UNIVERSITY OF CALIFORNIA
SANTA CRUZ

OPERATIONALIZING OPERATIONAL LOGICS

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Joseph C. Osborn

June 2018

The Dissertation of Joseph C. Osborn
is approved:

Professor Michael Mateas, Chair

Noah Wardrip-Fruin

Julian Togelius

Dean Tyrus Miller
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

Operationalizing Operational Logics

by

Joseph C. Osborn

Since their initial development by Wardrip-Fruin and further exposition by Wardrip-Fruin and Mateas, *operational logics* (OLs) have enjoyed broad use and inspired several approaches to game studies. Besides their direct application in describing specific games, OLs underlie several approaches to understanding how games communicate ideas and a variety of projects in player and game modeling and game generation. The key move in all these cases has been to step away from considering games as bags of mechanics and towards viewing them as assemblages of abstract operations from diverse logics.

This dissertation expands on the theory of operational logics in its first part, and in the second part applies this refined theory to problems of interest in game modeling, game design support, and automated game design learning.

To Melissa, to Joey and Norah, and to my father.

Acknowledgments

I could never have completed this work without my partner Melissa's patience, support, and forceful brand of project management. I love you.

The theoretical foundations of this work were inspired by and developed in collaboration with Noah Wardrip-Fruin and Michael Mateas, who over the long years of my PhD have gone from mentors to colleagues to friends.

The applications of this work, especially around automated game design learning, would have been outside my reach without Adam Summerville's expertise and friendship or overlong conversations with Christoffer Holmgård. Likewise, my work in game design model checking was helped along tremendously by conversations with Adam Smith and Mark Nelson (and in fact, their papers, along with those of Chris Lewis, inspired me to enter this PhD program in the first place). Thanks also to Jim Whitehead for connecting me with the crowd-sourced formal verification project, in the course of which I learned a lot about formal verification.

Thanks to the whole EIS lab and UCSC Computational Media Department, where I have always felt welcome and at home, while at the same time challenged beyond what I thought were my abilities. Special thanks to Melanie Dickinson, the heart of the lab's community, and to Eric Kaltman, Jacob Garbe, James Ryan, April Grow, Kate Compton, and Ben Samuel, with whom I have had so many edifying discussions.

Thanks to my committee, and apologies for the length.

Thanks to my family's love and support, without whom none of this would have been possible.

Part I

Operationalizing Operational Logics

Chapter 1

Why Game Modeling Languages?

What are games made of? Somehow, this ill-posed and seemingly subjective question became the focus of my PhD research in artificial intelligence and design support. It was probably unavoidable: the success or failure of AI projects depends on the choice of *knowledge representation [7],* and while there are dozens of theories of games none had yet been produced that was appropriate for my purposes. This dissertation explores one way of posing and answering this confounding quandary in the first section, and details some applications of my answer in the second. [1]

This work proceeds thanks to a particularly useful pair of perspectives: *operational logics* and *playable models.* Operational logics [268] describe at once how games *operate* and how they *communicate* these operations to players, while playable model theory explains how games, in concert with cultural knowledge, build up more complex structures supporting incremental learning and exploration. My goal is to apply AI to every part of the game development

[1] *Portions of this chapter originally appeared in "Combat in Games" [192].*

process, from design through testing, release, and iterative improvement. Operational logics give the building blocks for performing automatic analysis of game designs and game systems, while playable models yield, for particular models that an analyst has already identified, sources of specifications and expectations about how a game *ought to* behave.

The goal of this dissertation is to define a language for describing *what games are made of* in a way that is useful both to people (showcased in part one) and machines (the main subject of part two). *People* here includes designers, critics, scholars, and others who will obtain natural ways to express aspects of game designs and thoughts about games in formal notations. *Machines,* or more correctly computer programs, will gain representations for interpreting the phenomena and rules of games that already exist (whether given as formal rule systems or as opaque computer programs).

1.1 Introduction

In this dissertation, I argue that projects in game interpretation and game design science—whether performed by humans or by AI systems—ought to take operational logics as their starting point. In the service of this argument I situate my work over the last four years (since I began to explore operational logics) in the context of new theoretical developments made during the construction of my dissertation. I make four key contributions to game studies and game design support:

1. The refinement of operational logic theory (Chapters 1 – 6), including rephrasings of existing critical tools and computational systems in those terms.

2. The operationalization of operational logics, down to the level of specifying complete games in terms of how they compose operational logics (Chapters 7 – 8).

