93.3%	**82.8%**	**83.6%**

none of the policies perform well, but demonstrate that adding in "synthetic" DM actions improves performance markedly, illustrating some potential authorship pitfalls.

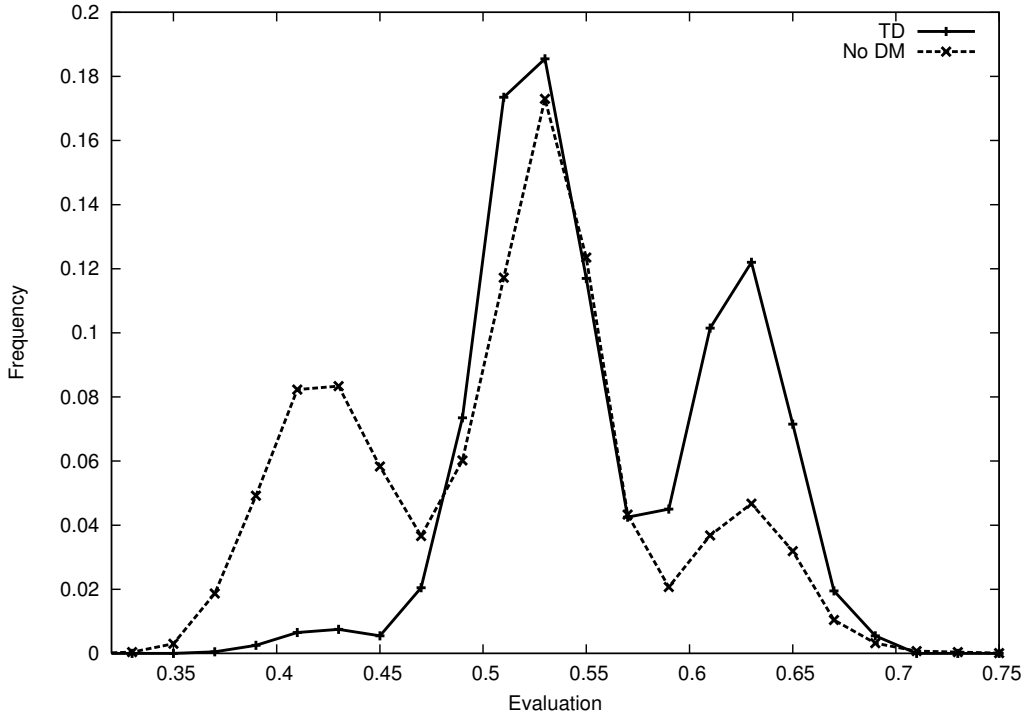
3.2.3.1 Tea for Three *and an* Anchorhead *subset*

Table 1 summarizes the performance of SAS+ sampling search, TD learning trained conventionally, and TD learning trained using the SASCE training regimen, each evaluated on both *Tea for Three* and a subset of *Anchorhead*. Performance is summarized by calculating the mean and median story quality of each drama-managed distribution, and reporting that in terms of a percentile of the non-drama-managed distribution (*e.g.* a percentile of 70% for the mean would say that the average story quality using drama management is better than 70% of stories without using drama management).

There are three interesting things to note. First and foremost, training TD using SASCE consistently improved its performance by all measures on both stories. Secondly, while search does marginally better than SASCE on the simpler *Tea for Three* story, its performance degrades significantly on the *Anchorhead* subset—which has a more complicated relationship between story structure and evaluation—while the RL methods continue to perform well. Finally, SASCE greatly improves median performance on the *Anchorhead* subset in comparison to conventionally-trained TD learning.

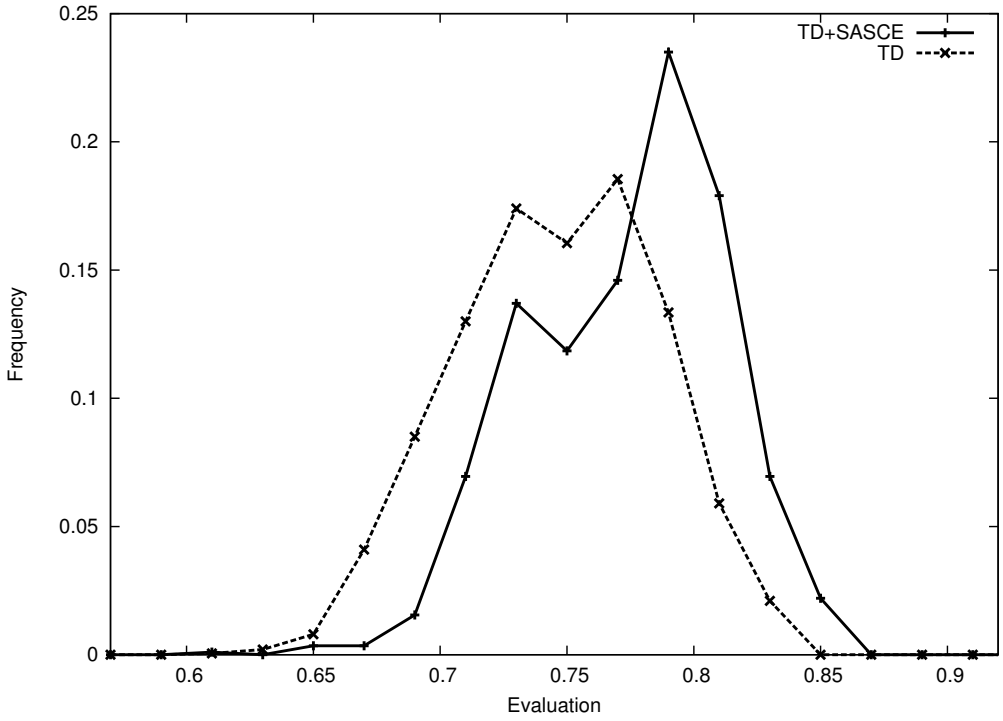


(a) *Tea for Three*

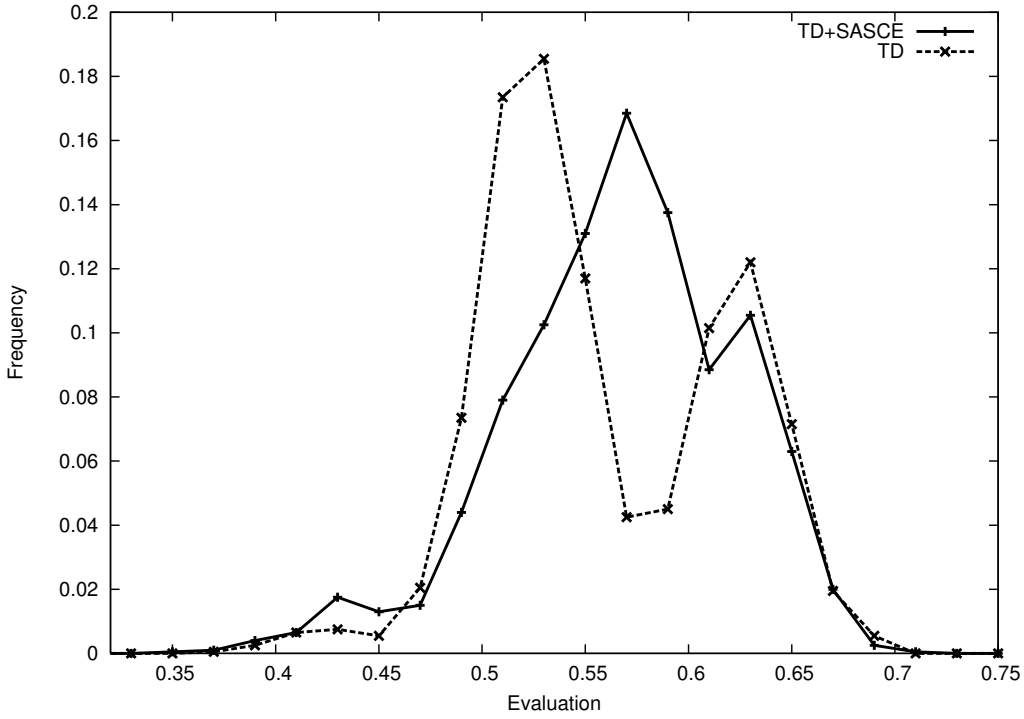


(b) *Anchorhead* subset

Figure 7: Story-quality distributions for conventionally-trained TD learning versus no drama management.



(a) *Tea for Three*



(b) *Anchorhead* subset

Figure 8: Story-quality distributions for TD learning trained using SASCE versus trained conventionally.

Figure 7 shows the detailed distributions on each story both without drama management and using TD learning without SASCE. Notice in the non-drama-managed distributions that the stories are qualitatively quite different: *Tea for Three* has a relatively smooth unimodal distribution, while the *Anchorhead* subset is distinctly trimodal. This suggests that there are points in the *Anchorhead* subset that have significant influence on the ultimate outcome. It is at these points that SASCE learns to make better decisions. This also suggests why search performs so poorly in the *Anchorhead*-subset domain: its relatively short horizon doesn't allow it to effectively make long-term global decisions. As a result it only performs well in "easy" domains where local decisions can result in globally good results.

While not shown in a figure, the story-quality distribution using search on *Anchorhead* remains trimodal like the distribution of randomly-generated stories, lending support to the hypothesis that search fails to find the important decision points that steer between the different types of stories. By contrast, the TD-learned policy's distribution is bimodal and almost entirely missing the stories that form the lowest of the three modes (see Figure 7(b)), indicating that it successfully learns how to avoid the worst batch of stories.

The policy trained with SASCE does even better, with a unimodal distribution that is shifted higher (see Figure 8(b)), indicating that it is not missing any major decision points. To give a more detailed idea of how SASCE improves TD learning, Figure 8 shows the full distributions of story qualities using each method on both stories we evaluated. Remember, qualitatively the goal is to shift the distribution to the right as far as possible, reducing the frequency of lower-scored stories and increasing that of higher-scored stories. Notice how in Figure 8(a), SASCE is consistently better: On the lower side of the distribution (below the mean) its frequencies are lower, while on the higher side they're higher. In Figure 8(b), the improvement takes on a different shape, keeping roughly the same proportion of highly-scored stories, but increasing the quality of the lower- and middle-scored stories significantly, pushing the main peak (and the median) towards the upper end of the range.

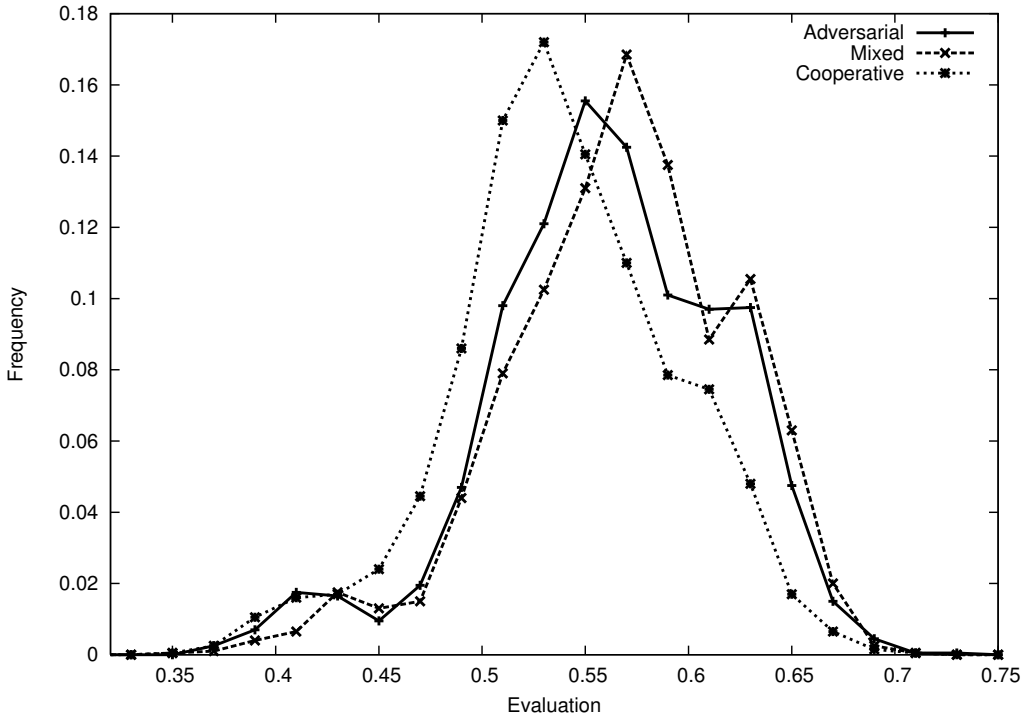


Figure 9: Story-quality distributions on the *Anchorhead* subset using different values of q for training. The three values shown are $q = 0.0$ (cooperative), $q = 0.5$ (mixed), and $q = 1.0$ (adversarial).

Table 2: Average number of plot points occurring in the *Anchorhead*-subset story before policies disagree on which action to take.

Adversarial			Mixed		TD
Mixed	**TD**	**Search**	**TD**	**Search**	**Search**
5.6	5.1	1.4	4.8	1.4	1.4

Recall that, using SASCE, a player model can be adversarial (negative α), cooperative (positive α), or mixed (choosing an adversarial player with probability q, and a cooperative player with probability $1 - q$). Interestingly, all mixed models except the fully cooperative player ($q = 0.0$) improved TD learning, both quantitatively and qualitatively. In the examples shown here, as in most of the experiments, training on the mixed player gave the best performance, as shown in Figure 9, although the mean and median are not significantly different. Training on the fully cooperative player consistently gave the worst performance. All types of SASCE players display a roughly unimodal histogram, in stark contrast to the bimodal histogram seen in conventionally-trained TD learning, and the trimodal distribution of the unmanaged stories (Figure 7(b)). This lends some further support to the hypothesis that the modes are due to distinct decision points.

Intuitively, training against an (even occasionally) adversarial player forces the drama manager to learn how to maximize the worst-case performance. That is, the manager learns how to take actions that will prevent the player from arriving in a state where it can then easily do harm to the story. This proves useful for a wide variety of actual player types, including players who are exploring. A cooperative player acts as an antidote for those cases where such states can also lead to remarkably good outcomes.

It is worthwhile to try to understand in more detail how the policies learned by the various methods differ. To that end, they can be used simultaneously to guide the simulated player until they reach a point at which they disagree on which action to take. Table 2 shows the average number of plot points occurring prior to disagreement for each pair of

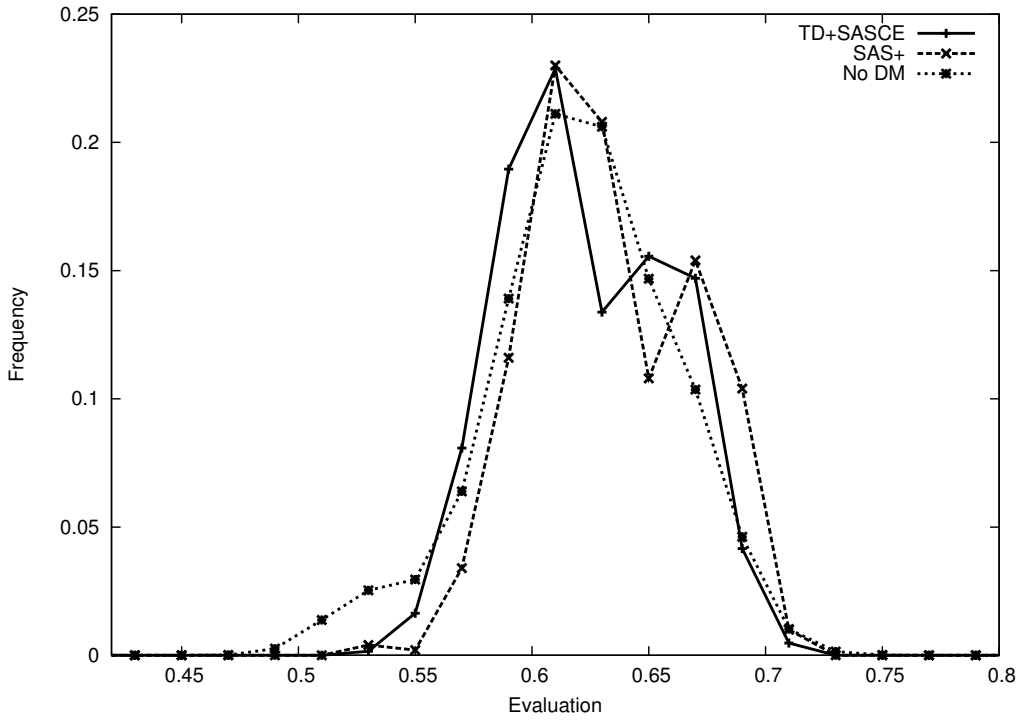


Figure 10: Story-quality distributions on *Anchorhead* for SASCE-trained TD learning, SAS+ search, and no drama management.

policies. Search clearly makes different decisions very early. TD without SASCE diverges from SASCE-trained TD around the fifth plot point. Interestingly, the policies trained with adversarial and mixed players diverge not much later, but have much more similar distributions, suggesting that the important decisions have already been made by the fifth or sixth plot point (on average).

3.2.3.2 Anchorhead

On the larger portion of the *Anchorhead* story, which includes two subplots that may interleave leading to two possible endings, none of the policies do particularly well, as illustrated in Figure 10.

As I did in Figure 5 in the initial *Anchorhead* search experiments, in order to determine whether the policies were performing poorly because of their inability to find a good policy, or because the available DM actions in this state space simply did not permit for a very good

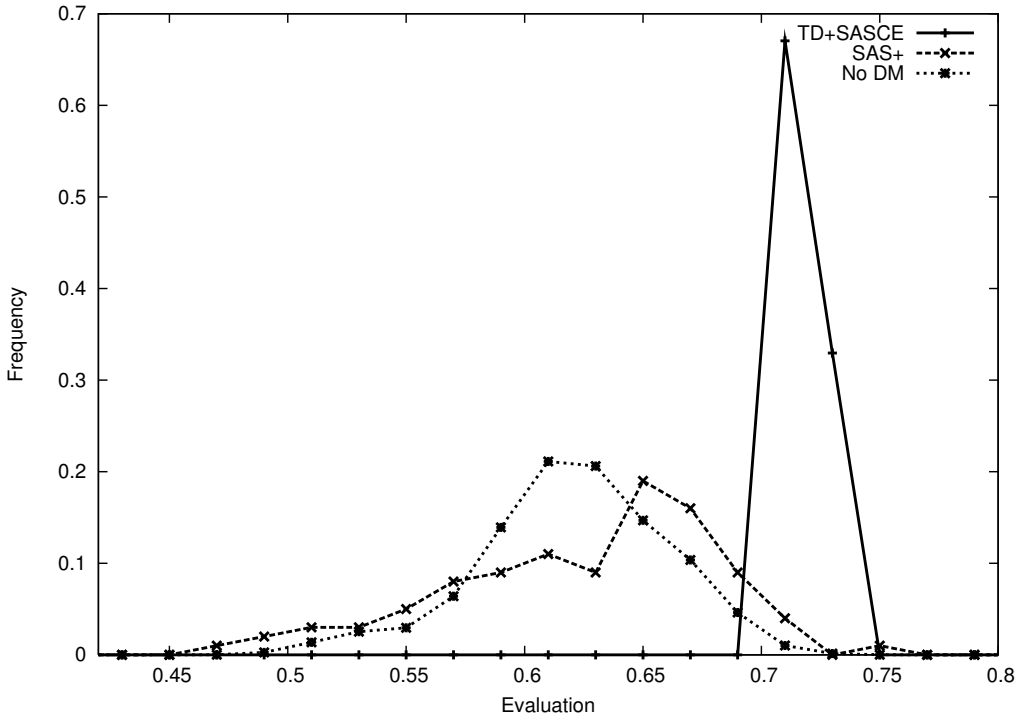


Figure 11: Story-quality distributions on *Anchorhead* with synthetic actions for SASCE-trained TD learning, SAS+ search, and no drama management.

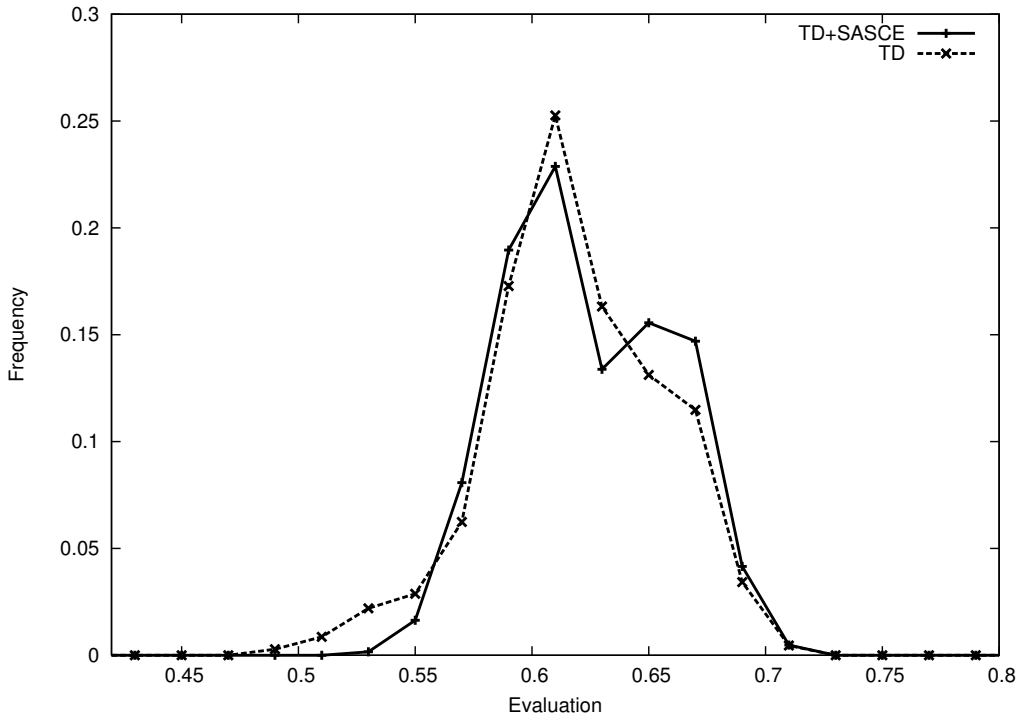


Figure 12: Story-quality distributions on *Anchorhead* for TD learning with and without SASCE.

policy, I ran a "synthetic" test in which there was a causing, hinting, temp-denying, and reenabling DM action for every plot point. This ought to give the drama manager complete control, and indeed as shown in Figure 11, the TD-learned policy is now consistently very good; SAS+ search, however, still does rather poorly, as it is unable to navigate the now even larger state space to make effective use of the added actions. While this does not *prove* that the poor performance without these added actions is due to a lack of sufficient actions, rather than inability to learn a good policy, it does lend some support to that hypothesis, and suggest that the drama manager should be given more actions to work with.

Interestingly, SASCE still noticeably improved the performance of TD learning in this case, as illustrated in Figure 12. Furthermore, mixed-strategy training in particular is what improved performance: the completely adversarial version of SASCE did poorly, as shown in Figure 13. These consistent results across all the stories lend some support to the two hypotheses: that SASCE forces the right type of exploration for this domain, and that a

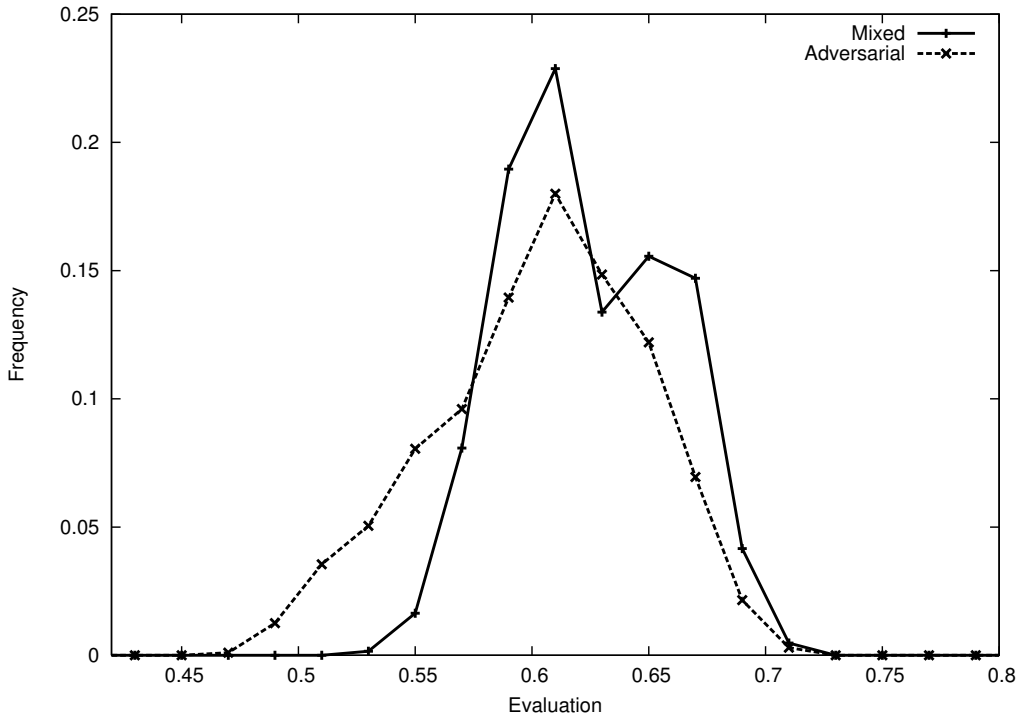


Figure 13: Story-quality distributions on *Anchorhead* for TD learning with mixed-strategy SASCE training ($q = 0.6$ in this case) and completely adversarial SASCE training ($q = 1.0$).

mixed-strategy version of SASCE in particular balances forcing exploration with an adversarial player without learning a policy optimized only for the adversarial case.

3.3 Conclusions

In this chapter, I start with the search-based drama management (SBDM) formalism proposed by Bates and Weyhrauch, and generalize it in two ways. First, I use it to model an existing story considerably more complex than those investigated previously, and in doing so develop an extended modeling framework with a set of reusable primitives for modeling other stories. Secondly, I find that computational performance of sampling search becomes much more problematic in this more complex story, and that motivates the move from specifically *search-based* drama management, to *declarative optimization-based drama*

management (DODM), which foregrounds the declarative modeling component as the center of the framework, and treats the optimization part as a pluggable implementation choice. Experiments then show that plugging in reinforcement learning can result in a considerable improvement on larger stories, particularly when coupled with a novel training regime.

CHAPTER IV

DODM AND PLAYER AGENCY[1]

If we return to Figure 11, there seems to be a dark side in this exceptionally good performance. The drama manager succesfully pushes the distribution of stories so only stories at the top end of the story-quality distribution take place. And the goal as specified was indeed to maximize story quality. But is this type of narrow distribution what we actually want?

When we examine the plots produced, it makes it easier to answer that no, this might not be want we want, as they are nearly all identical. Given its large array of "synthetic" DM actions allowing it to intervene in the story almost at all, the drama manager found a very small range of good stories, and used DM actions to cause those stories to always happen. To maintain player agency, we would prefer that there be a range of possible stories that could emerge during gameplay, especially if replay value is important. In non-synthetic examples this may turn out to not be an issue most of the time, but that is in a sense by luck: the DM was trying to stample out player agency but just wasn't able to in a given case. That suggests that perhaps we should extend the evaluation function so that it takes into account the diversity of possible stories as an explicit goal, rather than rating each story in isolation.

Jumping off from that suggestion, some parallel work (on which I was a collaborator) found this to be a significant flaw in the basic SBDM/DODM approach of maximizing an evaluation function [99]. The basic argument runs as follows. When maximizing an evaluation function, the only source of gameplay variation will be the unpredictability of the player. Given sufficiently powerful DM actions, the system will be able to force its idea

[1] A version of this chapter was previously published as [75].

of optimal story on the player, destroying the interactive agency of the experience.

The proposed replacement, targeted trajectory distribution Markov decision processes (TTD-MDPs), instead start with a desired *distribution of experiences* (trajectories through the story space), and recast the drama manager's goal as one of using the DM actions in a way that would make the actual distribution come as close to the target distribution as possible. Algorithmically, the TTD-MDP system builds a large tree sampled from the space of all possible trajectories; each node in the tree then solves an optimization problem to find a distribution over its available actions that will, according to the player model, cause the resulting distribution over successor plot points to come as close as possible to the distribution specified by the author.[2]

Although I participated in developing TTD-MDPs and consider them to have interesting uses, I argue here that their criticism of DODM is misplaced, at least as regards the framework itself (versus the particular choice of evaluation function in the *Anchorhead* experiments). To see why DODM does not inherently destroy interactive agency even when it optimizes successfully with a powerful set of DM actions, it's worth stepping back a bit and looking at what the goal of a drama manager is, and then reconsider how that might be declaratively expressed as an optimization problem. After all, the fundamental conceptual (as opposed to technical) issue in drama management is deciding what constitutes a good interactive drama. Given criteria for good interactive drama, we can design the DM to try to bring about such an experience.

4.1 Formal and material constraints

Mateas [61] proposes a theory of interactive drama integrating Aristotelian dramatic theory with Murray's [70] desiderata for interactive stories, in particular the goal of interactive agency. He proposes that good interactive drama achieves a good balance of *material*

[2] If "as close as possible" is formalized as KL-divergence, solving this optimization problem at each node locally produces a globally optimal solution [6].

constraints—the "constraints from below" that literally constrain what a player can do—and *formal (plot) constraints*—the "constraints from above" that constrain in the player's mind what, given the experience so far, is interesting, sensible, or worth doing.

The desired balance can be illustrated by contrasts with the extremes. In open-world sandbox and puzzle-based adventure games, the player can take many actions, but there is little plot that would give a reason to do anything in particular or serve as an interpretive framework tying events together. In linearly scripted games, meanwhile, the plot unfolds in a coherent fashion and everything the player does relates to a coherent whole, but many actions that make sense given the story are not supported by the game.

The actions in DODM serve to tinker with the formal and material constraints. Hints add new formal constraints by giving the player some additional narrative framework, without adding any new material constraints (the player may ignore the hint). Causers also add new formal constraints by directly causing some narratively important event to happen, but do so by also adding material constraints, since they temporarily remove the player's freedom to decide what should or shouldn't happen next. Deniers, meanwhile, add new material constraints, which can later be removed by undeniers.

4.2 Maximizing experience quality

The original DODM approach has an evaluation function that, given a completed experience (a sequence of plot points and DM actions), rates it based on various features that the author thinks an experience should have. The evaluation function therefore should be written so that it specifies to the DM what constitutes an experience in which the formal and material constraints are balanced; with such a function, the DM can tweak the constraints using its DM actions.

Although the terminology has sometimes been used loosely, the evaluation function in DODM rates the quality of *interactive experiences*, not the quality of plot-point sequences considered as stories alone. That is, DODM does not create interactive drama by taking a

set of desiderata for *non-interactive* drama and trying to bring it about in the face of interactivity. Rather, it tries to maintain a set of desiderata for the interactive dramatic experience itself. Some DM systems do describe the drama-management problem as mediation between authorial narrative goals and player freedom [145, 59], and that view has sometimes been proposed as the general goal of drama management [94, 98]. It is not however the way DODM systems have typically viewed the problem. Rather than starting with an author-desired narrative and working around the user to bring it about, the system instead starts with an idea of what constitutes a narratively interesting experience, and dynamically adjusts the material and formal constraints in the story world in order to ensure that such an experience comes about—working with the player to jointly create the narrative [141, 58].

Weyhrauch's evaluation function specifies a number of weighted features that capture his notion of a good experience in his *Tea for Three* story world.

One group of features serves mainly to encourage narrative coherence—more formal constraints and, where necessary, material constraints, to keep the player on track. These features include *thought flow*, which prefers stories where subsequent actions relate to each other; *activity flow*, which prefers stories that have spatial locality of action; and *momentum*, which prefers certain pairs of plot points that build on each other particularly well. Separately, the *motivation* feature prefers stories in which plot points are motivated by previous ones, such as a plot point in a detective story motivated by earlier clues.

These are explicitly preferences for the interactive experience, and would not necessarily be the same if evaluating a linear story. It may not be bad for narratives to have the action move around frequently between different locations, but Weyhrauch argues that if each plot point happened in a different location from the last, that would likely indicate in an interactive experience that the player was getting stuck in boring wandering around the world between plot points.

Given only these features, there is a danger that the system could identify certain plot-point progressions as ideal and force the player into them, adding too many material constraints and reducing interactive agency. To avoid this outcome, two versions of an additional evaluation feature—one proposed by Weyhrauch and one by me in the preceding chapter—aim explicitly at encoding interactive agency, though from different perspectives.

Weyhrauch's *options* feature identifies twelve meaningful goals a player might have at various points in *Tea for Three*. The goal "talk to George about the new will" is considered to be active between the time the player finds a note mentioning a new will and the time that the player either talks to George about it or is prevented from doing so by other events. The number of goals active at any given time is a rough measure of the degree of interactive agency available. The *options* feature encodes a preference for many such meaningful options to be available towards the beginning of the game, decreasing to fewer towards the end.

My *choices* feature looks at the issue bottom-up instead, measuring how many plot points could have followed each point in the story, considering the ordering constraints in the world and the effects of causers and deniers. This is a rough measure of how much freedom the player had to influence the direction of the story locally. If at some point only one plot point could possibly have come next (because the DM caused it directly, for example), then the same bit of story would have played out regardless of what the player did. If, on the other hand, many plot points could have come next, the player could locally influence the story to a much greater extent. The *choices* feature has the advantage that it can be computed automatically without additional authored knowledge, but the *options* feature has the advantage that it captures a higher-level notion of meaningful interactive agency. Both features capture a notion of preferring stories that preserve player choice, demonstrating that this can be reprsented directly in the optimality criterion.

Finally, *manipulativity* penalizes uses of DM actions that are likely to be particularly noticeable, such as moving objects that the player can see. This is a meta-feature encoding

a preference for the DM's operation to be unnoticed. Although we use agents in service of a narrative rather than merely simulating them as believable agents in their own right, we do still want them to remain believable.

4.3 Targeting an experience distribution

Roberts *et al.* [99] criticize maximization of a story-quality function, arguing that if the DM is too effective, it will bring about the same highly-rated story each time, destroying interactive agency and replayability. They propose that the goal should be to target a distribution of experiences, specified either by some mapping from an evaluation function (*e.g.* bad experiences should never happen, and good ones should happen in proportion to their quality), or by having the author specify a few prototype experiences and then targeting a distribution over experiences similar but not identical to the prototypes [96].

While this is a valid criticism of maximizing *story* quality, the goal of DODM is to maximize *experience* quality. If the experience quality criterion appropriately includes features related to player agency, such as *options* and *choices*, then the DM will not force the same story every time.

More problematically, by targeting a specific distribution of experiences, TTD-MDPs do not necessarily coerce the player any less than a hypothetical system that targets a specific maximum-quality story would. A story-maximizing system that does not take into account features for player agency could indeed directly cause its same top-rated story every time. The TTD-MDP system, however, fares no better. Given powerful causers and deniers, it can achieve its preferred distribution over experiences exactly, by directly coercing each particular play-through, leaving nothing to chance. In either case, the player of any particular story would have no interactive agency, since in both cases the system would use its DM actions to produce a specific story. The TTD-MDP system would change *which* story it was forcing the user into each time, so multiple players (or one player replaying) would see a more diverse set of stories. But they would not be a set of stories in which the

player had much say: randomly selecting a different story to force the user into each time is not an improvement in interactive agency.

To verify this, I performed an experiment to look at the DM actions chosen by the TTD-MDP system and the DODM system on the version of *Anchorhead* with a the "synthetic" set of DM actions. And here there is indeed a similar level of coerciveness. The TTD-MDP system claimed better replayability in this case, since it produced a wider variety of stories. However, both the TTD-MDP system and the search-based evaluation-function-maximization system acted almost maximally coercively: they each performed an average of around 15 DM actions per experience, in an experience 16 plot-points long. The TTD-MDP system varied which specific coercion it performed from run to run, but that does not constitute interactive agency, which requires that the *player*, rather than the system's nondeterminism, be able to meaningfully influence the outcome.

That both systems are quite coercive does point to a failure in the experience-quality evaluation function that I proposed for *Anchorhead*, and which both systems used. We can correct this by putting a greater weight on the *choices* feature, emphasizing that giving the player choices in what to do really is an important part of an interactive experience. When I increase *choices* from being 15% of the total evaluation weight to 50%, both systems drop to using an average of around 5 DM actions per experience.

Thus maximizing an experience-quality function need not destroy an experience's interactive agency, if it appropriately rates highly only those experiences that do actually have good interactive agency. If an evaluation function *does* capture interactive agency well, then an evaluation "spike" in an evaluation-function graph like that in Figure 11 would not be a problem. The fact that a DM produced all high-quality experiences would not necessarily imply, as has previously been suggested [99], that it was linearizing the experience and always making the same literal experience come about, since many different experiences may have similar high ratings.

How to write evaluation functions so that they really do capture interactive experience

quality remains an issue that would benefit from more experimentation in specific real interactive dramas. The experimentation thus far (by myself and others) has focused on the "synthetic" model of *Anchorhead* that has only causers, deniers, and reenablers, and lacks the hint DM actions that a DM could use to add formal constraints by providing more narrative to the player without unduly removing interactive agency. By contrast, a real application would likely use hints frequently.

Whether the TTD-MDP formulation still improves matters in a different way depends on how the target distribution is defined, and on what we consider to be the goals of interactive drama. When the target distribution is generated by a mapping from an experience-quality function, the results will be fairly similar to the results from an evaluation-function-maximizing approach, since both systems will be trying to avoid low-rated experiences and increase the probability of highly-rated ones according to the same function. The TTD-MDP approach will add some more nondeterminism in doing so; how much depends on how the mapping is constructed. Alternate ways of specifying a target distribution of experiences for TTD-MDPs, however, such as specifying several prototype experiences and inducing a distribution over experiences similar to those prototypes [96], suffer from a greater loss of interactive agency. If the player is being forced into one of several prototype experiences or minor variants, the fact that the specific experience they're forced into is chosen nondeterministically does not preserve interactive agency.

In either case, the nondeterminism of TTD-MDPs serves a different goal than that of interactive agency. Interactive agency requires that if a player does things *differently*, then they should be able to experience meaningfully different narratives. It does not require a player to experience different narratives even if they do the exact same thing, which is the goal at which TTD-MDPs aim. Indeed, some ideas for interactive narrative, such as the "kaleidoscopic narratives" discussed by Murray [70], depend on a player being able to tease out how acting differently results in a different narrative, which nondeterminism is likely to obscure.

4.4 DODM conclusions

Declarative optimization-based drama management (DODM), a generalization of the search-based drama management proposed by Bates [4] and Weyhrauch [141], conceives of drama management in two parts. First, the story-relevant parts of an experience are formalized in a *story graph*, the drama manager's potential interventions are given as a set of *DM actions*, and the author supplies an *experience-quality* function that encodes his or her notions of what would constitute a good experience. The point is well taken that if this experience quality function slips into a *story-quality* function, then the drama manager will zealously attempt to optimize away the player's interactive agency in the name of its notion of a good story.

If, however, the experience-quality function explicitly takes into account elements of a good interactive experience, such as choices available to the player and a desire for less manipulation of the story world by the DM, then maximizing experience quality does not sit at odds with interactive agency. I argue that TTD-MDPs, although aimed at this problem, instead primarily serve to add nondeterminism to their actions, which does not in itself produce interactive agency.

It's worth nothing that adding nondeterminism to a DM's actions might be desirable in itself in some cases. For example, experience-management techniques can be used for experiences other than interactive drama, which may have different considerations, such as genuinely external constraints. For example, a TTD-MDP system was proposed for guiding museum tours [97]. In that domain, the goal of reducing congestion really is an external goal imposed on the visitors, and is reasonably expressed by targeting a specific distribution of experiences so as to keep visitors nicely spread out. Another possible example is that training scenarios may also have an externally imposed requirement that a particular distribution of desired situations be encountered over a series of training runs.

Finally, the investigation here focuses on using this story-graph formalism for drama management, *i.e.* runtime intervention in an unfolding story. As argued in Chapter 1, an

advantage of declarative knowledge representations for games is that they may be applied in many settings, from design tools to runtime decision-making. An interesting avenue to explore, which I have been collaborating with colleagues on looking into, is using the story-graph and experience-quality formalizations from DODM at design time. In one experiment, we still have the goal of drama managment, but "compiled" ahead of time: the results of what the drama manager would do in various situations are compiled to equivalent test-and-branch logic, which is a more traditional, "manual" way of writing interactive stories, but here generated automatically [12]. In another experiment, the story graph is analyzed at design time, suggesting places where the author might add new plot points to produce a better experience [33].

CHAPTER V

GENERATING VIDEOGAME THEMES[1]

One way of stress-testing knowledge representation for videogame design is to try to use it to automatically generate videogames. The problem is not precisely the same as that of design support: a representation quite useful for support may not be complete enough to automatically generate games on its own. But it is still a useful experiment to attempt automatic generation, in order to uncover difficult aspects and corner cases of design-knowledge formalization. It is also a problem with its own independent appeal, much like other creative-generation challenge areas (automated story generation, music generation, etc.).

Generating videogames is a broad goal: there are many kinds of games, and many aspects of designs we might hold fixed while varying others. To find a starting point to make headway on the problem, recall the factoring of potentially formalizable game-design knowledge I gave in Section 2.2. The four areas are: abstract mechanics (game state and state-transition rules), concrete audiovisual representations and their associated mechanics (*e.g.*, 2d graphical movement and collision), thematic mappings (*e.g.*, game entities and actions interpreted as bars, rivers, cities, chases, etc.), and input mappings (how and what the player controls).

Research on automated game design could conceivably focus on just one of these factored areas. The most obvious might be abstract mechanics, since inventing new games is often seen as synonymous with inventing new "game rules" (though as I've argued, in videogames it is far more complex than that). And indeed there has been work on, for example, generating variants within classes of board games or arcade-style games [10, 40, 131].

[1] Parts of this chapter have been previously published as [74, 76].

Since our goal is to use game-generation research to push on knowledge representation, however, it makes sense to look at the interaction of at least two areas, to see how the problem looks at the seams. Furthermore, these abstract game-generation experiments ignore an almost ubiquitous, important, and challenging to formalize feature of videogames: the thematic mappings. This has connections to commonsense reasoning, another longstanding KRR problem in AI, and forms the starting point for this chapter's investigation—a somewhat more speculative project than the others.[2]

The question I investigate here is: What makes a game be "about" something, and can we formalize that sufficiently to generate games on demand that are about specific topics? On a very superficial level, we might say any game depicting a subject is in some sense about the subject: any game with cities in it is about cities, and any game with a bartender is about bars. But that hardly gets at the aboutness of *games* specifically. What makes games about something are the thematic *mappings*, which connect real-world references to gameplay. Thematic mappings turn an element of abstract game state, say a decrementing integer, into something the player interprets as a "falling ball". Or they turn two moving sprites, in the right context, into a "chase".

The simplest version of thematic mapping manipulation can be thought of as *reskinning* a game. Reskinning simply replaces the artwork in a game, making it "about" something else, but otherwise leaving the game's mechanics and input mappings unchanged. Therefore it is precisely about the interface between thematic mappings and the rest of the game, asking how one pairs the two. Reskinning is also an established concept, and a popular activity among videogame players, in part because it can often be done without access to the original game's source code. By identifying and replacing the game's art assets or textures, players can make unauthorized adaptations and customizations [106, 142]. From an

[2]That isn't to deny that game generation focused on abstract mechanics is fascinating in its own right. While this chapter focuses on thematic mappings, I have co-written a review of game-rule-oriented generative research elsewhere [83]. I've also contributed both proposed criteria for judging arcade-game rules [84], and some speculation on the future prospects of that line of work [130]. But that is another discussion.

applications perspective, reskinning of simple games opens up a number of design-support opportunities, in particular (and what I investigate here) to help authors think at a more abstract level about *generative spaces* of games, rather than designing individual games one at a time.

5.1 Reskinning as design support

As games have become a hot cultural phenomenon, an increasing number of game-design novices wish to express desired content or accomplish a goal using game-like expression. During the time I was a researcher at the UCSC Expressive Intelligence Studio, that one group alone received several cold calls a month from advocacy and education groups who wished to build small serious games to teach specific material or express a point of view, typically with little to no budget. What prevents these groups from being able to build their games is not access to game programming tools; there are now numerous game toolkits such as *Game Maker*,[3] as well as web technologies such as Flash, that greatly ease the programming burden for creating simple games.

Rather, the main roadblock is the game design process itself: the process of presenting desired content, such as bike safety, the policy implications of a proposed law, or high school algebra, in a way that makes use of the unique qualities of the game medium. These unique qualities include creating a strong sense of player agency, procedurally expressing the possible-worlds implications of player decisions, and enabling players to experientially explore rule systems.[4] Existing game toolkits such as *Game Maker* or *Alice*[5] provide no help with the design problem of mapping content into a system of game rules, but rather facilitate the programming task of implementing the game rules once design is already done. And indeed, we find that many games that deal with sophisticated real-world referents, such as political games and persuasive games, are produced by small studios studios run

[3] `http://www.yoyogames.com/gamemaker/`

[4] Mateas [62] and Bogost [8] discuss in more detail the particular expressive affordances of procedural systems, and how they differ from those of other media.

[5] `http://www.alice.org/`

by academics who are experts in expressive game design.[6] Therefore a goal of formalizing the application of thematic mappings is to facilitate novice game designers in the initial, creative design phase, by helping them brainstorm what it would mean to make a simple game "about" their particular concern.

5.2 Generative WarioWare *games*

Nintendo's *WarioWare* series consists of a series of small games, typically lasting a few seconds, that involve one simple element of gameplay such as "dodge the car". This style of game allows us to focus almost exclusively on the thematic mappings: the gameplay is simple enough that the mechanics can be formalized relatively easily, as a set of templated simple games. Nonetheless the space of possibilities is rich enough for the series to be interesting to game designers [34].

To allow *WarioWare*-style games to be reskinned, I break them down into three main components: *game spaces*, which define a space of possible games, such as a game where one entity tries to avoid another entity; *stock mechanics*, which are literal game code with templated slots; and *mapping rules*, which specify how a game space should map onto a stock mechanic to render a specific game within the space playable.

The game spaces are primarily thematic, but also include some assumptions about how the abstract mechanics would operate. For example, a game with one entity avoiding another has two entities (the avoider and the attacker), which must be chosen so that it "makes sense", in terms of common-sense knowledge, that one would be avoiding the other. This thematic content comes with an assumption that the game itself, however implemented, will involve a mechanic where the avoider tries to get away, and wins if successful, or loses if caught.

[6] Some well-known examples are *Madrid* (`http://www.newsgaming.com/games/madrid/`), a memorial game released shortly after the 2004 Madrid train bombings, from a team in which one author wrote his PhD thesis on the subject [29]; and *Food Import Folly* (`http://select.nytimes.com/2007/05/24/opinion/20070524_FOLLIES_GRAPHIC.html`), an editorial game for the *New York Times* online opinion pages about contaminated food imports, from a studio whose cofounder wrote a book on the subject [8].

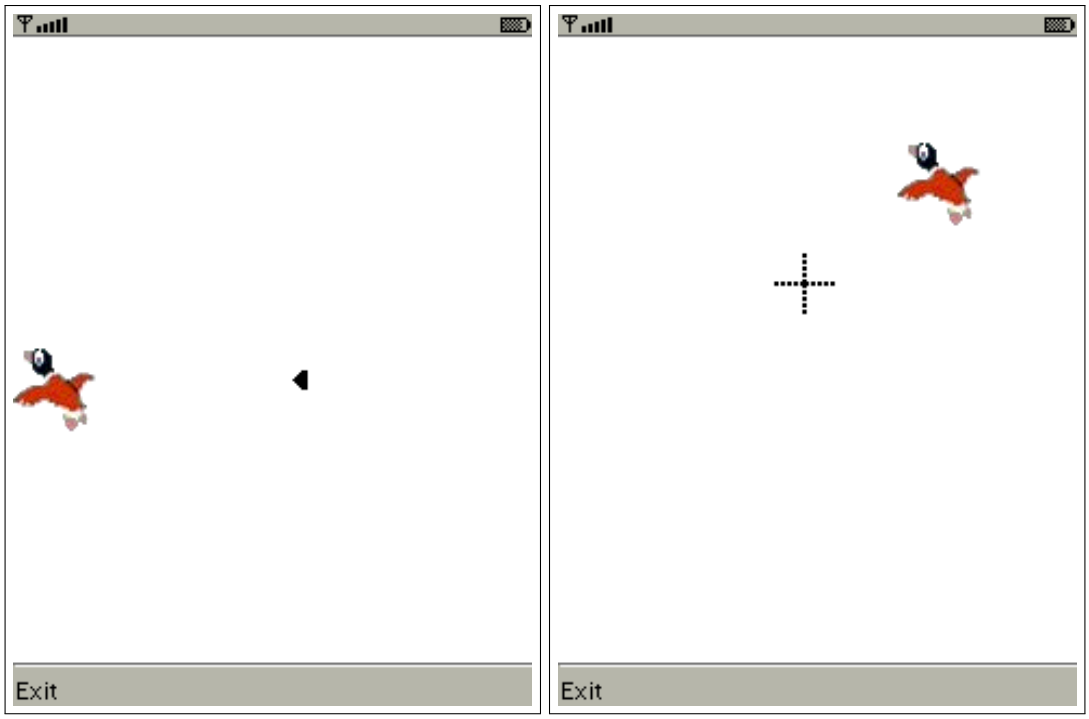


Figure 14: Two mappings of a game from the attacker-avoidance game space in which a duck avoids a bullet.