3. The exposition of strong connections between operational logics and formal logics, as a starting point for knowledge representation work (Chapters 9 – 10).

4. The elaboration of five significant computational-intelligence systems, all of which I constructed in the past two to three years, based on such knowledge representations as evidence for an operational logics-first approach (Chapters 11 – 15).

The theory and applications presented in this dissertation are powerful and can drive research agendas in game design support, automated game design learning, and even game studies. The schema of finding formal analogues to game design concepts in Chapter 9 has already been fruitful, and I hope that, with or without operational logics as an intellectual core, it becomes a commonly accepted way of exploring the possibilities of game design space—of course, I hope I will convince the reader that operational logics are indeed the preferred choice for that core world-view.

1.2 Why game modeling languages?

How *specifically* would the formal language I'm discussing benefit the groups of people described above? What are the *machines* I imagine creating and why bother helping them to understand game designs? What do I mean by "useful?" Moreover, there is no shortage of formalisms and semi-formalisms around games and game design, and I should show why these

have been insufficient to achieve the aims above (or at the very least, why the world needs yet another such formalism).

At this point, many game designers have advocated for more formality in their design language, from Crawford [79] up through Costikyan [75], Church [56], Koster [136], Cook [65], and Adams & Dormans [6]. These formalisms, proponents argue, give new ways to communicate design knowledge within teams and to explain an expert designer's understanding of game rules to junior designers. They can form the basis of design documentation and give tools for thinking through design problems, and designers can use these languages to codify their expertise and rules of thumb in the style of design patterns [41].

Subject experts outside of games can use these pattern libraries and design guidelines to support games with a purpose, such as persuasive, educational, or citizen-science games. These experts are often novices at game design (though they have considerable knowledge in their own domain) and can benefit in all the ways junior designers can; moreover they can use these languages and tools to communicate more effectively with game development teams, maintaining some degree of confidence that their proposed designs do not conflict with their subject matter.

In the same way, these languages can ground critical inquiry into game designs and the circumstances of game play and production. This can be of great use to scholars and critics of media (including games) as well as to game players, who can better interrogate the games they play and what arguments they are making, intentionally or unintentionally. Communities of game players often come up with their own terms of art around games and genres of interest, and this seems to be an important part of developing and teaching expert play. Finding the

common threads underlying these *ad hoc* inventions could ground ethnographic studies of these communities of play in the formal properties of the games themselves.

It is therefore clear that a broad variety of players and practitioners gain from useful and usable ways to describe games; what options do they have today? While there are frameworks for understanding game players [31, 272, 231], genres [16], rules [122], strategic depth [145], high-level structure [129], and virtual sensation [247], it remains difficult to synthesize insights that simultaneously cover how the game operates and how it communicates with its players about those operations; one of this work's two main goals is that the language described in this dissertation provides some hooks to begin integrating these important efforts. The primary mechanism by which I propose to do this is to give a vocabulary of and grammar for *ground terms:* a common foundational language in terms of which the other formalisms can be described. The theory of operational logics gives such a base platform, and in Chapter 6 I show how the frameworks of proceduralist readings [259] and playable models [165] can be realized in terms of operational logics.

The second goal of this dissertation is that the proposed language of game design provides useful affordances for the *automated* analysis of games by machines. A *machine* here is a program that interprets a game program, game design, or observations of game play for some purpose. The simplest case to consider is the automated game playing program that tries for optimal play; this could work by treating the program like a black box, or it might be given some domain knowledge like a board evaluation function or heuristic, or it might be trained on a vast set of records of tournament play. One can also imagine automated players which try for human-like play, or try to break a game design or program by performing random, pathological,

or novelty-seeking actions.

The space of automated interpreters is much broader than just AI players. I have prototyped automated players that also record what they see and build maps of game worlds or databases of character animations or background music, or even accumulate statistics about the items and enemies they have encountered. Of course, this can also be done without explicitly playing the game, for example by extracting data files or using tools from the reverse-engineering literature. The extent to which the extracted static assets are representative of the game in play is an interesting question to which I will return in Chapter 14.

Beyond "play," one might want to use software to visualize the activity of populations of players, perhaps using knowledge about the game design to contextualize that activity. One can also use automated analysis in the design stage; such a system could automatically test variations on a design or visualize a game's possibility space (or even the conceptual space around the currently realized design). Generally, it can often be shown whether a design fulfills certain desired *properties:* e.g., that certain content cannot be skipped or that events must occur in a particular order, that the player character must always or never be able to reach some area, or that a puzzle level requires that solutions meet some criterion (especially important for educational games).

Software is integral to the game-making process too: game-making tools which can analyze the game they are building could also be introspective about their product and offer critique, support, or inspiration. This could also be a boon for the area of *automated game design.* Simply making design-level concepts first-class in the tool's graphical or textual language can prevent certain classes of design or programming bugs by construction (or offer some equivalent

of compiler warnings) to make certain classes of design easier to express. One can also imagine ways to make games more accessible to people with different sensory or motor capabilities: a game could observe itself and describe its on-screen action verbally, repair design flaws like color juxtapositions which are indistinguishable to the player, reinforce audio cues with visual highlights, or even transform its own rules to change a real-time game into an "equivalent" (in some sense) discrete-time or turn-taking game.

Broadly, I want to help players and designers better understand the dynamics of the game they are making given its rules or behavior, to support better play and better design. All this depends on computational systems which can *understand* the games they are playing or making in terms which are amenable to automated analysis. While new game-making tools could be built around the design language described in this dissertation (and indeed I explore one in Chapter 12), I hope that the kinds of analyses described above do not depend on reimplementing existing games to take advantage of them. To that end I have spent some effort on *recovering* game designs from game programs (for example, in Chapter 15); this could support round-tripping from a game program to a model of its design and then, through code generation or some other game-specific mechanism, back to the game in question.

1.3 Why *another* game modeling language?

Other approaches to describing game designs have some ontological issues as well as pragmatic ones. For example, the Game Ontology Project [275] and related efforts like Game Design Patterns [41] decompose games with base terms that come from effectively infi-

nite grammars; there is no limit to the number of types of objects which might appear in games, or the number of design patterns that might be identified. This would mean that any innovation in game design would obsolete the theoretical tools, and even worse I would have to complete a huge ontology-development project before being able to do any serious work. Another such *unhelpful base* arises in rigorous formats like program source code, which have unambiguous meanings and a small set of underlying primitives but are so low-level that they only have *implicit* information about the design; the design gets hidden in the specification of the program's concrete behavior.

A second issue that arises is that many game description languages handle (by design) only the internal rules and simulation [145] or else only the external behavior and instantial assets of the system, and there is no clear way to synthesize these threads (thus the history of scholars and designers talking past one another in the ludology and narratology debates).

The necessity of understanding such connections is emphasized in many of the most influential writings about games. Juul's *Half-Real,* for example, foregrounds the connections between a game's rules and its fiction [130]. Similarly, the MDA framework, which emphasizes the connections between systems and player experiences, has for years been at the core of the Game Design Workshop offered at the Game Developers Conference [122]. Game Feel [247] is another step in the right direction but it does not always make the connections between these two worlds explicit enough, and it limits its scope to virtual sensation: cameras, characters, physics, and so on.

How can a designer or researcher follow through on what these ideas recommend: that one must think about how games function and how they communicate at the same time?

While other frameworks give names to the different categories one must think about (e.g., Juul's "rules" and "fiction") they offer little guidance for how to conceptualize or discuss entities that cut across the categories. This is also true of work that focuses specifically on conceptualizing game entities. For example, Koster entirely brackets off how games are experienced as media [136]. Koster's work is in dialogue with that of Ben Cousins, who writes about the "primary elements" of games as the conscious player interactions that cannot be further subdivided [77]. In practice, this put Cousins's work at the level of game mechanics (e.g., "jump" or "shoot") with no way to talk about the logics that support them.

Finally, most game languages are insufficiently broad to describe or account for a complete game, leaving some phenomena, relations, or design attributes on the floor. Even those languages that consider how people understand and make decisions about game play often stop at the level of game rules and feedback, leaving those to be defined in natural language or source code (equally problematic notations, but for different reasons). Game-making tools have this property as well: Game Maker and Unity, for example, require the use of a textual programming language for implementing mechanics which are not supported by its drag-and-drop interface. Explicit game modeling languages without such *Turing trap-doors* generally work for only limited classes of games.

These issues are not unique to the languages people use to describe games to other people. The specification languages and formalisms used in software verification have similar problems. Often, these languages are under-constrained and require lots of expertise to use since they must work for arbitrary programs; specifications and proofs of correctness can be harder to write than the programs themselves. Those formalisms which are comparatively

easy to use (for example, type annotations and type checkers) might cover only limited aspects of the program and leave some of its behavior under-specified. Orienting verification around programming languages and data types makes sense for working with software in general, but necessarily stops at the level of domain knowledge. For my purposes, these are in the wrong terms: only the simulation and not the perception or communication are addressed, and the simulation is handled at too fine-grained a level of abstraction. This means that the responsibility for incorporating domain knowledge correctly is solely in the hands of the programmer or modeler: specifications, invariants, proofs, and everything.