The stock mechanics are bundles of both abstract and concrete game mechanics, in this case implemented as code for the Java mobile platform (J2ME). They specify a fully playable game, but with templated slots for some entities in the game.[7] For example, one of the *WarioWare*-style mechanics in this system (called *Dodger*) has one object moving across the screen, while another object that the player controls moves up or down quickly to get out of its way, as shown in Figure 14 (left).

Finally, the mapping rules ensure that the mechanic assumptions that are implicitly part of a game space are respected, by mapping the game space onto stock mechanics in such a way that the common-sense thematic constraints are preserved. For example, Figure 14 shows two different ways in which a game about a duck avoiding a bullet can be mapped onto two different stock mechanics. In the game shown on the left, the avoider (a duck)

[7]Thanks to Nuri Amanatullah, Thib Guicherd-Callin, Jeremy Hay, and Ian Paris-Salb (UC Santa Cruz), whose modular *WarioWare*-like game engine for J2ME was used to implement these templated games.

is mapped onto an avatar that the player moves up and down, while the attacker (a bullet) is mapped onto a computer-controlled sprite that moves left across the screen towards the player. In the game shown on the right, the the avoider (a duck again) is mapped onto a computer-controlled sprite that moves randomly, and the player controls crosshairs to play the role of the attacker (a bullet again), which isn't explicitly shown. These examples of mappings are straightforward one-to-one mappings, but more complex mappings can also be constructed, where for example animate and inanimate objects map differently, and other parts of the game are changed based on properties of the specific terms being filled into the slots.

In this work, I experiment with three game spaces, and five kinds of stock mechanics.

Game spaces: An *Avoid* game is one in which one entity, the "avoider", must avoid (for the duration of the game) one or more other entities, the "attackers", which may attack the avoider either directly or via other objects. The player can play either role. An *Acquire* game is one in which the player must find an object within a time limit. A *Fill* game is one in which the player must fill a meter within a time limit. These abstract game types capture the abstract mechanics the system currently knows about.

Stock mechanics: A *Dodger* game is a 2d top-down game in which one object, the "dodger", tries to avoid one or more other entities, the "attackers"; it is used to implement some Avoid games and some Acquire games. A *Shooter* game is a 2d side-view game in which objects move across the screen, and can be shot by aiming crosshairs and firing; it is used to implement some Avoid games. A *Pick-Up* game is a 2d top-down game with a player and an object for them to pick up, usually through some obstacle such as a maze; it is used to implement some Acquire games. A *Pump* game has a reservoir of some sort that needs to be filled up; it is used to implement some Fill games. A *Move* game has a player moving some distance; it is used to implement some Fill games.

Each of the five types of game mechanics can then be matched with composable movement managers that determine how the non-player-controlled objects will move, based on

```
noun Avoider
noun Attacker
verb Attack_verb: shoot, attack, damage, chase, injure, hit
constraint: (ConceptNet CapableOfReceivingAction ?Avoider ?Attack_verb)
constraint: (WordNet hyponym ?Avoider "animate thing")
constraint: (or (and (WordNet hyponym ?Attacker "projectile")
                     (ConceptNet CapableOfReceivingAction ?Attacker ?Attack_verb))
                (ConceptNet CapableOf ?Attacker ?Attack_verb))
```

Figure 15: A definition of an example game space, specifying games where an Avoider avoids an Attacker. See text for explanation, and Figure 16 for the graphical view of this game space.

some common-sense reasoning about the thematic representation they've been assigned (see next section). For example, attackers in a Dodger game where the player plays the dodging side might chase the player (if animate) or travel in a straight line (if not). The interface mappings are currently bundled with the objects in the concrete game mechanics: If the player plays the dodging side in a Dodger game, then the controls will be arrow-keys to move. These concrete game mechanics capture the concrete representations and control mappings the system currently knows about. Finally, there is a stock set of sprites, each attached to a noun describing them, that can be used as the graphical representation of any of the objects in any of the games.

5.3 Defining game spaces via constraints

Games spaces are defined by specifying variables, which are marked as nouns or verbs, and constraints on the variables. The constraints are either on the values that individual variables can take, or on how the values of multiple variables must relate to each other. When defining a game space, the author may specify the constraints using relations in the ConceptNet [56] and WordNet [25] databases, as well as some combinations of the two and logical operators like *and* and *or*. Figure 15 shows the set of variables and constraints specifying an "avoid" game, where one noun avoids another noun, which we'll use as an example. Note that a user of the interactive tool need never see this raw game description;

see Figure 16 for the graphical view.

5.3.1 ConceptNet and WordNet

ConceptNet is a graph-structured common-sense knowledge base extracted from Open-Mind [112], a collection of semi-structured English sentences expressing common-sense facts gathered from online volunteers. ConceptNet's nodes are English words or phrases, and its links express semantic relationships such as *(CapableOf "person" "play video game")*. In our current prototype, *CapableOf*, *CapableOfReceivingAction*, *PropertyOf*, and *UsedFor* are the most useful relations.

WordNet is a hierarchical dictionary of English words. A word below another one in the hierarchy is a specialization of the higher-up one (the higher word is a "hypernym" and the lower one is a "hyponym" if a noun or "troponym" if a verb). This can be used to constrain variables in a game space to specific types of words; for example, constraining a noun to be a hyponym of "animate object" makes sure that inanimate objects aren't put into the slot of a stock mechanic where they wouldn't make sense. More generally, WordNet allows us to perform taxonomic generalizations over relations defined in ConceptNet.

Compared to more formally specified common-sense knowledge bases such as Cyc [54] and ThoughtTreasure [67], these databases use natural language and relatively loose semantics. This is nice because it allows an author to interact with the databases without having to learn how to navigate a particular formal ontological framework in order to define their game spaces. In addition, ConceptNet's natural-language approach to common-sense knowledge has been useful for a number of previous applications [55]. However, having the knowledge base in natural language does have some drawbacks when it comes to ambiguity and inability to usefully respond to complex queries, and indeed, helping an author navigate these difficulties is one of the main goals of our interactive approach to defining common-sense constraints.

ConceptNet has the more serious drawback of weak coverage: it knows that a duck

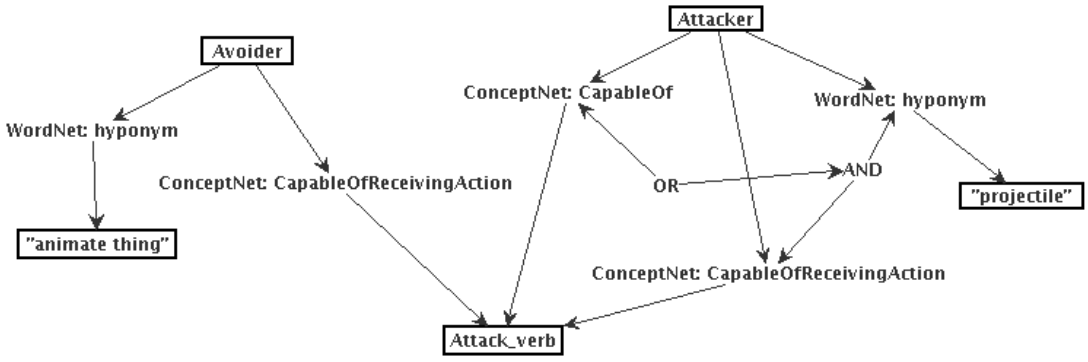


Figure 16: The graphical view of the attacker-avoidance game space specified in Figure 15.

can be shot, but not that a pheasant can be shot, for example. Fortunately, combining queries to ConceptNet with hierarchical information from WordNet mitigates this problem to a large extent. In specifying a particular ConceptNet constraint, the author can specify whether WordNet "inheritance" should be performed on any of the terms or variables in either direction (towards more general or towards more specific words). For example, the query about whether a pheasant can be shot should have hypernym (towards more general terms) inheritance enabled on "pheasant", and would therefore return true, because from WordNet we find that a pheasant is a type of animal, and ConceptNet knows that animals can be shot.

5.3.2 *Avoid* game definition

Figure 15 shows an example game space specifying games where an *Avoider* tries to avoid an *Attacker*. The game space is defined by three variables—those two nouns plus a verb (*Attack_verb*)—and several constraints between them.

The nouns can implicitly range over any noun for which the system has graphics, subject to the constraints. The verb is specified to be one of five verbs that the author has chosen as representative of the type of action to take place in the game. The constraints specify how to bind these variables to specific terms from ConceptNet so as to maintain the common-sense semantics of "attack" and "avoid".

The first constraint specifies that the Avoider has to be capable of serving as the direct object of the Attack_verb (represented by the *CapableOfReceivingAction* relation in ConceptNet); hypernym inheritance is done on the Avoider (not shown in the figure for simplicity). The second constrains the Avoider to being an "animate thing", since it must move to avoid the Attacker. Some inanimate things could sensibly function as Avoiders, especially humorously (a piece of bread trying to avoid a toaster, say), but specifying which of them makes sense gets trickier, so in this example the game space considers only animate things as Avoiders.

The third constraint is somewhat more complex. The most obvious constraint to add is that the Attacker must be CapableOf the Attack_verb (with hypernym inheritance on the Attacker). After trying this, however, it turns out to preclude many games that we think of as canonical in the avoider game design space. Something trying to avoid a bullet, for example, is excluded because a bullet isn't itself CapableOf "shoot": A bullet doesn't shoot, but is shot, and therefore isn't *CapableOf*, but rather *CapableOfReceivingAction* shoot. We informally except it to serve in the role of attacker and perhaps anthropomorphize it as "chasing" the Avoider, but those notions are too subtle for the common-sense databases that currently exist to capture. To take these cases into account, there is an alternate possibility for fulfilling the last constraint: If an Attacker is a projectile (according to WordNet), then we check whether the Attack_verb can sensibly act on *it*, rather than whether it can sensibly act out the Attack_verb.

Specific games in this game space are then mapped by author-specified rules to stock mechanics. The example earlier in Figure 14 was the result of two such mappings, of an *Avoid* game in which a duck avoids a bullet. In one case, the player plays the duck and avoids an incoming bullet in a side-scrolling mechanic; in the other, the player plays the implicit shooter of a bullet, and aims at the duck using a cross-hair targeting mechanic.

5.4 Interactively defining game spaces

The formalization thus far allows for automatically reskinning games within the constraints of an author-specified game space. Unfortunately, this is rather difficult to use. Given the vast size yet lack of completeness of common-sense knowledge bases, a game space defined by a given set of constraints will often produce counterintuitive results, including concrete games that the author doesn't expect, while excluding games the author might think are canonical in the game space. Thus an intelligent game authoring tool must support the author in dynamically exploring a game space, allowing him or her to ask the system why a given generated game is allowed by the current constraints and to iteratively modify constraints. In fact, it is this iterative exploration that is the heart of the leverage provided by an intelligent game design tool; the combination of human plus system can explore a much larger design space than an unaided human could, while the human provides heuristic search guidance that would be extremely difficult to encode in a fully automated system.

To interactively define game spaces, the interface centers around a graph-structured view of the variables and constraints defining a game space, with a number of tools for querying and modifying it. Figure 16 shows the constraint graph for the example attacker-avoidance game whose raw textual specification was shown in Figure 15. Variables and constraints can be added or removed interactively (the constraint-satisfaction backend re-runs in the background after modifications are made). Some filtering is done in the interface so that only relevant additions are shown; for example, an author selecting two nouns and clicking to add a constraint will only get a list of constraint types relevant for pairs of nouns, which excludes those such as ConceptNet's *CapableOf* that involve verbs.

An author can query a particular variable to get a list of terms that could be bound to that variable when generating a real game, under some binding of other nouns and verbs in the game space. More helpfully, he or she can click on a particular word to get an explanation of *why* it satisfied the constraints, allowing debugging of unexpected results.

Figure 17 shows an example from debugging an earlier iteration of the attacker-avoidance

Figure 17: Possible assignments to a particular variable, letting the author look for unexpected results and query why they satisfied the constraints.

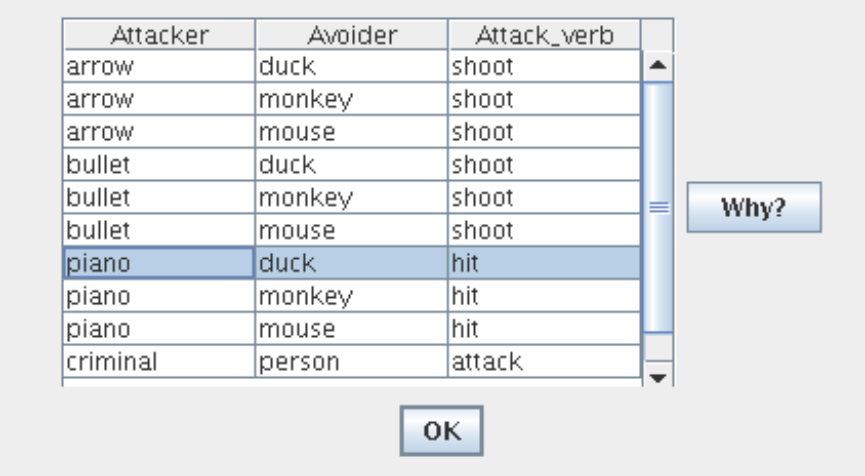


Figure 18: Possible assignments to all variables, showing a more contextual view of assignments than the single-variable view in Figure 17. The author can query particular assignments here as well.

game space. Before "projectile" was settled on in the third constraint (see Figure 15 and previous discussion), I had tried the more general term "device". The idea was that it would be nice if objects other than traditional projectiles could be used as projectiles anyway, such as a hammer being thrown at the avoider. Upon adding that option, though, one of the generated games involved a piano flying across the room. Without some sort of context that wasn't present in this simple game space (say, a gorilla shown as throwing the piano at the player), this didn't make much sense.

The view in Figure 17 quickly lets the author click "Why?" to find out why. The answer: a piano is a both a device (according to WordNet) and *CapableOfReceivingAction* "hit" (according to ConceptNet), so fits all the constraints. Presumably a piano is a device for making music, and its keys can be hit, which isn't really what we had intended those constraints to mean. One solution was to limit "device" to a more specific kind of device, "projectile". In an earlier, fully automated version of the system, I had to manually dig through debug output and trace through the ConceptNet and WordNet graphs in order to figure out why such unexpected results would appear. This painful manual process was one of the motivations for moving towards an interactive game design assistant supporting visual exploration of game spaces.

To get a bigger-picture view of how all the variables and constraints in a game space interact, the author can get a list of some possible complete games; Figure 18 shows an example. The author can click on any particular term in this list to get an explanation as well. To provide a visual overview, we can also show the author a set of graphical screenshots of some of the concrete games in the game space, based on the separately defined game space to stock mechanic mappings, on which he or she can click to get textual description of the values filled in for the variables.

The overall purpose of the interface is to make it possible for an author to efficiently use specific examples of games that do or don't fit his or her goals for the game space (or make sense at all, for that matter). This is intended to help the author reason about general rules

specifying what games should be included in and excluded from the space. This can be used in two different modes, offline and online. In offline mode, the author can select concretely instantiated games from the examples generated by the system. In this case, the authoring system is serving as a "brainstorming assistant", providing help in performing thematic mapping onto game mechanics, and helping the author to explore a larger design space than he or she might have been able to unaided. In online mode, the author deploys a generative design space, in which games are generated at runtime according to the constraints. In this case, the authoring system is helping the author to visualize game design spaces, and to refine constraints until the game space(s) reliably produce the author-desired range of games without further manual selection of specific games. This requires a higher level of specificity in the game spaces, however, since the author is not around at runtime to throw out the occasional poor match.

People often find it more intuitive to define categories by specifying prototypes rather than logical rules [100]. In recognition of that fact, some systems try to infer constraints from examples demonstrated by users, rather than requiring the users to specify explicit constraints; for example the graphical editor GRACE infers constraints on the relationships between elements of the design (*e.g.* in a CAD diagram) by user demonstration and some inference heuristics [2]. Those approaches succeed in domains where the constraints are simpler and more concrete, however. In the game-design domain, the state of common-sense reasoning isn't sufficiently advanced for us to reliably infer something like "a game about avoiding an attacker (where the attacker and avoider and their relationships 'make sense')" from some examples of games that should and shouldn't be in that space.

Rather than inevitably getting inferences from prototypes wrong, this system tries to help the author understand the constraints they're in the process of specifying—and in particular to understand what the system thinks the constraints they're specifying mean—by frequent reference to prototypical examples. The goal is to make failures of common-sense reasoning explicitly visible so the author can recognize and work around them, helping the

author to define an explicit set of rules that can be applied concretely and that are congruent with his or her goals. The hope is that this incorporates some of the intuitive advantages of a prototype approach without relying on inferences that, in the commonsense domain of thematic mapping onto game mechanics, would be unreliable and result in confusion.

5.5 Automatic theming conclusions

Although thematic mappings are not the aspect of game-design knowledge most easily formalized, it is possible to provide design support by limiting the scope of games and themes considered. Here, I present the idea of constraint-based game spaces, which provide support for authors to generalize specific games to a space of games that are in some sense analogous but for the reskinning. The example analyzed in detail here is generalizing the idea of a duck-hunt-style game to a reskinnable game space where the duck and hunter can be automatically replaced by another pair of *attacker* and *avoider*, respecting a set of constraints written in the language of the ConceptNet and WordNet commonsense-knowledge databases.

The formalization here is then applied in three different ways. First, it is used interactively during the development of new game-space formalizations themselves: authors can query the in-progress game space to find example matches, and to ask why a specific match was generated. Secondly, it can be used as a brainstorming tool, simply generating a series of games from the formalized game space, which the author can play in series and then select from. And finally, the auto-skinnable game space can itself be shipped, producing a generative *WarioWare*-like game that generates a series of games from the author's defined space.

Since the work here has deliberately aimed to focus solely on thematic mappings, the stock mechanics are entirely hard-coded, and the mapping process is quite simplistic. Spaces of thematic mappings are represented by flexibly editable constraints, but are then lowered onto mechanics quite directly. In addition, each stock mechanic is extremely

simple, and is assumed to have only one "reading": the *Dodger* game has two entities, in which one is dodging the other. Relaxing these restrictions is an area with ample scope for future work (see Chapter 9).

CHAPTER VI

MECHANICS-ORIENTED PROTOTYPE SUPPORT[1]

One way designers (in any field) negotiate the relationship between the materials of their craft, and the properties or effects they hope their designs will have, is by an iterative process of building and experimenting with prototypes; Donald Schön describes this as a conversation with the materials of a design situation [108]. When designing a game's abstract mechanics, a designer's materials are abstract computational processes and rule systems, and the goal of early-stage prototyping is to understand how their game operates, and how this compares to their goals for the game.

What do we mean by *understand how their game operates*? Returning to the mechanics–dynamics–aesthetics (MDA) view of game design [41] discussed in Chapter 2, the connection between the game as a technical artifact, and gameplay as a subjective, aesthetic experience, is via the dynamics of gameplay: the patterns of interaction and traversal that take place when a player plays the game. In building mechanics-oriented prototypes, the designer hopes to understand what sorts of dynamics can arise from their mechanics: which sequences of events are possible and impossible, which states are reachable via which routes, which gameplay elements depend on or are mutually exclusive with each other, which events always or never happen, which outcomes would be reachable by a player using one strategy versus another, and so on.

Understanding how a game operates as an interactive artifact doesn't directly answer experiential questions, such as what players are likely to do, or whether they'll find the game fun. But the designer who doesn't at least understand how their game *works* will be designing gameplay in the dark, since their game's dynamics constitute the raw experiential

[1] Parts of this chapter have been previously published as [72].

trajectories out of which the overall aesthetic experience emerges. Put differently, designers are trying to modify gameplay when they modify mechanics. The only reason to modify a mechanic is to modify what kind of gameplay arises, since mechanics themselves are not an end goal; and therefore the designer who wants to operate by something better than trial and error needs some way of understanding this relationship between their control space (mechanics sculpting) and the desired design space (gameplay-dynamics sculpting): what happens when they make a change, and why?

A common (and sensible) way for a game designer to improve his or her understanding of a design-in-progress is to playtest a prototype. Doing so gives the designer empirical information about what players do in the game (and when and how they do it), as well as about their subjective reactions. Game metrics are an approach to understanding games and gameplay by analyzing and visualizing information collected from players in playtests. There has been considerable recent work in using visualization and AI tools to improve the process of collecting and understanding this empirical information. The most well-known visualization is probably the "heatmap", a map of a game level color-coded by how frequently some event occurs in each part of the map, allowing a quick visual representation of, *e.g.*, where players frequently die [129]. In addition to raw event occurrence, more detailed information can be visualized, such as the reasons for events or timing of events (for example, a map can be color-coded by average time it takes players to reach each point). More complex analysis can be extended into analysis of gameplay sequences and patterns [23], characterization of play styles [24], and analysis of player experience [89], among other things.

In all these approaches, the source of information is *exclusively* the player. Empirical information is collected from players, by methods such as logging their playthroughs, tracking their physiological responses during play, or administering a post-play survey. Then this data is analyzed and visualized in order to understand the game and the gameplay it produces, with a view towards revising the design.

For some kinds of game-design questions, it's sensible or even necessary for our source of information to be empirical data from players. If we want to know if a target audience finds a game fun, or what proportion of players notice a hidden room, we need to have them play the game and find out. But an additional purpose of playtesting is for the designer to better understand their *game artifact*, in the sense of a system of code and rules that implements the game. This artifact works in certain ways, and the designer has some ideas about how he or she thinks it works.

Some common results of playtesting aren't really empirical facts at all. When the designer looks over a player's shoulder and remarks, "oops, you weren't supposed to be able to just walk there without having to talk to anyone", that isn't an empirical fact about players or gameplay that's being discovered, but a logical fact about how the game works: there is a way to get to a location without taking actions that the designer thought were necessary prerequisites for getting there. That can cause problems such as a level being too easy and boring, or the player not having seen backstory that the designer assumed they would have by a certain point; but the main point is that it was a possibility the designer assumed was a non-possibility, and the important information for the designer to learn is that it is in fact a possibility, and one they should know about.

6.1 Analytical metrics

When the designer finds out something new about their game artifact, such as a possible route through a level they had assumed was impossible, they update their mental model of how the artifact works, and continue with the design under the new understanding. This is an example of Schön's reflective conversation with the materials of a design situation at work: the artifact has in effect produced backtalk saying "here's a thing I can do after all!". Of course the artifact doesn't speak, so where does this backtalk come from? In player metrics research it is typically from empirical observation of the player. But this particular kind of backtalk produces an *analytical fact* about the artifact—not an inherently empirical

observation about player experience.

This kind of backtalk, leading to improved understanding of a game artifact, *can* be discovered through playtesting. But the only real role of the player in uncovering that kind of information is to put the game through its paces, so the designer can observe it in action. If we metaphorically think of the game artifact as a series of interconnected gears, the player is turning the crank to operate the gears so we can notice that they sometimes slip. But can't we determine when gears can slip without recruiting players? Could we turn the crank ourselves?

Figuring out which unwritten possibilities are implied by a game's abstract mechanics is particularly well suited to automated reasoning, since they are quite literally what can be formally derived from the mechanics. Designers already often build simple prototypes, with everything except the bare rule system stripped out (the visual representations employed by such prototypes are often abstract geometric figures, such as circles and triangles), in order to figure out how the rule system operates. Allowing automated derivation of the logical implications of a set of game mechanics can speed up this process considerably. By making the implications of a set of game mechanics immediately available, the assistant increases the "backtalk" of the design situation and allows for quicker design iterations.

Indeed, designers do manual versions of this kind of analytical study of their game. When they design rule systems and write code, they have mental models of how the game they're designing should work, and spend considerable time mentally tracing through possibilities, carefully working out how rules will interact, and perhaps even building Excel spreadsheets before the game ever sees a playtester. Can we use AI and visualization techniques to augment that thinking-about-the-game-artifact job of the designer, the way we've augmented thinking about player experience through empirical metrics research?

I propose the lens of *analytical metrics* as a parallel to traditional empirical gameplay metrics. One might also think of them as *design-time metrics*, in the sense that they can be available to the designer *while* designing a game, perhaps even integrated into design

and editing tools (the way CAD software integrates analytical capabilities).[2] In addition, it's important to note that analytical metrics need not be used strictly *instead* of empirical metrics. While they can provide some kinds of information earlier or more easily than playtesting, they can also work naturally alongside existing metrics and visualization approaches. Collecting both analytical and empirical metrics relevant to a given design question can add more information than either can do alone, showing how actual player behavior relates to the game's possibility space.

6.2 Gameplay possibility spaces

Empirical gameplay metrics are conceptually based on aggregating play traces: players play through a game, information about the events that take place and the game state at various times is logged, and the results are aggregated and visualized. A complete gameplay trace is simply a log of all state and events that took place during a playthrough, in our case logged at the level of the game's abstract formal model. The set of every possible gameplay trace is therefore a game's *possibility space*, and it is this space that automated analysis aims to elucidate.

6.2.1 Games as bags of traces

We can think of a game as being defined extensionally, as a bag of all possible gameplay traces. A bit of notation to formalize this view: if G is a game, then $\langle G \rangle$ is its extension. That is, $\langle G \rangle$ is the set such that $T \in \langle G \rangle$ if and only if $T = (T_0, T_1, \ldots)$ is a game trace that could possibly be produced when game G is played. Here we use T_n as shorthand for everything that is the case (state and event occurrences) at timestep n in a particular playthrough.

The simplest form of traditional playtesting is the kind of quality-assurance (QA) playtesting that has an army of testers playing a game repeatedly, looking for bugs. Some bugs are

[2]The term "design-time metrics" was suggested by Adam M. Smith (personal communication).

implementation-related, such as the game crashing, which are excluded from our scope. *Design* bugs are noticed when someone in the playtesting army makes something happen that shouldn't have. For example, perhaps they find a way to bypass a supposedly mandatory part of a level, or find a way to get themselves stuck in a room with no way to get back out of it. What the playtester has found in these cases is either a game trace $T \in \langle G \rangle$ that shouldn't have been there, or the absence of one that should have been there. Such a finding implies that the game possibility space $\langle G \rangle$ is not made up of exactly the play possibilities we wanted it to be comprised of. Therefore something in turn must be not quite right about the game G. Our conception of what $\langle G \rangle$ should contain may not have included, for example, the existence of any traces where the player ends in a room with no way to escape. Therefore the discovery that it *does* contain such traces as possibilities means that the mechanics comprising G need to be changed so that the game no longer allows those undesired possibilities.

In short, this variety of bug-finding playtesting consists of negotiating the relationship between the game G and the game-trace possibility space $\langle G \rangle$ that it implies. The designer has an idea of what $\langle G \rangle$ they are trying to produce, but what they actually produce directly is G. Playtesters probe that relationship by sampling individual traces from $\langle G \rangle$ until an anomaly is found.

In the case of formalized game mechanics, we can query this relationship directly, because we have an *intensional*, machine-readable representation of G, which can be used to prove things about, or extract samples from, $\langle G \rangle$. Thus, in the case of mechanics-oriented prototypes, we now have reduced the formal gap between the author's tools of the trade (mechanics) and eventual goal (gameplay) to the gap between an intensional and extensional definition of a game, something automated inference tools are well equipped to negotiate. It also highlights some of the source of design difficulty: the game's mechanics specify an intensional construction of a gameplay space, but it's often easier to think of desired properties extensionally, in terms of things that should or shouldn't happen, or should

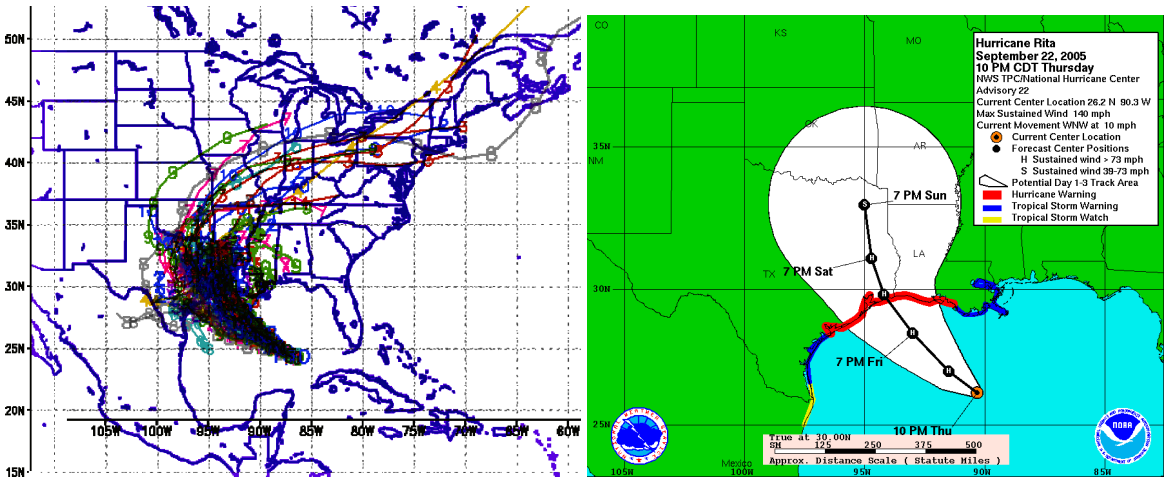


Figure 19: Ensemble forecast model (left) and estimated hurricane cone (right) for Hurricane Rita, 2005. The left visualization is simulating G to sample traces from $\langle G \rangle$, and the right visualization estimates spatial bounds on $\langle G \rangle$. Image credits: National Oceanic and Atmospheric Administration (NOAA), Hurricane Research Division (left) and National Hurricane Center (right).

happen in certain ways. This is a key difficulty in system-crafting in general, and close to the idea of what "emergence" means. Section 6.4 will propose a more specific set of strategies for retrieve and visualize useful design information about $\langle G \rangle$, given the formalized mechanics model G.

Thus far we could apply this definition equally to games and non-interactive simulations, since a simulation can also be defined extensionally as a bag of possible traces through the simulation. Consider the case of hurricane-forecast models. In the case of a deterministic simulation, $\langle G \rangle$ will consist of one trace for each set of possible initial conditions and choice of parameters. For example, Figure 19 (left) shows an ensemble model commonly used in hurricane track forecasting, with each path tracing out a possible hurricane track, according to a choice of initial conditions and model parameters. The visualization is produced by plotting one particular aspect of the trace, the spatial position in latitude and longitude, on a map (a trace will have many other bits of state, such as estimated wind speed and pressure, that are not shown in this visualization). Commonly we want to visualize the contours and limits of $\langle G \rangle$ rather than the samples directly.

In the case of our formalized models of game mechanics, these can be computed directly through logical theorem-proving. In the case of a complex simulation like a hurricane track model, they are derived by repeated sampling from the model, and aggregating the results. Figure 19 (right) shows a typical hurricane prediction cone, essentially a visualization of spatial bounds on the paths in $\langle G \rangle$, estimated from the aggregated simulation runs.

6.3 Game/player possibility spaces

A key aspect of games, of course, is that these game traces do not arise fully autonomously, but are produced when players play a game. The game trace is produced by the combined efforts of a player, who gives a game inputs, and processes internal to the game's logic, which execute code. We can think of a game in a sense, then, as a machine with an operator. Players (the operators) drive it through sequences of states and events. A game trace is therefore the log of one session of the machine's operation by a particular player. Call the sequence of player inputs during a particular session, $I = (I_0, I_1, \ldots)$, a *player input trace*. Then a game G is a transducer from player input traces to game traces: it takes in a sequence of player inputs (the player input trace), and as output logs a sequence of game state/events (the game trace).

A game's bag-of-traces extension $\langle G \rangle$ is the set of game traces that can be produced by *any* operator. Let P be an extensionally defined player model, *i.e.* a model of operator behavior. This model is similarly defined as a bag of traces, in this case a bag of player input traces $\langle P \rangle$. Note that this extensional definition is quite general, and leaves us agnostic about the status of the player in flesh and blood. A bag of input traces might be a single player's actual range of play; it may represent the aggregate play styles of a population of players; it may represent only a stylized hypothetical kind of play; or it may represent as little as a single input trace. For formal purposes it is simply a subset of all possible player input traces, used as an analytical tool to examine the operation of G under different modes of operation. Various such subsets will have different interpretations and flesh-and-blood

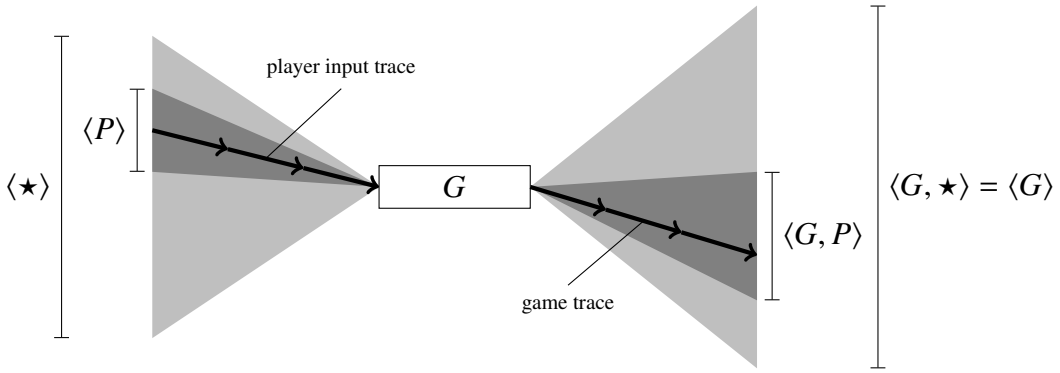


Figure 20: Schematic view of a game G as transducer from player input traces to game traces. On the left, a particular player model P is extensionally defined by a bag of possible input traces it can produce, $\langle P \rangle$, which in this case is a subset of the maximal player model, $\star$, the player which can produce any possible input trace. When a player gives a particular trace of input to the game, this results (possibly nondeterministically) in a game trace, illustrated by the arrow. The set of possible input traces $\langle P \rangle$ therefore is transduced to a set of possible game traces $\langle G, P \rangle$. The maximal player model captures all possible input traces, $\langle \star \rangle$, which, when used to operate G, can produce the set of game traces $\langle G, \star \rangle$, *i.e.*, all possible game traces this game is capable of producing under any circumstances: $\langle G \rangle$.

correspondents.

The purpose of introducing such a model is that we can then use $\langle G, P \rangle$ to describe the set of game traces that game G can produce when player P plays the game. Clearly $\langle G, P \rangle \subseteq \langle G \rangle$, since no operator, by definition, can drive a machine to outputs beyond the total set of outputs the machine can produce. It may also be helpful to think of the maximal player model, *i.e.* the one comprised of all possible player input traces, as a special model called $\star$, in which case $\langle G, \star \rangle = \langle G \rangle$. This equality clarifies that $\langle G \rangle$ is just the game's possibility space without any assumed restrictions on player behavior. Then $\langle G, P \rangle$ for a P where $\langle P \rangle \subset \langle \star \rangle$ is the—possibly smaller—possibility space of game G when we assume more restrictions on what kind of input player(s) P will feed it.[3] Figure 20 illustrates these relationships schematically.

[3]Only *possibly* smaller, because it's possible that some inputs to a game, at least in some situations, simply don't make any difference to the way the game progresses, so a smaller $\langle P \rangle$ doesn't necessarily imply a smaller $\langle G, P \rangle$.

6.4 Analytical metrics strategies

Therefore, understanding a game's possibility space involves understanding something about $\langle G \rangle$, the full possibility space, and $\langle G, P \rangle$, the possibility space that results from certain styles of gameplay. If we have a formalized model of mechanics, G, we can query it to get peeks into this possibility space, satisfy ourselves that certain things are true of it, and look at specific examples of traces that may be interesting or surprising.

I propose here seven specific strategies for interrogating game artifacts analytically, in order to gain design-time information about how the game they specify works.

6.4.1 Strategy 1: "Is this possible?"

The easiest questions to ask are of the form: can X happen? Examples: *Is the game winnable? Can the player collect every item? Can the player die while still in the first room? Can the player die right on this spot I'm clicking? Can both of these doors ever be locked at the same time?*

This strategy answers any yes/no question whose answer is determined directly by the rules of the game: whether a game state is possible, or an event can ever happen, given all possible player behaviors. In fact questions might not even involve variation on the player's part, but variation in the system: *Do enemies ever spawn at the bottom of a pit?* In short, we are asking, for some property p, is $p(T)$ true of *any* $T \in \langle G \rangle$?

This analysis strategy could be exposed directly to the designer in a query setup. Alternately, classes of answers can be visualized. For example, we can ask, for every square in a grid-divided level, whether the player could die there, and color-code the map accordingly, producing a version of the popular player-death heatmap that shows instead where it is *possible* for players to die.

Figure 21 shows on the left a heatmap of deaths in a few playthroughs of a room in a Zelda-like game; and on the right, a map of where it's possible to die, as determined through formal analysis of the rule system. There's a clear structural pattern immediately

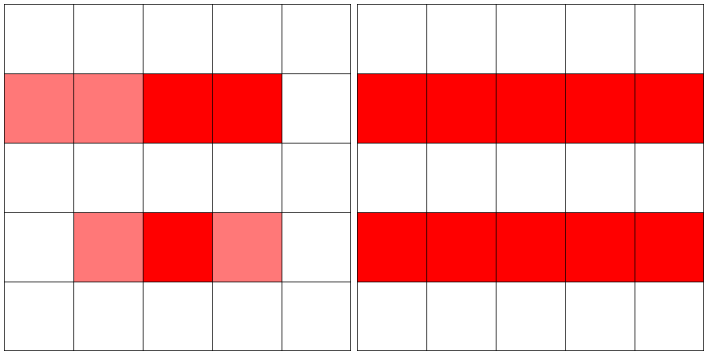


Figure 21: Heatmaps of player deaths: left is empirical deaths from several playthroughs, while right is analytically possible death locations.

visible in the second figure, derived from the game rules rather than from empirical playtest data: the player can only die in two specific rows. In the first figure, this pattern hasn't quite been made clear via the pattern of empirical player deaths. Especially with more complex patterns, it can take a lot of playtesting data to spot these sorts of structural features in the possibility space, which are usually caused by unnoticed interactions of rules, or interactions between game mechanics and level design. In addition, it can be useful to have both kinds of heatmaps, to allow the designer to disentangle which patterns are caused by the possibility space, and which are caused by patterns within player behavior. A figure like the second one can even be used to tweak level design at a more detailed level, *e.g.* to place safe spots. Finally, the analytical heat map can be immediately updated after any changes, whereas with empirical heatmaps, there is a significant latency between the designer making a change and having access to new, post-change metrics data.

In a logical framework, this strategy is implemented via theorem-proving, querying a model for true/false answers. Several alternate techniques have been used in existing work. Salge *et al.* [104] playtest games with a simulated player that evolves itself in order to try to achieve particular outcomes. More specific (and likely more efficient) algorithms can be used for limited cases of possibility queries. For example, flood-fill algorithms are sometimes used in existing level-design work to make sure there are no disconnected parts of a level, and graph-reachability algorithms can be used for similar purposes. A challenge

```
happens(move(player,north),1)
happens(attack(monster,player),2)
happens(attack(monster,player),3)
happens(die(player),3)
```

Figure 22: Gameplay trace of a player dying.

with using these special-case algorithms in a larger system is to automatically recognize when they're applicable, and in which variants; for example, basic flood-fill suffices for reachability in simple cases, but produces incorrect results (at least without augmentation) if *static* reachability isn't equivalent to dynamic reachability, for example because available paths through the level can be modified by movable walls or locks and keys.

6.4.2 Strategy 2: "How is this possible?"

Beyond finding *that* something is possible, a designer often wants to know *how* it could happen. In player metrics this is answered by collecting a log or trace of the actions the player took, along with information about game state (such as the player's position and health). These traces can be used both for debugging (to track down how something happened when it shouldn't have been possible), and as a way to understand the dynamics of the game world, by looking at the paths that can lead to various game states.

Figure 22 shows a fairly boring event log of a player dying after walking up to a monster, being attacked by it, and doing nothing in response. This is, like strategy 1, simply asking whether $p(T)$ is true of some $T \in \langle G \rangle$ for a property p—but then printing T rather than answering yes or no. The trace could be elaborated with the value of various game states at each point in the log, such as the player's position and health. However, its boringness naturally raises the follow-on question: can you show me not only ways that something is possible, but *interesting* ways that it could happen? For some outcomes, like those that should never happen, any log is interesting, but for others this is a trickier question. One approach is to let the designer interactively refine the trace they're requesting. In this example, they could ask for a way the player dies *without* ever standing around doing nothing,

and then go on to add more caveats if the result were still too mundane; I refer to this as *trace zooming* (see Chapter 8 for some more detailed examples).

In a logical framework, this can be posed as an abduction problem: finding a sequence of events that, if they happened, would explain how the sought-after outcome could come about. Zooming is implementable via abduction with added constraints on the explanations.

In a simulation framework like that of Salge *et al.* [104], the gameplay log of how something is possible would simply be a recording of the actions taken during the successful simulation run (although it may take time to recompute new runs if something like interactive trace zooming is offered). It's also possible to view finding a path to an outcome as a planning problem within the story world, and use a classical AI planner. For example, Pizzi *et al.* [91], using a heuristic search planner, find solutions to game levels and display them as comic-like sequences of illustrated events.

6.4.3 Strategy 3: Necessity and dependencies

Once we know what things are possible, and how they can happen, we might also want to know what *must* happen. Can you beat *Final Fantasy VI* without ever casting "Meteo"? Which quests can the player skip? Is this powerful sword I just added to the game needed or superfluous?

These kinds of questions also relate to the questions that can be asked via the first two strategies. Some kinds of necessity questions can be rephrased in terms of whether it's possible to reach a particular state that *lacks* the property we want to determine the necessity of. For example, whether it's necessary to level-up to level 5 before reaching the second boss is equivalent to asking whether it's possible to reach that boss while at a level of 4 or below. Other kinds of necessity questions can be rephrased in terms of zooming in on traces. For example, asking whether a particular sword is necessary to beat the game is equivalent to asking for a gameplay trace where the player beats the game, which doesn't contain the pick-up-that-sword event.

In empirical playtesting, it's common to collect metrics about usage: how many players achieve a particular quest, use each item, etc. Similarly to how we can juxtapose empirical data with analytically determined possibilities in strategy 1, in this strategy we can juxtapose empirical data with analytically determined necessities. Of course, if the empirical results show less than 100% for some item or event, it couldn't have been necessary, but on the other hand there may be things that 100% of our playtesters did which *aren't* actually necessary, which this game-analysis strategy would distinguish.

More automatic dependency analysis is also possible. For example, asking whether it's necessary for event A to precede event B, or vice versa, can let us build up a graph of necessary event ordering, which at a glance indicates some of the causal structure of the game world. That includes causal structure that wasn't explicitly written; for example, entering a locked room might have several explicit preconditions, like the player needing to find a key before entering, but also several implicit preconditions caused by interaction of other rules, like the player needing to find a particular suit of armor before entering (because there is simply no way they can successfully get to the room without having first acquired that suit of armor).

To my knowledge, no existing game-analysis work explicitly aims at this kind of automatic necessity or dependency analysis. In a logical framework, it's most straightforward to follow the approach, described in this section, of reducing necessity and dependency analysis to a series of queries implemented using strategies 1 and 2.

6.4.4 Strategy 4: Thresholds

Sometimes there are magic numbers delineating the boundaries of possible behavior, or of a certain regime of game behavior. *What is the shortest possible time to complete a* Super Mario Bros. *level? What range of money could a* SimCity *player possess when five minutes into the game?* This is asking for the *bounds* $\operatorname{argmax}_T p(T)$ for $T \in \langle G \rangle$ for a property p (equivalently, $\operatorname{argmin}_T p(T)$).

This strategy can give useful information often not discovered in initial playtesting, for example by finding speedruns of a level, or cheats to quickly finish it, that the usually not-yet-expert-at-the-game players in a playtesting session wouldn't have found. In addition, it can be paired profitably with empirical player data to give an idea of how close the particular range of data being observed comes to the theoretical bounds that the game's rules define. For example, any graph that graphs players as a distribution on a numerical scale could also draw bounds of the largest and smallest possible value—and not just largest or smallest *a priori*, as in a score that's defined to range from 0 to 100, but actually possible in a specific context, as in perfect play achieves a score of up to 97.

In addition to telling us whether it's possible for there to be playthroughs significantly different on a particular metric's axis, compared to the empirically observed ones, the relationship between the empirical range of data and the theoretical extrema can tell us something about the players in our playtest. For example, in the experiment described in Chapter 8, we discovered that the typical players in our playtest of an underground-mining game were much more cautious with returning to the surface to refuel than was strictly necessary. This information can then be used in concert with strategy 2 (finding traces with specific properties) to ask for examples at the thresholds, *e.g.* an analytically generated speedrun that beats a game as fast as possible.

In a logical framework, thresholds can be implemented using a branch-and-bound method. A possible solution (of any value) is first found, and then a constraint is added that a new solution be better than the one already found (*e.g.* shorter, if we're looking for the shortest playthrough). Then we look for another solution meeting the new constraints, and repeat until no additional solutions are found. This has the advantage of generality, but can be slow.

Future work on the problem could explicitly use an optimization method, whether a randomized one like genetic algorithms, or a mathematical one like linear programming. In addition, there are a wide range of algorithms to find maximal or minimal solutions to

more specific problems. For example, given a model of a level, we can find the shortest spatial path through the level using a standard algorithm like Dijkstra's algorithm. As with the specialized algorithms in Strategy 1, a difficulty in using these specialized algorithms would be automatically determining when they're applicable; for example, the shortest spatial path through a level may not be the shortest actually achievable path, given the game mechanics—it might not even be a lower bound, if the mechanics include teleportation. On the other hand, these kinds of differences might also give information; the difference between the shortest spatial path through a level and the shortest path that a player could possibly achieve through a level might give an indication of its worst-case difficulty, for example, since it would mean that there is some minimal level of off-perfect-path movement the player would have to perform.

6.4.5 Strategy 5: State-space characterization

The strategies so far can be seen as trying to get looks into a game's overall possibility space, $\langle G \rangle$, from various perspectives. Could we more directly analyze and characterize the possibility space itself?

One possibility is to try to visualize it. In any nontrivial game, the full branching state graph will be unreasonably large to display outright. However, it may be possible to cluster or collapse the possible dynamics into a smaller representation of meaningful states and dynamics [13]. In addition, techniques developed for summarizing the state space of empirical player data could be applied to summarizing sampled traces from the overall space of possible playthroughs [3].

More interactively, single traces can display some information about their neighbors in the state space; for example, branch points might show alternate events that could have happened at a given point in the trace, besides the event that actually happened in that trace. This can be used to "surf" the space of possible playthroughs, in a more exploratory manner than the specifically requested traces returned from strategy 2. Alternately, we

can start with empirically collected traces and observe their possibility-space neighbors; this can be used to bootstrap a small amount of playtesting data into a larger amount of exploration, by showing possible—but not actually observed—gameplay that is similar to the observed gameplay.

Moving into more mathematical territory, games largely based on mathematical approaches such as differential equations, influence maps, and dynamical systems [60] might be analyzed using standard mathematical approaches, such as finding fixed points or attractors or displaying phase-space diagrams. Some basic experimentation along these lines is sometimes done during Excel prototyping, but this area (to my knowledge) remains largely unexplored.

This strategy is perhaps the least well suited to logical methods, which aim at answering specific queries rather than developing general characterizations, requiring some machinery beyond basic inference. However, the trace-surfing approach is implementable by collecting many traces sampled through strategy 2, and displaying them (perhaps generating more as needed) interactively.

6.4.6 Strategy 6: Hypothetical player-testing

The first five strategies investigate the game's overall possibility space, $\langle G \rangle$, that is, $\langle G, \star \rangle$ in the language of Section 6.3: the investigation assumes no constraints on player behavior. We can probe the way the game responds to player interaction more specifically by trying to characterize only how a game operates with a particular, perhaps highly simplified, model of a player. That is, choose a P, and probe $\langle G, P \rangle$.

Such a restriction is not intended mainly to insert a *realistic* player model, but to investigate how the game operates in various extreme or idealized cases. For example, what happens when the game is played by a player who always attacks, except heals when low on health? If that player does very well, the game might be too simple. Or, in a multiplayer game, different players could be pitted against each other to see how they fare, which might

tell us something about the design space.

Concretely, this consists of implementing versions of strategies 1–4 conditioned on the player model, finding necessities, dependencies, etc., but simply applying the strategies to $\langle G, P \rangle$ rather than the full $\langle G, \star \rangle$.

In the case of a *fully* specified player model, indicating what the player will do in any possible situation, this can be performed by forward simulation. An advantage of our logical framework, however, is that we can query possibilities even under partially specified player models, that specify constraints on player behavior while leaving some parts open.

Approaches based on more direct implementations of forward simulation include: the Machinations system [21, 22], which simulates hypothetical players playing a Petri net game model, and collects outcomes after a number of runs; and Monte-Carlo "rollouts" of boardgames that pit two possible strategies against each other in a specific point in the game, to determine how they fare against each other [128].