Verification can also proceed by explicitly modeling a design in terms of higher-level logics like finite state machines, Petri nets, or communicating sequential processes; these are then verified using off-the-shelf solvers. This is a powerful technique but it requires subject expertise to identify which models are appropriate and how to build such models. Moreover, unless program code generation is part of the toolset, there remains a question as to whether the implemented program or game matches the verified design.

The thesis of this dissertation is that the problems of design languages and those of verification languages can be resolved simultaneously by getting away from code a little bit and focusing on the complete specification problem as a first-class problem. I explore why operational logics are an appropriate tool for the critical half of this project in Chapters 2 and 4, and show worked examples of grounding out specific critical frameworks in Chapters 5 and 6. Chapters 7, 8, and 9 (in fact, all of Part II) show how these structured logics can be used to solve substantial modeling and verification problems with relatively light-weight approaches.

Chapter 2

Operational Logics

Since their initial development in [268, 269] and further exposition in [166], *operational logics* (OLs) have enjoyed broad use and inspired several approaches to game studies thanks to the efforts of my co-advisors Noah Wardrip-Fruin and Michael Mateas. Besides their direct application in describing specific games [169], OLs underlie several approaches to understanding how games communicate ideas [43, 259] and a variety of projects in player and game modeling and game generation [89, 260, 162, 164, 244, 243]. The key move in all these cases has been to step away from considering games as bags of mechanics and towards viewing them as assemblages of abstract operations from diverse logics. [1]

Outside of the game studies literature, in recent years Wardrip-Fruin and I have each taught an introductory undergraduate game design class using operational logics as the unifying theme. Each module of the course addresses a different set of operational logics, building up a core set of literacies for interpreting and for designing games. Besides the analysis of

[1] *Portions of this chapter originally appeared in "Refining Operational Logics" [198] and "Combat in Games" [192].*

games, game-making tools are described in terms of the logics they support, which gives students context for understanding new tools' authorial affordances. Students progress through a variety of tools over the course of the quarter, and the operational logics framework gives them a touchstone for deciding which tool to use and how best to use it.

Recently, I worked with Wardrip-Fruin and Michael Mateas to refine and operationalize the theory of operational logics as well as to catalog the main categories of logics employed by a broad variety of games [198]. This project has formed the kernel of the first part of this dissertation.

2.1 What are Operational Logics?

In the previous chapter, I identified a hole in existing game ontologies and formalisms for describing game designs: very few approaches handle at once both how a game functions internally and how that simulation is communicated to players. Operational logics address this gap.

On a detailed level, looking at an operational logic (such as collision detection) both names a general strategy (how it combines an abstract process and a communicative goal) and gives a way of talking about how a particular game, or part of a game, employs the strategy (the specific algorithmic implementation, game state representation, and player experience). For example, at the level of operational logics, I interpret moving patterns of pixels—sprites—that stop when touching each other to be *objects* that *collide* due to collision logics. Sudden upwards vertical movement of such a sprite followed by its rapid descent is read as *jumping* in

the presence of *gravity* thanks to physics logics. When the teletype prints out a block of text explaining "You are in…," the player's sense of place among linked rooms requires that this text does not describe a different room when she *looks* again, but does when she *goes up stairs* or *enters portal* via a linking logic. Operational logics provide the combinations of interpretive framing and system behaviors necessary to make sense of the screen.

More formally: "Operational logics connect fundamental abstract operations, which determine the state evolution of a system, with how they are understood at a human level [166]." In other words, operational logics tie together low-level and abstract mechanics with the presentation of sensory stimuli to players to enable an understanding of what the game is doing and how it functions. Operational logics are neither beneath nor above game mechanics, but represent a different *slice* through a game which I feel is often a more useful view on how games function in the space between designers and players. Operational logics at once provide the concrete ingredients of game simulation rules which make up mechanics *and* the abstract channels for communicating game state evolution to players which lie outside the simulation.

As I later wrote, "Operational logics are combinations of *abstract processes* with their *communicative roles* in a game, connected through an ongoing *game state presentation* and supporting a *gameplay experience* [192]." These elements working in concert help (idealized) players interpret meaningful causal relationships between observed phenomena and a hypothesized underlying simulation.