6.4.7 Strategy 7: Player discovery

While hypothetical players can be useful for probing how a game behaves under various kinds of gameplay, we found (see Chapter 8) that designers often had difficulty inventing such hypothetical players. Instead, they wanted the process to work backwards: given a game prototype, can we automatically derive a simple player model that can consistently achieve certain outcomes? For example, rather than having to try out questions such as, "can this game be beaten by just mashing *attack* repeatedly?", some designers would prefer we analyze the game and come back with: here is the simplest player model that consistently beats your game.

That question can be seen as a stronger or more generalized version of trace-finding (strategy 2). Finding how a particular outcome is possible returns *one* possible instance where it could happen. Finding a player that can consistently make the outcome happen is a compressed description of many such instances. That is, the designers here are asking us

to find a player model P such that, for some property p, $T \in \langle G, P \rangle \implies p(T)$.

There are several ways to invent these kinds of player models. One approach is to sample many possible traces reaching the requested state (using techniques from strategy 2), and then inductively extract a player model from these traces, or perhaps several player models from different clusters of traces. There are already techniques from empirical gameplay metrics that can be used to infer such player models [24], which could be applied to extracted gameplay traces instead.

In a logical framework, a more direct approach is to directly infer whether there exists a player model from a class of simplified players that can reach the desired state, using logical abduction. For example: is there any single button a player can mash constantly to beat the game? If not, is there a 2-state button-mashing finite state machine that can consistently beat the game (perhaps alternating between "attack" and "heal")? If not, we can query for state machines with more states, or other kinds of more complex player models. These player models can be used to characterize the complexity of the game by varying their complexity on several axes. One axis of complexity is how "blind" the player model is: the button-mashing or alternate-between-two-states model ignores the game state completely. If there's no simple blind finite state machine that can beat the game, how about one that only looks at one game state (or two game states)? If a player model of *that* kind exists, it would tell us something about the game state that is relevant for decision-making.

In multiplayer games, player discovery can be related to game-theory terminology, such as finding optimal strategies (for various kinds of optimality), dominated strategies, etc. While using game theory for videogame analysis or balancing has often been discussed, it seems to resist practical application in part due to the mismatch in scale between the size of games that game-theory software typically handles, and even small videogame prototypes. In particular, most computational game theory assumes either a one-step game, or an iterated (staged) game with relatively few steps, typically as few as three or four; whereas most videogames go on for many timesteps, and have their dynamics emerge over at least

slightly longer timescales. Overcoming this problem has recently been tackled in work by Jaffe *et al.* [43] in developing a framework, called *restricted play*, that tests games against carefully restricted agents.

Finally, a large class of gameplay algorithms can be used as player-discovery algorithms of a sort, especially if they produce interesting internal structure that tells us something about the game they learn to play. For example, a reinforcement-learning algorithm that learns a state-value function would be an interesting source of data for augmenting the kinds of state diagrams in strategy 5, adding to them information about how valuable each state is from the perspective of reaching a particular goal.

CHAPTER VII

MODELING GAMES IN LOGIC

To perform the kinds of game analyses proposed in the preceding chapter, the game's rules need to be in a machine-readable form, so we can use them to infer things about the game. But which kind of machine-readable form? Any computer game is machine-readable in the sense that it can be executed. That is one particular kind of inference about a game's rules: temporal projection, the ability to determine what would happen to a rule system if it were executed forwards in time with particular inputs. But we'd like to perform many other kinds of inference besides playing a game.

Some other kinds of inference can be derived from temporal projection, via black-box simulation: sample many possible playthroughs of a game and mine information from the sample. But if we actually have the rules of the game, why not open up that black box, and use them directly? For example, it does not require simulation to determine that, in chess, a bishop that starts on a black square can never end up on a white one: that is a direct consequence of the rules of how bishops can move (only diagonally) and the board layout (diagonally adjacent squares have the same color).

To capture mechanics in a fluent, elegant way, the representation should correspond fairly closely to the mechanics descriptions we'd get if we asked a game designer to write down the basic mechanics of a game prototype—think of writing down the rules that make up Space Invaders in a dozen or so bullet points. Moreover, to be usable in an iterative-design setting, these formal representations need to be easy to modify—imagine adding or removing one of those bullets.

There are some existing game-description formalisms, which have their advantages and

disadvantages (and the advantage of already existing), but ultimately do not suit my purpose here. One class are those which are easy to modify, but special-case and not fully declarative. These are most often designed as parameterized representations of a particular space of games. Examples include METAGAME [90], a grammar-based formalization of variants of chess-like games, and Togelius & Schmidhuber's [131] encoding of Pac-Man-esque games into chromosomes suitable for evolutionary computation. At the other end of the spectrum are fully declarative logic-based formalizations, of which the primary exaple is the Stanford Game Description Language, GDL [57]. While superficially seeming to be suitable, this has two major downsides. Descriptions are generally quite brittle, modifiable only through complex "formula surgery" that can approach re-formalizing a domain when changes are made. And the targeted domain is almost exclusively oriented towards board games, which makes applying it to videogame design quite awkward.

I propose instead that the properties we'd like in a game-mechanics representation are similar to those studied under the label of "commonsense logics", and I adopt one such logic, the event calculus, as particularly well-suited for representing videogames.

Symbolic logic as a general category is a way to declaratively specify the sort of white-box knowledge we want, without limiting up front what kinds of inferences we want to draw from it. The most well-known variety of symbolic logic is standard first-order logic. We'll start from there, but then note its drawbacks, and instead adopt the event calculus (EC), a temporal logic for reasoning about dynamic artifacts. In addition to its built-in formalization of *time*, a rather important aspect of a videogame-mechanic formalization, it also aims at *elaboration tolerance* [65], a concept from John McCarthy that proposes editing logical formalisms should be possible to do locally, *e.g.* by adding a new rule, without requiring far-reaching reformalization.

7.1 First-order logic

First-order logic is as close as one gets to a "standard" logic, and serves as a point of departure for many other logics, which reuse its basic terminology. First-order logic formulas are built out of *predicates*, *functions*, *constants*, *variables*, *quantifiers*, and *connectives*. Formulas are interpreted over a *domain of discourse*, which consists of the entities we are making statements about.

- A *predicate*, such as $is\_weapon()$, takes zero or more parameters, and is true or false for each choice of parameters. Predicates are used to make statements about what is true in general, and what is true of which things. We will also assume the usual arithmetic predicates in infix notation, such as $\geq$.

- A *function*, such as *mayor*(), which takes zero or more parameters, and gives an entity from the domain. For example, *mayor*(*chicago*) names a specific person.

- A *constant*, such as *broadsword*, names a specific entity in the domain. We write these in lowercase, to distinguish them from variables, written starting with a capital letter.[1]

- A *variable*, such as *P*, can take on the value of any entity in the domain.

- The two *quantifiers*, for-all ($\forall$) and there-exists ($\exists$). These specify, respectively, that a formula holds for all possible values of a variable being quantified, or that it holds for at least one possible assignment.

- The standard logical *connectives*: conjunction ($\wedge$, "and"), disjunction ($\vee$, "or"), negation ($\neg$, "not"), and implication ($\rightarrow$, "implies").

[1] This is the convention used in logic programming. Confusingly, traditional mathematical notation uses the opposite convention.

7.1.1 Herbrand interpretations

When used for knowledge representation in artificial intelligence, and especially in logic programming, the interpretation of a set of formulas is commonly restricted to the *Herbrand interpretation*, an intepretation with a few simplifications.

The first is that distinct constants are taken to name distinct entities: $broadsword \neq hat$ in any interpretation. While this is often intuitive, it does rule out interpretations that could be true in a more general semantics of first-order logic: for example, if we saw an unknown person at the mall and denoted them $mallperson$, and then another unknown person at a restaurant and denoted them $restaurantperson$, in a Herbrand intepretation by giving them these constant names, we would be committing to the view that they are not in fact the same person. The plus side is computational tractability and the fact that we don't have to explicitly specify that $broadsword$ and hat are in fact distinct entities, which is a case that comes up considerably more often in specifying game mechanics.

The second is that functions are interpreted as entity constructors, essentially a way of naming an entity by reference to a constant or variable. For example, if every town has a mayor, $mayor(C)$, would be a way of referring to a city's mayor without giving it a separate constant name. These limited kinds of functions cannot be used for inference, only naming: if mayors of towns are a piece of state that needs to be inferrable, then we would need to encode the function as a relation, $mayor(P, C)$, which could be true or false for a particular person and city.

The third simplification is that the universe of description consists only of explicitly named entities: those names by constants, and those named by functions. Since these are finite in our consideration, this implies a finite domain. Apart from being a necessary assumption to use many kinds of inference engines in practice, this simplification suits videogame description fairly well, since unlike writing logical formulas describing the real world, we are not trying to build a model of a large, external world with possibly unknown entities, but constructing a prototype which is expected to have exactly those entities we

put into it.

7.1.2 Formalization example

Here is how one might formalize a snippet of a simple weapon-equipping logic in first-order logic:

$$is\_weapon(broadsword)$$

$$damage(broadsword, 3)$$

$$strength\_needed(broadsword, 5)$$

$$\forall P, S, SN, W(strength(Player, S) \land strength\_needed(W, SN) \land equipped(P, W) \rightarrow S \geq SN)$$

$$\forall P, W(equipped(P, W) \rightarrow \neg\exists W2(W \neq W2 \land equipped(P, W2)))$$

The first three formulas assert facts, in this case facts about the item *broadsword*'s properties: it is a weapon, deals damage of 3, and requires a strength of 5 to equip. The fourth and fifth formulas can be glossed as follows:

- If a player P has some strength S, and a weapon W requires some strength SN to equip it, and is currently equipped by the player, then the player's strength S must be at least the minimum needed to equip it, SN. Because $A \rightarrow B$ is equivalent to $\neg B \rightarrow \neg A$, this implies that if the player's strength is *not* at least some threshold, then weapons with a strength needed above that threshold are not equipped.

- If a player P has a weapon W equipped, then there is no other weapon $W2$ that they have equipped. This is written above in the way a designer might think of it, but a logician would typically remove the negations and rewrite the formula as:

  $$\forall P, W, W2(equipped(P, W) \land equipped(P, W2) \rightarrow W = W2)$$

  i.e., the player can only have two weapons equipped if they are in fact not distinct.

From these declaratively specified formulas, we can infer that certain things are true or false. Can a player of strength 4 equip the broadsword? No. Why? From the three facts

$strength(player, 4)$ (given), $strength\_needed(broadsword, 5)$ (part of the definition), and $\neg(4 \geq 5)$ (numerical fact), the fourth formula in the snippet of game formalization allows us to conclude $\neg equipped(player, broadsword)$.

Apart from a certain cumbersome verbosity in some of the specifications (solvable with syntactic sugar), this covers specifying *static* aspects of a game's mechanics reasonably well. But things get more complex when state changes over time. A broadsword's damage is always 3 in this game, so we can write $damage(broadsword, 5)$. But a player's hitpoints vary, so we can't write something like $hitpoints(player, 35)$. Sometimes $hitpoints(player, 35)$ would be true, but other times $hitpoints(player, 20)$ might be true.

To deal with time-varying state, we need to qualify predicates with a time—not just true or false, but true or false *when*. One way is to add a time parameter to all predicates that can change. Thus $hitpoints(player, 35, 0)$ might indicate that the player starts the game (at time 0) with 35 hitpoints.

Once this time parameter is added, we need a way to specify how the values evolve over time—for example, hitpoints go down if a player suffers an attack. That is: if a player has H hitpoints at time T, and they are (successfully) attacked at time T by an attack causing damage D, they will have hitpoints $H - D$ at time $T + 1$.

To implement this scheme, we need a notion of *events*, which happen at times. We'll introduce a predicate $happens(E, T)$, indicating that an event E happened at time T. Then we can formalize the player taking damage:

$$\forall P, T, D, E, H(hitpoints(P, H, T) \wedge happens(E, T) \wedge damage(E, D) \rightarrow hitpoints(P, H-D, T+1))$$

This forms the minimum of a calculus of time-varying state, along with events that change (or can be caused by) that state. Fortunately, this maps closely to a more general system that has already been worked out to provide such a formalism, called the event calculus.[2] By adopting the event calculus, compared to the approach above, we gain a

[2] In particular, the discrete-time event calculus [68].

more general way of specifying which events change which state, along with a body of representational idioms that have been worked out for formalizing various situations; the latter is covered well in a book-length treatment by Mueller [69]. In addition, it solves some problems for us most robustly. Looking at the formula above again, we may ask: what happens if the player *isn't* damaged by an event at time T? It seems the default shoul be that they have the same hitpoints at time $T+1$. But as written this doesn't happen unless we write a second formula that is almost the same as the first, but indicates that if the things mentioned in the first don't happen, then the hitpoints stay the same. We'll return to this problem in Section 7.2.1.

7.2 The event calculus[3]

The event calculus is, as its name implies, a formalism for events, along with the time-varying state they interact with. It codifies, within first-order logic, the addition of these two kinds of entities (events and time-varying state), and axioms specifying their relationship.

Rather than adding a time variable to each time-varying predicate, as our ad-hoc example in the previous section, the event calculus introduces the notion of a *fluent*, which is essentially a time-varying predicate. A fluent looks like a predicate, but rather than referring to it as a bare predicate that can be true or false, full stop, it must always be accessed via another predicate that clamps the reference to a specific time, at which it's true or false.

Thus rather than $hitpoints(player, 35)$, where hitpoints is a predicate, we turn it into a fluent, and write:

$$holdsAt(hitpoints(player, 35), 0)$$

This is of course essentially the same as our ad-hoc solution of $hitpoints(player, 35, 0)$, but with the idea of adding time to a predicate taken up a level to the *holdsAt* predicate. From a practical perspective this will allow us to now write axioms about *holdsAt*, which

[3] A version of this argument in favor of modeling videogame mechanics in the event calculus was previously published as [77].

will apply for any fluent, instead of having to repeat them for every time-varying predicate.

Events are codified as we introduced them in the previous section: an event can *happen* at a time. Often events will be *terms* rather than *atoms*, *i.e.* constructed via functions with parameters. For example, to indicate that a foe F attacks player P at time 10, we write:

$$happens(attacks(F, P), 10)$$

Given *holdsAt* and *happens* as ways of accessing fluents and events, the dynamics of the game can then be written as formulas that relate fluent change and events. Now, we can write a game mechanic saying that two sprites $S1$ and $S2$ collide if they're ever at the same point:

$$\forall T, S1, S2, X, Y(holdsAt(at(S1, X, Y), T) \wedge holdsAt(at(S2, X, Y), T) \rightarrow happens(collide(S1, S2), T)$$

7.2.1 The frame problem and EC solution

The frame problem is a general term for the difficulty of *completely* formalizing a situation, in a way that accounts for all the "background" or "default" assumptions [66]. It manifests in a number of ways, but as relevant here, it involves the problem of when something *doesn't* happen in a dynamic situation: when does state *not* change, and what effects do events *not* have?

A traditional solution is to explicitly specify the answers to those questions, using *frame axioms*. A frame axiom simply specifies "non-dynamics", the conditions under which things don't change. Returning to the hitpoints example, we might write,

$$\forall P, T, E, H(hitpoints(P, H, T) \wedge (\neg happens(E, T) \vee \neg damages(E, P)) \rightarrow hitpoints(P, H, T+1))$$

This explicitly specifies that the player's health remains the same unless an event that damages the player happens. However, this requires maintaining the frame axioms so they're consistent with the "positive" dynamics. If we add a new "heal" event that increases health when it happens, the frame axiom must now be edited to say that the player's health

stays the same unless an event that damages the player *or* an event the heals the player takes place.

Is there a way we can automate this process? What we want intuitively is a sort of "default no" for dynamics: the dynamics are what *and only what* we specify. That is the familiar behavior from normal programming, where game state only changes when a piece of code makes it change, and events only change things that they are explictly told to change.

7.2.1.1 Inertial fluents

The first step towards saner defaults is to make state *inertial*: true fluents stay true unless something makes them false, and false fluents stay stay false unless something makes them true. To implement inertia, we need state changes to be encapsulated. The event calculus provides two predictes to do so: $initiates(E, F, T)$ says that if event E occurs at time T, then fluent F will be true after T. Its opposite is $terminates(E, F, T)$, which says that if event E occurs at time T, then fluent F will be false after T. Now we can write formulas, which are two of the axioms of the event calculus (discrete-time version) to implement inertial fluents:

$$\forall F, E, T(holdsAt(F, T) \land \neg\exists(happens(E, T) \land terminates(E, F, T)) \rightarrow holdsAt(F, T + 1))$$

$$\forall F, E, T(\neg holdsAt(F, T) \land \neg\exists(happens(E, T) \land initiates(E, F, T)) \rightarrow \neg holdsAt(F, T + 1))$$

In place of the frame axioms, which we had to keep updated, we have two EC "universal" frame axioms implementing non-changing state in general [109].

The *initiates* and *terminates* predicates are themselves implemented with another two EC axioms:

$$\forall F, E, T(happens(E, T) \land initiates(E, F, T) \rightarrow holdsAt(F, T + 1))$$

$$\forall F, E, T(happens(E, T) \land terminates(E, F, T) \rightarrow \neg holdsAt(F, T + 1))$$

7.2.1.2 Circumscription

One more addition is needed to do away with frame axioms entirely. To infer that a fluent retains its value with the EC's inertia axioms, we need to be able to *show* that, for each event, either the event didn't happen at a given time, or that it didn't change the fluent, so that one of the axioms applies. This would seem to still require a type of frame axiom, where we have to explicitly specify when events don't happen (*e.g.* collisions don't happen when objects *aren't* in the same location), and which fluents they don't *initiate* and *terminate* when they do happen.

Fortunately, the addition of inertia removes the temporal dimension of the problem, and reduces it to the problem of how to encode a "default false", which has a number of solutions. The event calculus is therefore usually used in conjunction with McCarthy's *circumscription* [64] semantics, a nonmonotonic logic that minimizes unnecessary inferences. The notion from logic programming of *negation-as-failure*, that anything that can't be proven true in a model is false, is perhaps a more intuitive statement of the semantics we want: anything we did not specify as dynamics of a game, and which isn't derivable from those dynamics, is not in the game's dynamics. The two notions turn out to be the same for our purposes: in the context of the event calculus, circumscription is equivalent to negation-as-failure in the stable model semantics for logic programming [49]. We'll use this version in the prototyping tool described in the following chapter, since logic-programming systems provide a practical basis for implementation.

With the circumscriptive, inertial event calculus, we have a way of formalizing game mechanics in a reasonable way—not reasonable for a designer to write directly, it may be admitted, but with a reasonable semantics that maps well to the positive specification of a game's dynamics. In the next chapter, I build a prototype tool encoding game mechanics in the event calculus, and investigate what designers think about the usefulness of its inferences. Then, I look at how the same game prototyped in the event calculus can be both *played* in real time, and queried for analytical feedback as described in Chapter 6, making

both sources of data available to a designer.

CHAPTER VIII

A LOGIC-BASED PROTOTYPING TOOL[1]

Drawing on the history of design-support systems recounted in Chapter 2, and the formalization of abstract game mechanics developed in Chapters 6 and 7, we are in a position to build a prototyping tool to assist designers in understanding early-stage, mechanics-heavy prototypes. But what precisely should such a tool look like? What would designers find useful? To answer those questions, I conducted a series of interviews with three teams of independent game designers, to determine if and how an automated game-mechanics analysis system could be useful for answering design questions that they had about game designs they were working on at the time of the study. The findings of that study are discussed in Section 8.1.

Then, I developed (in close collaboration with Adam M. Smith [118, 119, 120]) an early-stage prototyping tool, in which a game's mechanics are formally encoded in the event calculus, and connected to interface elements inspired by paper prototyping (tokens, spaces, and lines between them). These prototypes are both *playable* (via temporal projection in Prolog) and *queryable* (via queries fed to an answer-set-programming solver). I discuss some results we found juxtaposing these two playtesting methods on a formalization of a popular Flash game, *Motherload*.

8.1 Designer interviews

To validate the concept of a game-design assistant that helps designers reason about the interaction of game mechanics, and to collect a set of requirements for the kinds of reasoning it should be able to perform, I conducted a study with three small teams of independent

[1] Parts of this chapter have been previously published as [78, 118, 119].

game designers, each of whom was in the midst of a design project. We followed a contextual design methodology [143], investigating to what extent a game-design assistant would be useful for the design questions they were facing at the time, by proposing and testing out design-assistant prototypes on the problems they were actually working on. Since game designers don't necessarily have a good model for what an assistant might actually be able to do for them, this required an iterative process of interviews with focusing prototypes, where feedback from one interview feeding the next focusing prototype.

Since this study investigated designers' response to the proposed mechanics-analysis system, rather than an interface or visualization, the approach was what might be termed a "reverse Wizard-of-Oz" study, using a functional backend but human operators as the interpreters between designers and the system. A regular Wizard-of-Oz study has a fully functional interface, but implements the guts of the system with humans behind the curtain (sometimes unbeknownst to the users, who are led to believe that the system is a fully computational prototype). As Nigel Cross observed when performing an early Wizard-of-Oz study (though before the term was coined), that has advantages and disadvantages: "Doing so provides a comparatively cheap simulator, with the remarkable advantages of the human operator's flexibility, memory, and intelligence, and which can be reprogrammed to give a wide range of computer roles merely by changing the rules of operation. It sometimes lacks the real computer's speed and accuracy, but a team of experts working simultaneously can compensate to a sufficient degree to provide an acceptable simulation" [18, p. 107].

This AI-centered contextual-design approach differs from a more common methodology in HCI of doing interface-centered contextual design: prototyping non-functional interfaces in order to understand how a user would interact with a system, and, from that understanding, identifying required functionality. That approach, however, tends to work well only if the functional hooks are not overly complex. During the study, the researcher needs to tell the potential user what the system would have done if it were functional; and, after the study, the identified functionality needs to be implemented to the specifications.

With complex AI systems, it is difficult to accurately tell the user what a system would have done if it had existed, and to gauge their response to this non-existent functionality; it is also fairly easy to identify wishlist features that turn out to be impossible to implement as envisioned. Therefore, I followed a functionality-first style of prototyping, identifying what it is designers would like such a system to be able to do. I served as the interface, translating designers' queries into the system, and translating the answers back for them; this allowed me to get at what queries, if any, are useful, even if it were not easy for designers to formulate he queries in symbolic logic. This approach does still contrast from a purely AI-driven development style, by applying HCI-derived methodology to the design of the AI system's features.

I started by interviewing each team about their project, current design questions, and existing prototyping techniques. This wasn't intended as a full ethnographic interview, but rather as a light-weight process mapping that allows me to understand enough of their design practice to converse with them intelligently in the rest of the interviews, make sensible proposals, and refer to agreed upon elements of their design process [88].

From there, I proposed some scenarios where we thought a system that reasons about mechanics could provide answers to relevant design questions. These proposals varied from floating an idea to see if it sounded interesting to the designer, to paper mockups of a hypothetical interface, to prototypes of a backend reasoning system that could provide answers in specific situations.

A particularly strong participatory aspect of this design process was necessary for a number of reasons. There is little existing empirical work on game design, so beginning with a purely observational study to understand how designers work, and using that as the basis for designing a system, would be unlikely to serve as a solid basis for the design of a game-design assistant in the short to medium term. In addition, game design is, as with many creative design practices, quite idiosyncratic, with design styles often strongly influenced by a designer's personal design practices. As a result, a significant degree of

deference is necessary to designers' control of their own artistic practices, and therefore we need specific opinions and reactions about how the proposed tool might fit into those practices. On a more practical level, independent game designers, the most likely early adopters of such a tool, have a large degree of control over the tools they use, so perceived usefulness is at the very least necessary for such a tool to be actually used.

Of the three teams of independent game designers interviewed, one consisted of a single individual, and two of two-person teams. To avoid publishing preliminary design information, I partially anonymize two of the three case studies, by substituting similar examples from existing commercial games when specific references to game-design features are necessary, and discussing other design issues in general terms. The first case below, however, is discussed without anonymization by agreement with the designers, Chronic Logic.

8.1.1 Case study 1: NARPG

NARPG, for "Not an RPG", is partly a parody of the gameplay of role-playing games (RPGs), especially the kinds in which the player spends the majority of her time fighting battles in order to collect items that defeated enemies drop, known as "loot". In NARPG, the battles are automated, and the gameplay consists entirely of picking up loot, fitting it into the inventory, equipping or de-equipping armor and weapons, using health potions, and so on. The design goal is to create a casual game, playable by non-hardcore gamers in small slices of time, in which the primary gameplay task is a puzzle-like optimization of inventory. Players have to decide when they should pick up valuable items (such as sacks of gold) and useful items (such as armor), given a fixed-size inventory (though some items may change the size of the inventory) and physical shape constraints in the inventory (represented on a 2D grid).

In this case study, the designer was in the later stages of design; the core mechanics (fundamental rule systems) had already been established when I began interviewing. The design effort was therefore focused on level design. During level design, given fixed core

mechanics, the designer creates objects and spatial layouts that appropriately balance challenge and reward. Design questions that arose during level design for NARPG include: Are some items unnecessary, in that a player can effectively ignore them and still win? Are some items too powerful? Given a specific level (spatial layout, enemy encounters and objects), how well do various player strategies (as suggested by the designer) fare?

When I began working with the designers, they were answering these sorts of questions via cycles of modifying and playing a prototype version of the game, which was more or less a working version of the game with placeholder art and interfaces.

After implementing a formalized version of their game mechanics in our reasoner, I worked with them to answer a series of design questions. One large category of design questions they had was what gameplay would be like for different types of players; for example, how would the player fare who always picks up the strongest armor and weapons they can find, uses health potions, and does nothing else? While forward simulations using the standard prototyping process could begin to provide insight on this question, the formal representation of the mechanics allowed them to view simulated play traces with specific characteristics. Forward simulation within a traditional procedural prototype, by contrast, could require potentially millions of runs until one with the desired properties is generated. In addition, the designers were particularly interested in "backwards" reasoning from outcomes to mechanics changes, *e.g.* what the smallest or largest value for a particular quantity (health, sword strength, etc.) should be to still achieve a desired outcome. In general they had no shortage of design questions they felt comfortable posing to the formalized game mechanics, and found the mechanics-reasoning approach fairly easy to work into their design process. An interesting query type that the designers brought up, and that I had not previously considered, was finding player models that can achieve a particular outcome—the reverse of testing out a particular, designer-given player model to see what outcomes it produces. This is now discussed as strategy 7 in Chapter 6.

8.1.2 Case study 2: A real-time strategy game

The second case study was a real-time strategy (RTS) game in the middle stages of design. The designer had already built a series of small playable prototypes, each aimed at one sub-part of the game: the economic system, the combat system, and base building and base defense scenarios. However, the core mechanics had not yet been established. The questions being explored at this stage in his design were a mixture of mechanics and interface/player experience questions. Mechanics questions included: Do all objects (units, buildings, etc.) play a useful role? Given the interactions between the game subsystems (economy, base defense, etc.), do the gameplay dynamics avoid overly convergent, dominant strategies? Interface/player experience questions included: Do players try to do things that the game doesn't support, or not even try out things that the designer expected would be interesting? Do people figure out what to do, or get confused by the options available? Is the game fast-paced or slow-paced?

The mechanics-reasoning system didn't immediately interest this designer, partly because at this stage of the design he was more interested in gaining high-level insight into potential design directions, as well as the user-experience aspect of what sorts of gameplay would be interesting or produce the kind of experience he was after. Automated reasoning about mechanics came across to him as more useful for design-debugging questions for later in the design process, like whether certain units made the game unbalanced.

To try to get at higher-level design questions, I proposed modeling the games at a very abstract level, based in part on some of his existing boxes-and-arrow paper design sketches, representing high-level mechanics and player choices. He sketched out a visual design language, in particular for the economy aspect of the game, with an abstract view of everything in the game as a source or a sink of resources, and different types of nodes or arrows connecting them.

Since the goal of the requirements analysis at this stage was to collect requirements for

the inference machinery, the main purpose served by the prototype of a visual design language here was as *focusing prototype*, intended to elicit ideas about what kinds of reasoning questions he might want to ask, if hypothetically he had already modeled a design. However, to a great extent, he was most interested in a visual design language itself as a way of storyboarding games. Similarly, what he found most helpful about the abstract model of economies as interconnected sources and sinks was simply that it was a useful representation for thinking about the problem, regardless of whether any automated reasoning was provided. Likewise, he was interested in one proposal I suggested, an abstract rock-paper-scissors model of unit combat, mainly because it provided him with mental tools for thinking about the unit combat design problem, not because of any queries it could answer. Thus, interestingly, this designer was very interested in formal representations, but primarily not for the backend reasoning they support, but for the front-end representational ability they provide, that helps in thinking about the design. He perceived the backend reasoning as being useful for tuning a design later in the design process.

One backend reasoning application did arise from one of his prototypes. This prototype had tested out ideas for base-defense mechanics by having the player build bases with static defenses that a computer player then assaulted. He abandoned this prototype because most of the outcomes gave no design feedback besides "the computer player failed because it played stupidly". Improving the AI to fix each discovered problem was tedious and not getting at the point of the prototype, so he ended up building a two-player networked prototype, replacing the computer player with friends who volunteered to try it out. Explicit reasoning about mechanics may have provided a mechanism for getting design feedback from the first prototype: given a base configuration and a set of units, the reasoner can generate a plan to defeat the base's defenses. The designer can then compare that plan with his ideas of how he had expected the base to be (or not be) defeatable. Since in this particular case he had already moved onto testing this scenario with human opponents, it was difficult to determine what design feedback this would have given if it had been

available at the time of the abandoned prototype.

The design wasn't yet at the stage of integrating the different sub-domains of gameplay into a unified game, but he had some ideas about what prototypes he would build to do so. Most of those ideas weren't amenable to much automated support, because the main design question he had for integrating the different subsystems, at least initially, was how the player would perceive the result—how they split their attention between combat and resource management, whether they find the combination confusing or too difficult to manage at the same time, and so on.

8.1.3 Case study 3: An evolution-based game

The third case study was very early in the design phase, essentially at the point of having brainstormed some design ideas and identified avenues for exploration. It had a number of possible components, but one of the main novelties was a genetic-algorithm style dynamical system where some elements of the game evolve via mutation and crossover operations.

The main design questions at this early stage were: What's interesting about having an evolutionary dynamic in the game? What kinds of outcomes would be interesting? How can those tie in with other parts of the game, such as combat?

This presents a somewhat different set of design questions than for games further along in the design process. At this very early stage, the driving question is: What in this general design space might be interesting? Their existing design process consisted mostly of design discussions and inventions of hypothetical scenarios where the mechanic might produce interesting results, as well as scenarios where it might cause problems.

The main uses of mechanics reasoning at this stage of the design was to enable designers to more quickly answer "what if?" or "could that happen?" sorts of questions that came up during brainstorming. For example, a simple evolution model can be used to check what outcomes are common, whether specific queried outcomes are possible, whether any of the outcomes from a class of outcomes are possible, and so on.

A stumbling block preventing automated mechanics reasoning from being more useful to this team at their early stage of the design process was the fact that I served as the tools' interface in the "reverse Wizard-of-Oz" style. With the brainstorm-heavy design process, they would have preferred to have a tool they could take home and interact with on their own, to really get an idea of how it could help them explore a design space, or what else they might want it to do. The opportunistic aspect of this early stage of design necessitates a tighter interaction loop than is possible with myself as an intermediary.

8.1.4 Discussion

From the case studies and responses to the series of focusing prototypes in each case, we can abstract some requirements for a game-design assistant.

There is a split between designers who primarily want a backend reasoning versus front-end modeling tool, with one of the interviewees primarily wanting the modeling tool, and two the reasoning tool. One of the designers (case study 2) worked and prototyped in considerably different ways than the other two. Whereas the design process of the other two was fairly mechanics heavy, mapping nicely to a model of automated mechanics reasoning, his was much more interface heavy. This "interface-in" rather than "mechanics-out" design style led to a number of design questions, such as questions about player perception and attention, that are difficult for an automated reasoning system to answer.

In addition, that designer, perhaps not coincidentally, had a much more interface-in view of what a design-assitant tool should do. He was most interested in the possibility of a storyboarding tool for game designers to be able to use to quickly sketch and visualize designs, with a visual design language and sets of built-in vocabulary for common design domains. This leads to an interesting proposal for what the game-design equivalent of a CAD tool's 3d modeling is: not sketching of the game's appearance in a superficial sense, but sketching of the game's interconnected processes and elements. Methodologically, this designer also disagreed with the backend-first approach: He had some interest in automated

reasoning, but mostly thought of it as a possible next step to consider after a visual modeling tool, which might present opportunities to hang automated reasoning off of some of its widgets.

When discussing possibilities with designers, I found it useful to frame most suggested queries in terms of simulation, even though they were not actually being answered by simulation. My initial attempts to follow the logical-reasoning literature's conceptual splits into simulation, planning, abduction, and so on, mostly led to confusion about what the tool could do. For example, planning, *i.e.* finding a sequence of player actions that would cause a particular outcome, is conceptually a form of directed simulation: find a simulation run that has a particular outcome, and then see what happened in it. These sorts of metaphors led to a good deal of interest in how the system could do reasoning more complicated than standard simulation, such as "backwards" reasoning to find what the largest or smallest value of some parameter would have to be to result in a particular outcome. An interesting technique that developed in several of the prototypes was using different questions to isolate different causes of outcomes. For example, sticking with a fixed player model and asking if a particular situation is possible gets at to the role world conditions play in possible outcomes; fixing world conditions and asking if a player can achieve an outcome gets at whether a particular kind of gameplay (for example, a player exploiting a design flaw) could cause an outcome.

One complication is that many questions that initially sound objective turn out to have subjective components. A designer who wants to know if there are multiple ways to achieve something usually means multiple *meaningfully different* ways, for some definition of "meaningfully different" that involves perceived player experience. A significant line of future work will be on finding ways to map these fuzzier kinds of design questions to more black and white questions that can be answered by logical reasoning. More concretely, many design questions envision games with some randomness, and are interested in frequency of outcomes, which is traditionally not something well supported by logical

reasoning.

The question-answering approach of this interview process was most useful in the designs somewhat further along, which could be seen as testing design ideas, rather than at the stage of trying to invent them. In the brainstorming stage, one of the case studies (#2) found it more useful to go to a very abstract model of the game that removed most of the literal mechanics, and preferred to use more of a visual modeling way of thinking about the design. The other (#3), seemed to still be interested in more of a mechanics-simulation approach, but wanted a tool with a reasonably user-friendly interface that could be used without us present to really integrate it into a brainstorming process.

An interesting possibility raised by this split between exploratory prototyping, which looks for possible design goals, and testing-style prototyping, which checks whether particular proposed designs have those goals, is a regression test for design. If design goals identified in the exploratory phase are noted, then during the testing phase, a series of regression tests can be run to make sure the design goals haven't been broken by recent changes, much as in software engineering regression tests check to see if previous bugs were reopened by new modifications.

One feature that designers rarely found interesting that I had thought might be useful was querying for elements of a design that meet some criterion; for example, show all enemies that could be the first enemy encountered, or all squares the player can reach without jumping. One hypothesis (besides the possibility that it just isn't a useful mode of inquiry) is that such queries would be most naturally posed in a graphical information-visualization manner, rather than literally as queries returning a list of results; for example, setting filters in a visual representation of a game to color-code objects that meet a particular property. A query mechanism would also be useful in a system that investigates design suggestions, since bits of proposed design would need to find parts of the existing design to which they're applicable.

Games with separate components that can be prototyped separately lead to considerably

different design processes from those that have a more unified core mechanic. In NARPG, which was built around a central mechanic, much of the prototyping was of the testing sort, and there was little desire for a built-in design vocabulary: the design innovations were in the core mechanic, and the vocabulary from existing games that it does use (such as battles and an inventory system) is easy enough for the designers to think about without. In the RTS, by contrast, the designer wasn't particularly focused on economy design by inclination, so would have found some prompting on how to think about economies useful. In addition, separate prototypes lead to questions of how to integrate the different components into the complete game. A regression-testing approach may prove useful there (*e.g.* to make sure changing something in the economy doesn't break something in the combat), but none of the designs had advanced to the integration stage during our study.

Finally, the direction proposed by domain-oriented design environments and especially metadesign mapped well to some of the design issues we encountered. Game-design vocabulary is a mixture of existing terms inherited from previous games (*e.g.* RTS-design vocabulary) and novel ideas. Thus designers may want the ability to design higher-level abstractions, and to import existing representations where they exist. For example, the second case study would have found it useful to have his thinking prompted by a toolbox of off-the-shelf RTS design vocabulary already encoded in the system.

8.2 The BIPED prototyping system

BIPED is a prototyping system for early-stage, mechanics-oriented prototypes, that encodes their mechanics in the event calculus, specified in a Prolog-like syntax. Designers can query the mechanics, as outlined in Chapter 6, as well as connect them to a set of graphical elements modeled on paper prototypes, and have players play them in real-time.[2]

8.2.1 Mechanics specification language

[2]The system is named BIPED because it has two legs: human and machine playtesting.

```
agent(pc).
agent(thief).
item(loose_coins).
item(assorted_gems).

game_state(has(A,I)) :- agent(A), item(I).
game_event(drop(A,I)) :- agent(A), item(I).
terminates(drop(A,I),has(A,I)) :- agent(A), item(I).
possible(drop(A,I)) :-
   holds(has(A,I)), agent(A), item(I).
```

Figure 23: Snippet of a prototype's mechanics, defining part of an inventory system. The last rule, for example, says: the game event that an agent drops an item is only possible if the agent has the item, for any agent or item.

Game mechanics are defined in a subset of Prolog, with event-calculus semantics. An example defining an inventory mechanic is shown in Figure 23. The designer can use logical predicates and constants to specify the entities in the game world and their properties. In this example, there are two agents, the player character and a thief; as well as two items, a bunch of loose coins and set of assorted gems. Entire levels can be described in a similar manner, specifying, for example, the rooms of a dungeon and hallways between them.

A special set of predicates is recognized by the language, corresponding to the event-calculus primitives. These can be used to specify the dynamic parts of the game's mechanics, organized around *game state* and *game events*. The game state is a set of properties that vary over time, under the influence of game events. The game engine and analysis engine are implemented so as to provide a common semantics for these predicates, ensuring what is possible in the human-playable version is possible in the logical version and vice versa. Returning to the example in Figure 23, the *has* game state specifies, for any agent and item, whether the agent has that item. The next rule specifies a *drop* event (for any agent and item). The following rule gives the event's effect: it terminates the *has* state (sets it to false). The final rule specifies that a *drop* event is only possible when the *has* state *holds* (is true). Similar, but not shown in this example, are *initiates*, which is analogous to *terminates* but specifies that an event sets an element of game state to true; and *conflicts*,

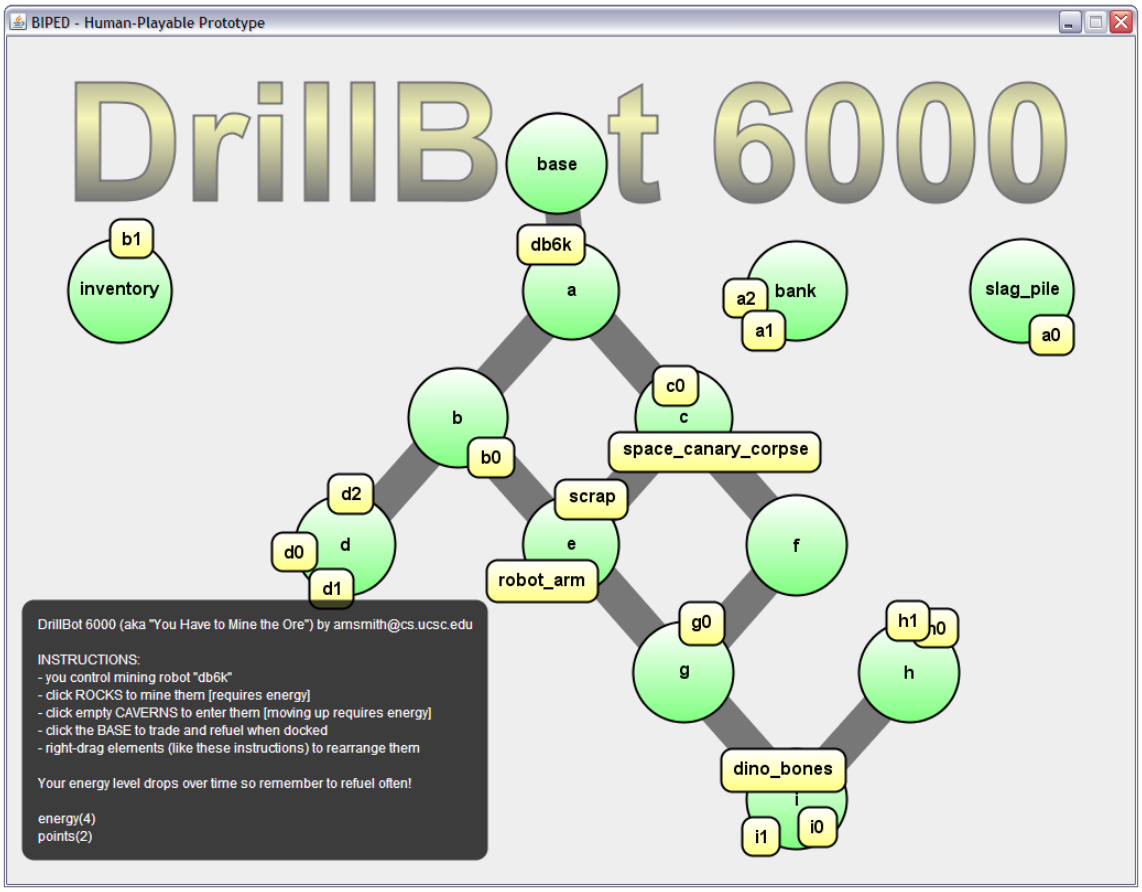


Figure 24: Human-playable prototype of *Drillbot 6000*.

which specifies restrictions on when events can happen simultaneously. The *initially* predicate can also be used to specify the elements of game state that hold at the beginning of the game.

Notice that unlike the standard versions of the event-calculus predicates, there is no explicit *time* variable in these statements. In BIPED rules are implicitly quantified over $\forall T$. This excludes some possible rules one might want to write, but suffices for most purposes, and is much more concise and less error-prone.

8.2.2 Interface elements

To complete the prototype, a set of interactive visual elements are available to specify

```
ui_title('My adventure game').
ui_space(R) :- room(R).
ui_token(I) :- item(I).
ui_triggers(ui_click_space(R), move_to(R)) :- room(R).
ui_triggers(ui_click_token(I), grab(I)) :- item(I).
```

Figure 25: Bindings from UI elements to a game world.

a game's playable representation. Based on the kinds of representations often used in simple physical prototypes, the main interface elements are clickable tokens and spaces, with optional connecting lines between spaces. Figure 24 shows an example of the interface presented to the player. Figure 25 gives a simple example of setting up interface elements and connecting them to a game world. In this code example, there is a visual board space for every room in the game world (as well as a token for every item). Clicking on a board space that represents a room is set to trigger a *move_to* event in the game world for that room (similarly for tokens and grabbing of items).

As with actual physical prototypes, there are many ways of using the visual elements to represent the state of the game world, and mappings need not all be as direct as in this example. Spaces can instead be used to represent inventory, with tokens on the space representing items in the inventory; or they can be buttons, with clicks causing a state change that may be reflected in a different part of the visual representation; or space and token combinations can represent turns or phases of the game (just as a physical dealer token is passed around a poker table). Similarly, connecting lines need not only represent how tokens can move between spaces; they might represent other relationships between entities represented by the spaces, or elements on which a button would operate when pressed. As a result of this flexibility, the designer does need to recall that there is no automatic meaning of the UI elements in terms of the game-world state: instead, it is their responsibility to give the representation elements game-relevant semantics.

In addition to the visible elements, the event system is a major representational tool. Time in the game world logically stands still until an event happens. The only sources

of events are mouse interactions and expiration of timers. The effect of a living game world can be achieved using a ticker to create a regular stream of tick events, which can trigger interesting game-world events. The *ui_triggers* predicate defines these mappings from mouse-interaction and timer events to game-world events; it checks to ensure that it only triggers game-world events if they are *possible* at the time.

My collaborators and I have been able to mock up a wide range of physical prototypes using this set of elements, and have avoided introducing a larger range of special-purpose elements in order to keep the prototypes simple and easy to modify. We have, however, included a number of aesthetic elements that do not directly represent game state, such as instructions, title, victory/loss animations, and background music, that can be used to provide an interpretive framework for the human play testers. Aesthetics of even early-stage prototypes can set the mood for the game and impact the subjective play experience, even if they are not the main focus [110]. In particular, end conditions that support a subjective distinction between good and bad outcomes are a key feature of games [47].

8.2.3 Supporting play testing

To play-test games created using BIPED, the designer may begin with either machine or human testing; each process informs the other, so alternating between them may be useful.

Human play testing often begins with self-testing. The designer loads a game definition that she has created into the game engine, and is presented with an initial playable game. Even before mechanics are fully specified, it is often gratifying to see on-screen a level previously only described in the abstract. As the mechanics of a paper prototype are added to the first computational prototype, many design decisions have to be made to create the desired rigid rule system. A lightweight cycle of revision followed by self-testing can allow the designer to quickly flesh out these rules while playing the game themselves, giving a first glimpse at the gameplay possibilities afforded by their prototype.

When testing with others, it is important to formalize any parts of the game that the

```
happens(fires_at(jack,right),0).

display_to(time(1),jill,health(2),enemies_at(right)).
display_to(time(1),jill,health(2),self_at(left)).
display_to(time(1),jack,health(1),enemies_at(left)).
display_to(time(1),jack,health(1),self_at(right)).

happens(fires_at(jill,right),1).
happens(frags(jill,jack),1).

display_to(time(2),jill,health(2),enemies_at(right)).
display_to(time(2),jill,health(2),self_at(left)).
```

Figure 26: Partial trace from a multiplayer shooter game prototype, illustrating events and game state over time.

designer may have been keeping in their head during self-testing. By utilizing timers, on-screen instructions, and background music, the designer can externalize the desired dynamics and mood of the game (*e.g.* fast-paced and frenzied). Human play testing is best mediated by the designer, who can then observe player engagement and hesitation, as well as make verbal clarifications. Because playable prototypes from BIPED are user-tweakable, standalone prototypes, however, they can be sent to and played by remote individuals as well (unlike physical prototypes or computational prototypes with extensive dependencies).

A designer can specify scenarios and conditions, and the analysis engine will provide him or her with gameplay traces (if any exist) that start from those scenarios and meet those conditions. Figure 26 shows a short excerpt of a trace from a multiplayer shooter prototype, in which an agent, interestingly, inflicts damage on himself. To look for an exploit, the designer might ask for traces starting in a particularly tricky scenario, which end in victory only a few timesteps later. If any traces exist, the designer has a specific example of the behavior to forbid in the game rules; if not, the engine has proved that no such exploit is possible (which would be difficult to determine with certainty using only human play testing). In cases where there are many uninteresting traces, the designer can restrict the conditions in order to "zoom in" on more plausible gameplay. Alternatively, the designer can run experiments in which the scenario and conditions are held constant, and some element of the rules is changed. This gives the designer backtalk (in Schön's sense)

regarding a possible rule change, rather than more detailed inspection of a particular rule set.

8.2.4 Implementation

The implementation is primarily split between two engines. The game engine, supporting human-playable prototypes, was implemented in the Scala programming language, which targets the Java Virtual Machine for easy redistribution of games. The rules of a particular game, implemented in Prolog, are interpreted using jTrolog,[3] a Prolog engine for Java, so game prototypes can be modified by end users without recompilation. The game engine executes the game rules by advancing time in response to UI events, querying Prolog to find whether visual elements have changed (*e.g.* whether tokens have moved), and which elements of game state hold on the next time step.

The analysis engine, supporting the construction of complete formal rule systems, was implemented in Lparse/Smodels,[4] an answer-set-programming toolchain. A compiler written in Prolog translates the game-prototype definitions into answer-set programs usable by Lparse.[5] Additionally, a small Lparse "game engine" ties the rest of the game prototype (excluding UI) into the event-calculus semantics. The analysis engine is also responsible for checking that a game prototype is complete (defining a minimal set of required predicates) and checking any additional designer-specified static properties (*e.g.* there are no rooms in the level description without an adjoining hallway).

An interesting result of this translation is that when the analysis engine is looking for traces, it treats time as a symbolic constraint rather than simulating game worlds forward in time. In this way, it is as easy to put constraints on initial conditions as it is on end conditions, or on any point in between. In the game engine behind human-playable prototypes, logical time, while discrete, behaves much more intuitively, advancing step by step

[3] `https://jtrolog.dev.java.net/`

[4] `http://www.tcs.hut.fi/Software/smodels/`

[5] Since Lparse's input language and semantics both derive from Prolog, this is not very difficult to do, at least in our restricted setting.

in response to timers and human interaction (effectively finding a single trace).

8.2.5 Example prototype

To exercise play testing with BIPED, we created *DrillBot 6000* (previously illustrated in Figure 24). In this game, the player moves a mining robot through underground caverns, drilling out rocks of mixed value, while managing energy usage by periodically returning to the surface to refuel and trade items. This game was designed as if to be an early prototype version of the popular Flash game *Motherload* from XGen Studios.[6] The prototype version focuses on the core mechanics: moving around underground, mining, and refueling (whereas *Motherload* includes shopping for upgrades and story elements).

8.2.5.1 Game mechanics

To describe the mechanics of the *DrillBot 6000* game world, the game definition asserts that *position* and *energy* are elements of game state (that apply to the one robot), and that subterranean rocks can be *present* in a cavern, *bagged* in the robot's inventory, or possibly *banked* after trading. In terms of game events, we allow mining of rocks, moving up or down between caverns, refueling, trading rocks, and spontaneous energy drain. The game prototype also defines the consequences of these events, and when they are possible. For example, the *mine* event for a rock *initiates* the *bagged* state for that rock, *terminates* its *present* state, and drains a unit of *energy*. This *mine* event is *possible* if: the rock is *present*, the location of the rock is reachable from the robot's current *position*, and the robot has *energy*. The rigid rules for other game events are defined similarly. Finally, the definition asserts initial conditions: the robot starts fully energized at the base, and all rocks are present.

[6] `http://www.xgenstudios.com/play/motherload`

8.2.5.2 UI bindings

While the game mechanics described above are enough to allow machine play testing, for human testing we needed to expose the abstract game world to the player. Caverns are mapped to board spaces, and the up/down links between caverns are visualized with connecting lines. Individual tokens represent the minable rocks, and a special token represents the robot itself. The UI event of clicking a rock's token is bound to the *mine* event for the corresponding rock. Likewise, clicking a cavern's space is bound to either a *move_up* or *move_down* event to that cavern. These bindings are expressed concisely without need to reiterate the necessary conditions (*e.g.* the proper *move_up* or *move_down* event is selected by virtue of the game-world definition of when these events are *possible*).

We bound the position of rock tokens to cavern spaces using the abstract level definition and the *present* status of the rock to select a space. When rocks are not present, the player should have a way of knowing the rock's *bagged* or *banked* status. The additional spaces called *inventory*, *bank*, and *slag_pile* are used as the location for rock tokens that are no longer present but have differing bagged or banked states (valueless rocks cannot be banked, and their tokens are sent flying to the slag pile with a quick animation). Spaces themselves are anchored to the board with an optional predicate; these positions were consciously set to portray the directionality of links between caverns.

To give our prototype an element of the time pressure present in *Motherload*, there is a ticker for which the tick event is bound to the game world's *drain* event, draining a unit of the robot's energy. Thus, robot energy drains at a regular pace, but faster when the player actively triggers game-world events that consume energy. Energy is replenished by clicking on the base's space, which triggers the game-world *refuel* and *trade* events simultaneously.

A game definition also specifies several non-interactive elements. A large title and background music set the tone for a lively real-time mining game. On-screen, written instructions lay out both the premise of the game and explain how the player can use the mouse to take actions in the world of rocks, caverns, and a robot. Some elements of game

state not mapped to spaces and tokens are depicted textually, in this case energy level and score. Finally, when the game determines that no additional actions are possible, a few tokens fly onto the screen to announce the game's outcome.

8.2.5.3 Human play testing

As suggested by Fullerton [30, p. 252], since we were testing the foundations and structure of the game, we primarily tested *DrillBot 6000* by self-testing and testing with confidants. Self-testing revealed that our early iterations had allowed unintended gameplay; for example, you could mine a rock at arbitrary distances. Additionally, we found the first version of the game too simple, and decided to add several additional rocks and caverns. When testing with others, one tester initially felt pressured by the speed of the automatic energy drain. While we could have adjusted the speed of energy drain or the maximum energy level, at this stage of the prototype we were interested in more foundational questions, rather than game balancing. To get feedback on the game's other mechanics, we showed the tester how to convert the game to a turn-based one by removing the ticker. All three testers claimed to enjoy the game, and could reach rocks at the deeper levels after some practice. Interestingly, no testers related to the game as a puzzle or path-planning challenge, even in turn-based mode; all focused instead on the action aspect of the game motivated by the continuously draining energy.

8.2.5.4 Machine play testing

While human play testing validated that the game's basic concept was interesting, BIPED allowed us to turn to machine play testing to ask more detailed questions that would have been tedious or impossible to test with human testers. Because players focused on improving their speed, in machine play testing we decided to look at the limit case, corresponding to a player that could execute several actions without incurring any time-based energy drain (which was possible but difficult with our timer settings). This would allow us to focus on the energy cost of mining and moving as the limiting factors, effectively discovering speed

runs.

In one experiment, we looked for gameplay traces that maximized the number of treasures the robot could bank by the end of a fixed number of timepoints. In 15 timepoints, we found that the ideal player could bank up to five valuable rocks. Then we asked about a player who would never refuel, wondering if this would place some rocks out of reach. Over this game length we found that the ideal player could still bank five valuable rocks, indicating that refueling is needed relatively infrequently. This was interesting, because our human testers refuelled quite often, as a hedge against energy drain, indicating that they were much more cautious with managing energy than strictly necessary.

In the setup for this experiment, when looking at general gameplay traces, we found undesirable traces in which several rocks were mined simultaneously. This revealed that we had not completely specified when game actions should conflict. That issue was not revealed in human testing, because the UI bindings happened to make it impossible to mine several rocks at the same time (which would not be the case if mining had been tied to clicks on spaces instead of on tokens). Finding that this design choice was incompletely specified both forced us to think about what mechanics we actually did want, and avoided the persistence of a UI-masked bug that could reappear in later prototypes.

In another experiment, we looked at a few properties of the level's design. One question involved asking how many caverns a player could explore in a given interval (where the robot starts and ends in the base). This query is particularly tedious to answer through human play testing, since it would require trying a vast number of possibilities. It also goes beyond simple graph search, because it takes into account the effects of time-varying presence of rocks due to mining. In 15 timepoints, a player can explore up to eight caverns before returning to the base. Making what we thought would be a radical change to the level's design, we added a direct link from the base down to the deepest cavern in the level. Running queries for reachable caverns and banked rocks again, we were surprised to find this change made no difference in the properties of the optimal play-through.

8.3 Formalized mechanics conclusions

Automated mechanics analysis can elucidate aspects of early-stage game designs, though it remains at a stage where many questions are unanswered. My exploratory interviews with designers turned up significant interest and some successful applications, but also considerable mismatches between expectations, in one case quite strong ones. A direction suggested by the mismatches is to prioritize visual sketching of games' logic over formal inference of it, although these may not be either-or choices. The BIPED prototyping system demonstrated that an EC-based formalization can be played in addition to queried, efficiently enough to serve as the basis for a real-time game. It also demonstrated the complementary nature of human and machine prototyping in answering different kinds of design questions, and more powerfully when combined than either separately.

CHAPTER IX

CONCLUSIONS

The goal of this thesis has been to create knowledge representation and reasoning strategies that can support the game-design process. Understanding the complex way in which gameplay emerges from game mechanics is a key challenge of game design, and unlike in more established design domains, there are few tools to help designers with that challenge—the norm is a mixture of custom programming, modeling in general tools such as Excel, prototyping with paper and pencil, and extensive playtesting. This thesis pursues a strategy of knowledge-representation-first investigation of game-design assistance: analyze a domain, formalize knowledge in that domain useful for building design-assistance tools, and then build tools on top of that knowledge representation.

The knowledge-representation-first strategy starts by looking at what, in the large and often poorly understood space of game design, can be nailed down concretely into a machine-readable formalism. And then, it asks: what can automated analyses using this formalism do to support designers? Specifically, what can they do to help a designer cross the mechanics/gameplay divide, that is, to understand the impact of mechanics design decisions on the dynamics and aesthetics of gameplay experiences?

9.1 Systems built and research contributions

In this dissertation, I have built formal representations and tools on top of those representations in three domains: in interactive narrative, where the key problem is managing nonlinear story progression; automated game generation, where the question is how to craft playable and interpretable interactive rule systems; and early-stage game prototyping, where the key problem is how to understand the dynamics emerging from a game's mechanics.

9.1.1 Interactive narrative

Of the three domains investigated here, interactive narrative is the one with the most existing research, including a proposed method for bridging the gap between nonlinear story progression and authorial goals, in the face of uncertainty about what the player will do during gameplay. In his dissertation, Peter Weyhrauch [141] proposed an annotated *story graph* as a formalism for tracking progression in a branching interactive story. This is then combined with a *story quality function* that encodes an author's judgment of completed experiences' desirability, built through a weighted sum of several story-independent quality features (such as tension arc). With these two components, a runtime *search-based drama manager* (SBDM) searches online for interventions to make into the story world that are predicted to maximize the final story quality.

Weyhrauch experimented with SBDM on a demonstration story he wrote, *Tea for Three*, but it has not (until now) been applied to a significant "real-world" interactive story. I applied it to the award-winning *Anchorhead* by Michael S. Gentry, in order to investigate what (if anything) would have to be done to use SBDM in this scaled-up context, applied to a story written without the author having any knowledge of this drama-management formalism. The first contribution necessary to do so was that I significantly expanded the representational power of Weyhrauch's story formalism. Even in abstracted story-graph form, modeling *Anchorhead* required new concepts such as the ability to track multiple endings and subplots, and context for events—both physical contexts such as locations, and story contexts such as whether a conversation is in progress. Furthermore the notion of "experience quality" for interactive stories needed to be extended to handle the dynamic tradeoffs involved here, so I developed new features to be used in experience-quality functions, along with a detailed analysis focusing on the interaction between optimizing such features and the player's interactive agency.

Having extended the formalism to allow *Anchorhead* to be reasonably represented as a story graph, and authorial preferences to be encoded in an experience-quality function,

I then returned to replicating the technical experiments that Weyhrauch had performed. Here, I found that the original SBDM sampling search did not scale to *Anchorhead*, which was much larger and with a more complex (harder to approximate by random sampling) search space than that of *Tea for Three*. To address this, I separated the declarative aspects of the formalism from the optimization strategy, generalizing search-based drama management (SBDM) into declarative optimization-based drama management (DODM), an optimization problem that can be solved using a variety of online or offline strategies. I further developed this into a probabilistic version targeting story distributions, based on the novel machinery of targeted trajectory distribution Markov decision processes (TTD-MDPs).

9.1.2 Automated game generation

The second case study, on automated game generation, tackles the complexities of automatically designing videogames. This problem is considerably broader than generating game *rules*, since a successful videogame has a wide variety of elements, leading to the final experience. In order to better decompose the design problem, I identified four broad classes of such elements, to my knowledge the first analysis to analyze videogame design as a multi-knowledge-base problem in this manner; these elements are the *abstract mechanics* of the game (what corresponds most closely to traditional game rules), the *concrete representations* the game uses to communicate game state and affordances to the player (such as audiovisual elements), the *input mappings* by which the player interacts with the game (differentiating it from a non-interactive simulation or screensaver), and the *thematic mappings* by which the game gives semiotic content to the game world (making a game be set in a castle, or about city-building).

In order to start from the part of this decomposition that in a sense seems most difficult to automate, I focus on the thematic mappings, specifically the problem of automatically generating games that are "about" something. In the case where the other elements are

held fixed, this can be viewed as automatically "reskinning" games: taking a fixed underlying game and changing it from *e.g.* a car-chase game into a road-runner/coyote game, by changing the thematic mappings.

To capture this "aboutness" of games, I develop a formalization of thematic mappings based on constraint graphs defined using the ConceptNet and WordNet databases of commonsense knowledge. An author defines a *game space* of sensible reskinnings; for example, in a chase game, specifying the expected relationships between the thing chasing and the thing being chased. Authors build these constraint graphs using an interactive editor, which allows the game spaces to be iteratively refined. Once such a game space is formalized, it can be used either at design time as a brainstorming tool, or at runtime as a game that spits out reskinned versions on demand.

9.1.3 Mechanics-oriented prototype support

The third case study tackled computational support for mechanics-oriented videogame prototyping. Mechanics-oriented prototypes focus on the abstract rule system of a videogame, and are often used by game designers in early stages of development to test out ideas before doing a full implementation. Traditionally they may be done in anything from paper form (moving tokens around on a table) to simple computational tools like Excel spreadsheets. Computational support for prototypes aims to make them more informative, loosely inspired by the way CAD systems do something similar for architecture, turning traditional drawing of blueprints into something more interactive and informative.

The main technical underpinning of this work is an elaboration-tolerant formalization of abstract game mechanics in the event calculus, a temporal logic. Unlike pure specification languages, the goal here is iterative prototyping to support a designer's exploration of a set of game mechanics' emergent dynamics. Through automated analysis, designers can better understand the gameplay possibility space their game prototype implicitly defines. I developed a characterization of the mechanics/dynamics gap as one between an

intensionally defined game artifact (the game code/mechanics/etc.), and the space of gameplay possibilities that the artifact supports or forecloses, an extensional set of *gameplay traces* defining the game's playspace. Unlike in formal verification, the goal here is not to *verify* that the playspace meets criteria, but rather to *understand* and *explore* it. Through automated derivation of traces with particular properties, the designer can explore their game's possibility space, zooming in on expected or unexpected possibilities, and iteratively tweaking the game in accordance with what they find. I term this style of working *analytical metrics* or *design-design metrics*, by analogy to the already widely used empirical metrics, which are derived by having players playtest an instrumented version of a game, and logging their plays.

To understand how (or whether) designers would find such a tool useful, I performed a "reverse Wizard-of-Oz" study in which I served as the human natural-language interface to this functionality, formalizing their designs-in-progress and talking through queries they had about them. I then built[1] an interactive prototyping system, BIPED, which supports abstract prototyping of games in this formalism, with a graphical interface in the style of token-based paper prototypes. Using this system, game prototypes can both be played (in real-time) and queried using the same game description. This enables designers to combine insights from human playtesting and automated analysis, in the same setting and using the same elements—for example trends observed in empirical metrics can be juxtaposed with possibility spaces outlined by analytical metrics.

[1] In collaboration with Adam Smith.

9.2 Broader impacts

This dissertation's research has been aimed at making progress towards a much larger research agenda (which I believe it has helped initiate), of formalizing game design knowledge in a manner that is susceptible to automated reasoning, which can be applied in design-assistance systems, semi-automated design systems, or even perhaps fully automated design systems. One way of gauging whether it has had the desired effect in helping to initiate and push forward this agenda is to look at what other research it has enabled and influenced, especially in those areas which were nascent at the time I embarked on this project.[2]

9.2.1 Impact on automated game design

A significant area where this work has had influence is in the field of automated videogame design (also known as videogame generation), which has come into existence roughly alongside the timespan of this work. To my knowledge, the analysis in Chapter 2, Section 2.2 considering the videogame design problem from the perspective of AI, was—when first published in 2007 [74]—the first to look at it in generality, treating videogames as a complex designed artifact composed of a number of design domains (audiovisual design, mechanics design, input-mapping design, and thematic design), rather than treating it as synonymous with the construction of sets of *game rules*. And the thematically focused game generator of Chapter 5 was the second system to focus on automated design of an aspect of videogames other than rules, preceded only by the rather interesting system EGGG (the Extensible Graphical Game Generator) [85], which takes in rules for *e.g.* a card game, and automatically generates a suitable computer interface.

This work has become fairly highly cited. A particularly close bit of follow-on work to my *WarioWare*-generation work has been pursued by some colleagues, working on a

[2]Due to the fact that most of my thesis work was completed and published as papers in 2004–2011, but this final dissertation itself is only being published now in 2015, this section has a bit of a time-travel aspect to it, as I look at prior work that this dissertation has already influenced.

generative formalization of meaning in videogames. Starting with a somewhat more complex game, Treanor *et al.* [135] find dozens of different readings of the classic arcade game *Kaboom!*, and then build a formalization of how to layer thematic mappings onto specific readings. This developed into the *Game-O-Matic* system, in which users provide a small description of what they want a game to be about (*e.g.*, professors chasing grants while avoiding committee meetings), and then skins this thematic mapping onto one of its analyzed sets of game mechanics [133, 136]. Compared to the work here, *Game-O-Matic* focuses on taking a *specific* desired theme and automatically applying it to many game mechanics; whereas the system I built focuses on taking games from a small set of stock mechanics, and automatically generating a set of plausible themings of those games.

An important collection of subsequent work is found in the game-generation project of Michael Cook and Simon Colton (and collaborators), which is also characterized by its focus on multi-faceted treatment of videogame generation. While the work in my dissertation here generates games by holding all but one facet fixed and generating the remaining one, they focus on simultaneous interplay of design aspects, such as aesthetics, mechanics, and theme [14, 15, 16]. Finally, another large influence on game-generation can be found in tools that generate games using constraints over formal models of mechanics—better discussed in the following section.

9.2.2 Impact on game-dynamics formalization

Chapters 6 and 7 investigated formalizing dynamics in order to support mechanized negotiation of the artifact/play-trace boundary—or put differently, the mechanics/dynamics boundary in the mechanics/dynamics/aesthetics (MDA) model of game design. This strategy of automatically analyzing game dynamics has since become a common theme of research on videogame design based on methodologies from artificial intelligence, an area I helped to jump-start by co-organizing a AAAI workshop series on the role of AI in the game-design process [81, 82].

The most direct influence can be seen in the work of Adam Smith, who adopts a similar technical formalism for game mechanics, based on the event calculus and answer-set programming. But rather than using it primarily for design support, in his work this approach enables a different goal, of building mechanized models of the entire design process [113]. In addition to building tools that human designers use to better undertand and improve games, he models the next-order theory of how they would do so: how they would query games, look at playspaces, and revise their designs accordingly. These may be caricatured models of a game designer, especially initially [117], but even caricatured models can help elicidate a theory of game design, and the effects of various game-design strategies.

A second area in which this work has had influence is in the recently emerged technique of *constraint-based procedural content generation* (constraint-based PCG). Much research in PCG can be characterized as *search-based* [132], in which a designer specifies a fitness function, and a heuristic-search algorithm (such as an evolutionary algorithm) optimizes content for the fitness function. Questions of game dynamics in this approach are most often handled by simulation: the fitness function will include a simulation component that plays the game 100 or 1000 times or however many there is computational time for, and assigns a score based on a function of the simulation outcomes (for example, game balance or challenge). An alternative, if we have a formal model of game mechanics from which dynamics properties can be automatically derived, is to directly specify constraints over desired dynamics, and then generate content that meets those constraints [115, 116]. This allows certain kinds of properties, especially "hard" requirements, to be specified much more naturally. Better understanding the relative strengths and weaknesses of search-based and constraint-based PCG is currently an active area of research (for some speculation, see [130]).

A third area on which this dissertation's work on formalizing game dynamics has had influence is in understanding games via characterization of the space of possible play traces.

Work by Tremblay *et al.* [137] and Jaffe *et al.* [42, 43] is most directly in this vein, extending it in interesting ways. Tremblay *et al.* generate maps of player risk in stealth games by analytically characterizing the state space (what I call analytical metrics); this allows a designer to make decisions about things such as placement of safe places for the player to pause. Jaffe *et al.* characterize games by investigating the results of having various simple player models play it, a similar strategy to that discussed in Section 6.4.6; but they extend the generally qualitative focus of my work by focusing instead on quantitative study of game balance. Finally, several researchers have investigated play traces themselves as a unit, looking into whether useful generic tools can be built to manipulate and extract information from them [87, 114].

9.3 Future work

AI-based videogame design assistance, rooted in knowledge representation and reasoning, has many directions in which it can go. I mention here three areas I think particularly promising.

9.3.1 Modularity and reusability

The first direction is to develop more modular and reusable game formalizations. Games larger than those I consider here are often built up of heterogenous parts and subsystems, with their own logics, vocabularies, design norms, and points of connection between the parts: dialog systems, combat systems, inventory systems, in-game economies, and real-time movement logics, to name a few. I sidestep this by considering certain restricted aspects of games in isolation: the plot structure of interactive narrative, the thematic references of game "skins", and the mechanics/dynamics insight of paper-prototyping style game sketches. Another way of partly sidestepping the problem is to "flatten" a game's subsystems to one aspect that is considered at a time. For example, Dormans [21, 22] models games as resource flows between nodes based on a Petri-net formalism. This allows a certain modularity to emerge in the graphical connections, as game subsystems can be

connected to each other, disconnected, and rearranged—but only as long as all everything is discussed in the unifying language of resource flows.

Tackling complex game-design problems will require formalisms that are more heterogenous and encapsulate more design-level information about *why* and *how* things are connected. For example, Goel, Rugaber and collaborators consider the problem of designing game-playing agents for a particular game's mechanics, and then adapting those agents as mechanics change without starting over. This requires representations of both the internal structure of the agent and the agent's relationship to the game mechanics, *i.e.* why it is doing certain things and how those decisions are dependent on mechanics [37, 45, 46, 103].

The concept of modular game critics [86] also pushes in this direction, explicitly reifying parts of the game-design problem as having their own norms; as does the work of Smith [113] in connecting representation of game elements to mechanization of the design process.

9.3.2 Connection with game-studies research

A second direction for future work is to better connect computational models of games with game-studies research. Formalizing games for computational purposes inevitably produces, at least implicitly, a formalist theory of some aspect of games or game design. The more humanities-oriented academic field of game studies also sometimes produces such theories, though usually not formal enough for our direct usage ("formalist" in a humanities context does not necessarily mean *implementable on a computer*).

A closer connection could proceed from either direction. Suitable concepts from game studies could be adopted; for example, theories such as operational logics [63] that are unformalized but have implications for representational structuring may provide a useful basis for formalisms. In addition, taxonomies, pattern studies [7], and ontologies [146] from game studies, currently described mainly in natural language might be adaptable into computational knowledge representations.

In the other direction, computational models can be developed more explicitly into game-studies theories. At least implicitly, formalizing games requires (and constitutes) analyzing them. Generative formalisms can then be seen as a way of investigating the fecundity and fidelity of that analysis. Although it is not positioned in that way by its creator, the *Ludi* board-game-generation system [10] can be seen in these terms, as a formalized theory of game design in the space of a certain style of board games ("connection games"), embodied in a computational model. Procedural-content generation systems are another emerging example where systems implicitly and sometimes explicitly embody design and structural theories, for example of platformer-game design [121].

9.3.3 Design assistance and HCI

A final wide-open area is the front end of design-assistance tools for game design [71]. What should usable tools look like? How should they be integrated with designers' existing tooling and workflows? Is "tool" even the best conceptualization of that integration, or might we consider other metaphors, such as expert, co-designer, or intelligent material [48]? How should visualizations and other information be presented intelligibly? In short, what should the HCI story of design assistance look like?

This dissertation has investigated the back end of knowledge representation for game-design assistance, essentially the functional requirements of what models can represent and do. It has not ignored designers' processes in doing so, engaging in the third case study in a requirements analysis through a "reverse Wizard-of-Oz" study design to elicit what functionality designers might want out of a design system. But this does not tell us much about what the front end should look like, and this dissertation does not purport to answer the non-trivial HCI questions involved.

A direction such research might take as regards the knowledge-representation portion includes more clearly distinguishing knowledge representations for human usage, and those

for semi-automated or fully-automated design. So far I have proceeded under the assumption that it's desirable to have "multi-use" representations that can be used for simple design feedback or partial automation as desired. But to truly build a "CAD for game design" there will almost certainly be more special-case knowledge needed. Examples of representational areas to investigate in a front-end-focused design-assistance tool include the salience of information, the purpose of design decisions, and the coexistence of multiple partial designs.

REFERENCES

[1] Adams, E. and Dormans, J., *Game Mechanics: Advanced Game Design*. New Riders, 2012.

[2] Alpert, S. R., "Graceful interaction with graphical constraints," *IEEE Computer Graphics and Applications*, vol. 13, no. 2, pp. 82–91, 1993.

[3] Andersen, E., Liu, Y.-E., Apter, E., Boucher-Genesse, F., and Popović, Z., "Gameplay analysis through state projection," in *Proceedings of the 5th International Conference on the Foundations of Digital Games (FDG)*, 2010.

[4] Bates, J., "Virtual reality, art, and entertainment," *Presence: The Journal of Teleoperators and Virtual Environments*, vol. 2, no. 1, pp. 133–138, 1992.

[5] Bentley, P. J., ed., *Evolutionary Design by Computers*. Morgan Kaufmann, 1999.

[6] Bhat, S., Roberts, D. L., Nelson, M. J., Isbell Jr., C. L., and Mateas, M., "A globally optimal algorithm for TTD-MDPs," in *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 1196–1203, 2007.

[7] Björk, S. and Holopainen, J., *Patterns in Game Design*. Cengage Learning, 2005.

[8] Bogost, I., *Persuasive Games: The Expressive Power of Videogames*. MIT Press, 2007.

[9] Bogost, I., Ferrari, S., and Schweizer, B., *Newsgames: Journalism at Play*. MIT Press, 2010.

[10] Browne, C., *Evolutionary Game Design*. Springer, 2011.

[11] Chandrasekaran, B., "Design problem solving: A task analysis," *AI Magazine*, vol. 11, no. 4, pp. 59–71, 1990.

[12] Chen, S., Nelson, M. J., and Mateas, M., "Evaluating the authorial leverage of drama management," in *Proceedings of the 5th Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*, pp. 136–141, 2009.

[13] Cohen, P. R., Davis, J. A., and Warwick, J. L., "Dynamic visualization of battle simulations," Tech. Rep. 00-16, University of Massachusetts Computer Science Department, 2000.

[14] Cook, M. and Colton, S., "From mechanics to meaning and back again: Exploring techniques for the contextualisation of code," in *Proceedings of the AIIDE 2013 Workshop on on Artificial Intelligence and Game Aesthetics*, pp. 2–6, 2013.

[15] Cook, M., Colton, S., and Gow, J., "Initial results from co-operative co-evolution for automated platformer design," in *Proceedings of the 15th Conference on Applications of Evolutionary Computation*, pp. 194–203, 2012.

[16] Cook, M., Colton, S., and Pease, A., "Aesthetic considerations for automated platformer design," in *Proceedings of the 8th Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*, pp. 124–129, 2012.

[17] Coons, S. A., "Surfaces for computer-aided design of space forms," Tech. Rep. TR-41, Massachusetts Institute of Technology, 1967.

[18] Cross, N., *The Automated Architect: Human and Machine Roles in Design*. Pion Limited, 1977.

[19] Cross, N., *Designerly Ways of Knowing*. Springer, 2006.

[20] Davis, R., Shrobe, H., and Szolovits, P., "What is a knowledge representation?," *AI Magazine*, vol. 14, no. 1, pp. 17–33, 1993.

[21] Dormans, J., "Machinations: Elemental feedback patterns for game design," in *GAMEON-NA 2009: 5th International North American Conference on Intelligent Games and Simulation*, pp. 33–40, 2009.

[22] Dormans, J., "Simulating mechanics to study emergence in games," in *Proceedings of the AIIDE 2011 Workshop on Artificial Intelligence in the Game Design Process*, pp. 2–7, 2011.

[23] Drachen, A. and Canossa, A., "Analyzing spatial user behavior in computer games using geographic information systems," in *Proceedings of the 13th International MindTrek Conference*, pp. 182–189, 2009.

[24] Drachen, A., Canossa, A., and Yannakakis, G. N., "Player modeling using self-organization in Tomb Raider: Underworld," in *Proceedings of the 5th International Conference on Computational Intelligence and Games (CIG)*, pp. 1–8, 2009.

[25] Fellbaum, C., ed., *WordNet: An Electronic Lexical Database*. MIT Press, 1998.

[26] Fischer, G., "Seeding, evolutionary growth and reseeding: Constructing, capturing, and evolving knowledge in domain-oriented design environments," *Automated Software Engineering*, vol. 5, no. 4, pp. 447–464, 1998.

[27] Fischer, G., McCall, R., and Morch, A., "Design environments for constructive and argumentative design," in *Proceedings of the 1989 Conference on Human Factors in Computing Systems (CHI)*, pp. 269–275, 1989.

[28] Fischer, G., "Domain-oriented design environments," *Automated Software Engineering*, vol. 1, no. 2, pp. 177–203, 1994.

[29] Frasca, G., *Play the Message: Play, Game and Videogame Rhetoric*. PhD thesis, IT University of Copenhagen, 2007.

[30] Fullerton, T., *Game Design Workshop*. Morgan Kaufmann, 2nd ed., 2008.

[31] Gero, J. S., "An overview of knowledge engineering and its relevance to CAAD," in *Proceedings of CAAD Futures 1985*, pp. 107–119, 1985.

[32] Giaccardi, E. and Fischer, G., "Creativity and evolution: A metadesign perspective," *Digital Creativity*, vol. 19, no. 1, pp. 19–32, 2008.

[33] Giannatos, S., Nelson, M. J., Cheong, Y.-G., and Yannakakis, G. N., "Suggesting new plot elements for an interactive story," in *Proceedings of the 4th Workshop on Intelligent Narrative Technologies (INT)*, pp. 25–30, 2011.

[34] Gingold, C., "What WarioWare can teach us about game design," *Game Studies*, vol. 5, no. 1, 2005. http://www.gamestudies.org/0501/gingold/.

[35] Gingold, C. and Hecker, C., "Advanced prototyping." Presentation at the 2006 Game Developer's Conference (GDC), 2006. http://chrishecker.com/Advanced_prototyping.

[36] Goel, A. K. and Chandrasekaran, B., "Functional representation of designs and redesign problem solving," in *Proceedings of the 11th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1388–1394, 1989.

[37] Goel, A. K. and Rugaber, S., "Interactive meta-reasoning: Towards a CAD-like environment for designing game-playing agents," in *Computational Creativity Research: Towards Creative Machines*, pp. 347–370, Atlantis Press, 2015.

[38] Goel, A. K. and Stroulia, E., "Functional device models and model-based diagnosis in adaptive design," *Artificial Intelligence for Engineering, Design, Analysis and Manufacturing (AI EDAM)*, vol. 10, no. 4, pp. 355–370, 1996.

[39] Hewitt, C., "The challenge of open systems," *Byte*, vol. 10, no. 4, pp. 223–242, 1985.

[40] Hom, V. and Marks, J., "Automatic design of balanced board games," in *Proceedings of the 3rd Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*, pp. 25–30, 2007.

[41] Hunicke, R., LeBlanc, M., and Zubeck, R., "MDA: A formal approach to game design and game research," in *Proceedings of the Challenges in Game AI Workshop at AAAI 2004*, 2004.

[42] Jaffe, A., *Understanding Game Balance with Quantitative Methods*. PhD thesis, University of Washington, 2013.

[43] Jaffe, A., Miller, A., Andersen, E., Liu, Y.-E., Karlin, A., and Popović, Z., "Evaluating competitive game balance with restricted play," in *Proceedings of the 8th Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*, pp. 26–31, 2012.

[44] Järvinen, A., "Gran Stylissimo: The audiovisual elements and styles in computer and video games," in *Proceedings of Computer Games and Digital Cultures Conference*, pp. 113–128, 2002.

[45] Jones, J., Parnin, C., Sinharoy, A., Rugaber, S., and Goel, A. K., "Adapting game-playing agents to game requirements," in *Proceedings of the 4th Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*, pp. 132–137, 2008.

[46] Jones, J., Parnin, C., Sinharoy, A., Rugaber, S., and Goel, A. K., "Teleological software adaptation," in *Proceedings of the 3rd IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, pp. 198–205, 2009.

[47] Juul, J., "The game, the player and the world: Looking for a heart of gameness," in *Proceedings of the 2003 Digital Games Research Association Conference (DiGRA)*, pp. 30–45, 2003.

[48] Khaled, R., Nelson, M. J., and Barr, P., "Design metaphors for procedural content generation in games," in *Proceedings of the 2013 Conference on Human Factors in Computing Systems (CHI)*, pp. 1509–1518, 2013.

[49] Kim, T.-W., Lee, J., and Palla, R., "Circumscriptive event calculus as answer set programming," in *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 823–829, 2009.

[50] Lawson, B. R., *The Language of Space*. Architectural Press, 2001.

[51] Lawson, B. R., "CAD and creativity: Does the computer really help?," *Leonardo*, vol. 35, no. 3, pp. 327–331, 2002.

[52] Lawson, B. R., "Oracles, draughtsmen, and agents: The nature of knowledge and creativity in design and the role of IT," *Automation in Construction*, vol. 14, no. 3, pp. 383–391, 2005.

[53] Lawson, B. R. and Loke, S. M., "Computers, words, and pictures," *Design Studies*, vol. 18, no. 2, pp. 171–183, 1997.

[54] Lenat, D. B., "CYC: A large-scale investment in knowledge infrastructure," *Communications of the ACM*, vol. 38, no. 11, pp. 33–38, 1995.

[55] Lieberman, H., Liu, H., Singh, P., and Barry, B., "Beating common sense into interactive applications," *AI Magazine*, vol. 25, no. 4, pp. 63–76, 2004.

[56] Liu, H. and Singh, P., "ConceptNet: A practical commonsense reasoning toolkit," *BT Technology Journal*, vol. 22, no. 4, 2004.

[57] Love, N., Hinrichs, T., Haley, D., Schkufza, E., and Genesereth, M., "General game playing: Game description language specification," Tech. Rep. LG-2006-01, Stanford Logic Group, 2006.

[58] Loyall, B., "Response to Janet Murray's 'From game-story to cyber-drama'," in *First Person: New Media as Story, Performance, and Game*, pp. 2–9, MIT Press, 2004.

[59] Magerko, B., "Story representation and interactive drama," in *Proceedings of the 1st Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*, 2005.

[60] Mark, D., *Behavioral Mathematics for Game AI*. Course Technology PTR, 2009.

[61] Mateas, M., "A neo-Aristotelian theory of interactive drama," in *Proceedings of the AAAI Spring Symposium on Artificial Intelligence and Interactive Entertainment*, pp. 56–61, 2000.

[62] Mateas, M., "Procedural literacy: Educating the new media practitioner," *On the Horizon*, vol. 13, no. 2, pp. 101–111, 2005.

[63] Mateas, M. and Wardrip-Fruin, N., "Defining operational logics," in *Proceedings of the 2009 Digital Games Research Association Conference (DiGRA)*, 2009.

[64] McCarthy, J., "Circumscription: A form of non-monotonic reasoning," *Artificial Intelligence*, vol. 13, no. 1–2, pp. 27–39, 1980.

[65] McCarthy, J., "Elaboration tolerance," in *Proceedings of the 4th Symposium on Logical Formalizations of Commonsense Reasoning*, 1998.

[66] McCarthy, J. and Hayes, P. J., "Some philosophical problems from the standpoint of artificial intelligence," *Machine Intelligence*, vol. 4, pp. 463–502, 1969.

[67] Mueller, E. T., *Natural Language Processing with ThoughtTreasure*. Signiform, 1998. `http://www.signiform.com/tt/book/`.

[68] Mueller, E. T., "Event calculus reasoning through satisfiability," *Journal of Logic and Computation*, vol. 14, no. 5, pp. 703–730, 2004.

[69] Mueller, E. T., *Commonsense Reasoning*. Morgan Kaufmann, 2006.

[70] Murray, J., *Hamlet on the Holodeck: The Future of Narrative in Cyberspace*. MIT Press, 1998.

[71] Neil, K., "Game design tools: Time to evaluate," in *Proceedings of DiGRA Nordic 2012*, 2012.

[72] Nelson, M. J., "Game metrics without players: Strategies for understanding game artifacts," in *Proceedings of the AIIDE 2011 Workshop on Artificial Intelligence in the Game Design Process*, pp. 14–18, 2011.

[73] Nelson, M. J. and Mateas, M., "Search-based drama management in the interactive fiction Anchorhead," in *Proceedings of the 1st Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*, pp. 99–104, 2005.

[74] Nelson, M. J. and Mateas, M., "Towards automated game design," in *AI*IA 2007: Artificial Intelligence and Human-Oriented Computing*, pp. 626–637, 2007. Springer Lecture Notes in Computer Science 4733.

[75] Nelson, M. J. and Mateas, M., "Another look at search-based drama management," in *Proceedings of the 23rd AAAI Conference on Artificial Intelligence*, pp. 792–797, 2008.

[76] Nelson, M. J. and Mateas, M., "An interactive game-design assistant," in *Proceedings of the 2008 International Conference on Intelligent User Interfaces (IUI)*, pp. 90–98, 2008.

[77] Nelson, M. J. and Mateas, M., "Recombinable game mechanics for automated design support," in *Proceedings of the 4th Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*, pp. 84–89, 2008.

[78] Nelson, M. J. and Mateas, M., "A requirements analysis for videogame design support tools," in *Proceedings of the 4th International Conference on the Foundations of Digital Games (FDG)*, pp. 137–144, 2009.

[79] Nelson, M. J., Mateas, M., Roberts, D. L., and Isbell Jr., C. L., "Declarative optimization-based drama management in interactive fiction," *IEEE Computer Graphics and Applications*, vol. 26, no. 3, pp. 32–41, 2006.

[80] Nelson, M. J., Roberts, D. L., Isbell Jr., C. L., and Mateas, M., "Reinforcement learning for declarative optimization-based drama management," in *Proceedings of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 775–782, 2006.

[81] Nelson, M. J., Smith, A. M., and Smith, G., eds., *Proceedings of the AIIDE 2011 Workshop on Artificial Intelligence in the Game Design Process*. AAAI, 2011. Tech. Rep. WS-11-19.

[82] Nelson, M. J., Smith, A. M., and Smith, G., eds., *Proceedings of the AIIDE 2013 Workshop on Artificial Intelligence in the Game Design Process*. AAAI, 2013. Tech. Rep. WS-13-20.

[83] Nelson, M. J., Togelius, J., Browne, C., and Cook, M., "Chapter 6: Rules and mechanics," in *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, Springer, 2015. (To appear.).

[84] Nielsen, T. S., Barros, G. A. B., Togelius, J., and Nelson, M. J., "General video game evaluation using relative algorithm performance profiles," in *Proceedings of the 18th Conference on Applications of Evolutionary Computation*, 2015.

[85] Orwant, J., "EGGG: Automated programming for game generation," *IBM Systems Journal*, vol. 39, no. 3–4, pp. 782–794, 2000.

[86] Osborn, J. C., Grow, A., and Mateas, M., "Modular computational critics for games," in *Proceedings of the 9th Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*, pp. 163–169, 2013.

[87] Osborn, J. C. and Mateas, M., "A game-independent play trace dissimilarity metric," in *Proceedings of the 9th International Conference on the Foundations of Digital Games (FDG)*, 2014.

[88] Palmiter, S., Lynch, G., Lewis, S., and Stempski, M., "Breaking away from the conventional usability lab: The customer-centered design group at Tektronix, Inc.," *Behaviour & Information Technology*, vol. 13, no. 1–2, pp. 128–131, 1994.

[89] Pedersen, C., Togelius, J., and Yannakakis, G. N., "Modeling player experience for content creation," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 1, no. 2, pp. 121–133, 2009.

[90] Pell, B., "Metagame in symmetric, chess-like games," in *Heuristic Programming in Artificial Intelligence 3: The Third Computer Olympiad*, Ellis Horwood, 1992.

[91] Pizzi, D., Cavazza, M., Whittaker, A., and Lugrin, J.-L., "Automatic generation of game level solutions as storyboards," in *Proceedings of the 4th Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*, pp. 96–101, 2008.

[92] Price, G. R., "How to speed up invention," *Fortune Magazine*, pp. 150–228, November 1956.

[93] Reintjes, J. F., *Numerical Control: Making a New Technology*. Oxford University Press, 1991.

[94] Riedl, M. O., Saretto, C. J., and Young, R. M., "Managing interaction between users and agents in a multi-agent storytelling environment," in *Proceedings of the 2nd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2003.

[95] Riley, J. P. and Lawson, B. R., "RODIN: A system of modeling three dimensional roof forms," in *Proceedings of CAD 1982*, 1982.

[96] Roberts, D. L., Bhat, S., St. Clair, K., and Isbell Jr., C. L., "Authorial idioms for target distributions in TTD-MDPs," in *Proceedings of the 22nd AAAI Conference on Artificial Intelligence*, pp. 852–857, 2007.

[97] Roberts, D. L., Cantino, A. S., and Isbell Jr., C. L., "Player autonomy versus designer intent: A case study of interactive tour guides," in *Proceedings of the 3rd Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*, pp. 95–97, 2007.

[98] Roberts, D. L. and Isbell Jr., C. L., "A survey and qualitative analysis of recent advances in drama management," *International Transactions on Systems Science and Applications*, vol. 3, no. 1, pp. 61–75, 2008.

[99] Roberts, D. L., Nelson, M. J., Isbell Jr., C. L., Mateas, M., and Littman, M. L., "Targeting specific distributions of trajectories in MDPs," in *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI)*, pp. 1213–1218, 2006.

[100] Rosch, E., "Natural categories," *Cognitive Psychology*, vol. 4, pp. 328–350, 1973.

[101] Ross, D. T., "Gestalt programming: A new concept in automatic programming," in *Proceedings of the Western Joint Computer Conference*, pp. 5–10, 1956. Summarized with commentary in [102].

[102] Ross, D. T., "A personal view of the personal work station: Some firsts in the fifties," in *Proceedings of the ACM Conference on the History of Personal Workstations*, pp. 19–48, 1986.

[103] Rugaber, S., Goel, A. K., and Martie, L., "GAIA: A CAD environment for model-based adaptation of game-playing software agents," in *Proceedings of the 2013 Conference on Systems Engineering Research*, pp. 29–38, 2013.

[104] Salge, C., Lipski, C., Mahlmann, T., and Mathiak, B., "Using genetically optimized artificial intelligence to improve gameplaying fun for strategical games," in *Proceedings of the Sandbox 2008 ACM SIGGRAPH Videogame Symposium*, pp. 7–14, 2008.

[105] Schell, J., *The Art of Game Design: A Book of Lenses*. CRC Press, 2008.

[106] Schleiner, A.-M., "Loiselit puuttuvat peliin: Pelien muuntelu ja hakkeritaide," in *Mariosophia*, Gaudeamus, 2002. English translation, as "Parasitic Interventions: Game Patches and Hacker Art", available at `http://www.opensorcery.net/patchnew.html`.

[107] Schön, D. A., *The Reflective Practitioner*. Basic Books, 1983.

[108] Schön, D. A., "Designing as reflective conversation with the materials of a design situation," *Research in Engineering Design*, vol. 3, pp. 131–147, 1992.

[109] Shanahan, M., *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia*. MIT Press, 1997.

[110] Sigman, T., "The siren song of the paper cutter: Tips and tricks from the trenches of paper prototyping," *Gamasutra*, September 13, 2005. `http://www.gamasutra.com/view/feature/2403/`.

[111] Simon, H. A., *The Sciences of the Artificial*. MIT Press, 1969.

[112] Singh, P., "The public acquisition of commonsense knowledge," in *Proceedings of the AAAI Spring Symposium on Acquiring (and Using) Linguistic (and World) Knowledge for Information Access*, 2002.

[113] Smith, A. M., *Mechanizing Exploratory Game Design*. PhD thesis, University of California, Santa Cruz, 2012.

[114] Smith, A. M., "Open problem: Reusable gameplay trace samplers," in *Proceedings of the AIIDE 2013 Workshop on Artificial Intelligence in the Game Design Process*, pp. 22–27, 2013.

[115] Smith, A. M. and Mateas, M., "Variations Forever: Flexibly generating rulesets from a sculptable design space of mini-games," in *Proceedings of the 2010 IEEE Symposium on Computational Intelligence and Games (CIG)*, pp. 273–280, 2010.

[116] Smith, A. M. and Mateas, M., "Answer set programming for procedural content generation: A design space approach," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 187–200, 2011.

[117] Smith, A. M. and Mateas, M., "Computational caricatures: Probing the game design process with AI," in *Proceedings of the AIIDE 2011 Workshop on Artificial Intelligence in the Game Design Process*, pp. 19–24, 2011.

[118] Smith, A. M., Nelson, M. J., and Mateas, M., "Computational support for play testing game sketches," in *Proceedings of the 5th Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*, pp. 167–172, 2009.

[119] Smith, A. M., Nelson, M. J., and Mateas, M., "Prototyping games with BIPED," in *Proceedings of the 5th Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*, pp. 14–16, 2009.

[120] Smith, A. M., Nelson, M. J., and Mateas, M., "Ludocore: A logical game engine for modeling videogames," in *Proceedings of the 2010 IEEE Symposium on Computational Intelligence and Games (CIG)*, pp. 91–98, 2010.

[121] Smith, G., Whitehead, J., and Mateas, M., "Tanagra: Reactive planning and constraint solving for mixed-initiative level design," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 201–215, 2011.

[122] Steinberg, L. I. and Mitchell, T. M., "The redesign system: A knowledge-based approach to VLSI CAD," *IEEE Design & Test of Computers*, vol. 2, no. 1, pp. 45–54, 1985.

[123] Stiny, G. and Gips, J., "Shape grammars and the generative specification of painting and sculpture," in *Information Processing 71: Proceedings of IFIP Congress 1971 (Volume 2 — Applications)*, pp. 1460–1465, 1971. `http://www.shapegrammar.org/ifip/ifip1.html`.

[124] Sullivan, A., Chen, S., and Mateas, M., "From abstraction to reality: Integrating drama management into a playable game experience," in *Proceedings of the 2nd Workshop on Intelligent Narrative Technologies (INT)*, pp. 111–118, 2009.

[125] Sutherland, I. E., *Sketchpad: A Man-Machine Graphical Communication System*. PhD thesis, Massachusetts Institute of Technology, 1963. Digitized version with a new preface available as Tech. Rep. UCAM-CL-TR-574, University of Cambridge Computer Laboratory, 2003.

[126] Sutton, R. S., "Learning to predict by the methods of temporal differences," *Machine Learning*, vol. 3, no. 1, pp. 9–44, 1988.

[127] Tesauro, G., "Practical issues in temporal difference learning," *Machine Learning*, vol. 8, no. 3–4, pp. 257–277, 1992.

[128] Tesauro, G. and Galperin, G. R., "On-line policy improvement using Monte-Carlo search," in *Advances in Neural Information Processing Systems (NIPS)*, pp. 1068–1074, 1996.

[129] Thompson, C., "Halo 3: How Microsoft Labs invented a new science of play," *Wired*, 2007. August 8, 2007. http://www.wired.com/gaming/virtualworlds/magazine/15-09/ff_halo.

[130] Togelius, J., Nelson, M. J., and Liapis, A., "Characteristics of generatable games," in *Proceedings of the 5th Workshop on Procedural Content Generation in Games*, 2014.

[131] Togelius, J. and Schmidhuber, J., "An experiment in automatic game design," in *Proceedings of the 2008 IEEE Symposium on Computational Intelligence and Games (CIG)*, pp. 111–118, 2008.

[132] Togelius, J., Yannakakis, G. N., Stanley, K. O., and Browne, C., "Search-based procedural content generation: A taxonomy and survey," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 172–186, 2011.

[133] Treanor, M., Blackford, B., Mateas, M., and Bogost, I., "Game-O-Matic: Generating videogames that represent ideas," in *Proceedings of the 3rd Workshop on Procedural Content Generation in Games*, 2012.

[134] Treanor, M. and Mateas, M., "Newsgames: Procedural rhetoric meets political cartoons," in *Proceedings of the 2009 Digital Games Research Association Conference (DiGRA)*, 2009.

[135] Treanor, M., Mateas, M., and Wardrip-Fruin, N., "Kaboom! is a many-splendored thing: An interpretation and design methodology for message-driven games using graphical logics," in *Proceedings of the 5th International Conference on the Foundations of Digital Games (FDG)*, pp. 224–231, 2010.

[136] Treanor, M., Schweizer, B., Bogost, I., and Mateas, M., "The micro-rhetorics of Game-O-Matic," in *Proceedings of the 7th International Conference on the Foundations of Digital Games (FDG)*, pp. 18–25, 2012.

[137] Tremblay, J., Torres, P. A., and Verbrugge, C., "Measuring risk in stealth games," in *Proceedings of the 9th International Conference on the Foundations of Digital Games (FDG)*, 2014.

[138] van Harmelen, F., Lifschitz, V., and Porter, B., *Handbook of Knowledge Representation*. Elsevier, 2008.

[139] Wardrip-Fruin, N., *Expressive Processing: Digital Fictions, Computer Games, and Software Studies*. MIT Press, 2009.

[140] Weisberg, D. E., *The Engineering Design Revolution*. CadHistory.net, 2008.

[141] Weyhrauch, P., *Guiding Interactive Drama*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1997. Tech. Rep. CMU-CS-97-109.

[142] Wirman, H., *Playing The Sims 2: Constructing and negotiating woman computer game player identities through the practice of skinning*. PhD thesis, University of the West of England, 2011.

[143] Wixon, D., Holtzblatt, K., and Knox, S., "Contextual design: An emergent view of system design," in *Proceedings of the 1990 Conference on Human Factors in Computing Systems (CHI)*, pp. 329–336, 1990.

[144] Woodbury, R. F., "Searching for designs: Paradigm and practice," *Building and Environment*, vol. 26, no. 1, pp. 61–73, 1991.

[145] Young, R. M., Riedl, M. O., Branly, M., Jhala, A., Martin, R. J., and Saretto, C. J., "An architecture for integrating plan-based behavior generation with interactive game environments," *Journal of Game Development*, vol. 1, no. 1, 2004.

[146] Zagal, J. P., Mateas, M., Fernández-Vara, C., Hochhalter, B., and Lichti, N., "Towards an ontological language for game analysis," in *Proceedings of the 2005 Digital Games Research Association Conference (DiGRA)*, 2005.