A *communicative role* (also called a communicative strategy) describes how the logic must be employed authorially to communicate its operations to players as part of the larger game system. One can group logics into families which share the same communicative role, though

they might support that role with different processes or state presentations. For example, the generic communicative role of collision detection could be stated as "Virtual objects can touch, and these touches can have consequences." If this is not revealed to players—e.g., if the objects' visual locations fall out of sync with their simulated positions—the role is not fulfilled and the logic falls apart. The communicative role is the phenomenon in terms of which designers author game rules, and which designers hope that players engage with and understand. Of course, such communication does not always succeed for all audiences, but the most common logics are widely (if subconsciously) understood by game-literate players. It is through this understanding that players develop their ability to play: to understand the game-world such that they can take action intentionally and interpret its results, an important part of the experience of agency many games provide [270].

An *abstract process* is a specification for how a process operates. For example, the abstract process for collision detection could be stated as, "When two virtual objects intersect, declare the intersection." This specification is agnostic as to the specific algorithm and implementation—the coordinate spaces would be quite different for 2D and 3D games, as well as for games using rough bounding boxes versus pixel-accurate methods. Not all possible implementations of an abstract process may succeed in supporting the communicative role of the logic for all audiences or in all contexts. Candidate implementations must therefore be *effective* as well as *feasible*: as Mateas and Wardrip-Fruin assert, the definition of the abstract process must be implementable on a computer [166].

The term "process" is used broadly here and covers both state-altering procedures and predicates; I find it useful to call the obligations a logic places on its implementation *abstract*

operations, which one may also call *operators* of the logic. Some abstract operations specify relations or invariants maintained by the logic, while others describe the sorts of queries the logic supports, and still others opportunistically trigger abstract operations when predicates of a logic become true or false.

A *game state presentation* is how players see, hear, and feel the specific behavior of the operational logic in the context of the game. Games' platforms can offer diverse options for presentation: compare vector versus raster graphics, stereoscopic 3D versus flat images, and the proprioceptive experience of motion-controlled versus keyboard-controlled games. Different ways of presenting the game state can require very different data—often called game *assets*—to support presentation of the logic's operations. For example, the only asset specific to the presentation of collision detection in *Pong* is a sound triggered when the ball collides with a wall or paddle. But in a *Call of Duty* game, the presentation of collision detection for bullets and bodies alone may involve many animation and sound assets required to realistically present different types and locations of damage. This is a general trend as the level of detail represented in a game grows finer: more data assets are required for a logic's presentation. The crafting and selection of data is a key way game creators can suggest more concrete audience interpretations, especially for logics that have been established with relatively abstract communicative goals: pattern-matching logics can readily support *crafting* if the patterns resemble the crafted artifact, and linking logics can naturally evoke *spatial connection* if the nodes and links are styled like a mass transit map.

The *gameplay experience* is what happens when a player encounters the logic. The player engages simultaneously with the sensory experience of the game state presentation, the

intentional act of performing actions which change the game state, and the interpretive process of determining a mapping between physical interaction devices and the available actions (in terms of the operational logics' abstract operations). This is where the game's creators hope the communicative role will be fulfilled, and also where players may discover possibilities the designers never intended. Often player understanding and discovery is not immediate, and it may be imperfect, especially in its details—players learn through experimentation, after all. This is especially true on the boundaries between logics.

Even in a simple game such as *Pong* or *Breakout*, the physical reaction triggered when a ball collides with a paddle can differ depending on where along the paddle the ball collides; this basic connection between collision and physics may take time for players to grasp. Nonetheless, the communication that balls and paddles can collide, and that balls bounce off the paddle when this happens, takes place quickly in nearly all initial play sessions (confirming the expectations of game-literate players).

2.2 How operational logics communicate

Compared to abstract operations, the communicative roles and game state presentations of operational logics are relatively under-theorized. While the translation from abstract operations to implemented code can be fairly natural, most examples of communicative roles and presentations have been either very specific (e.g., how a health bar shows the state of the health resource) or extremely broad (e.g., two-dimensional shapes on a screen standing in for colliding objects); this is true even in the discussion above! It is clear that the role and state pre-

sentation of the logic are tightly connected, just as the role and abstract operations are tightly connected. A concrete presentation must effectively reveal the workings of a logic in the same way that an implementation of the abstract processes must effectively enact them.

Recall that the communicative role of a logic is what a player must understand from the game to support an interpretation that the logic is present. This requires a transformation from invisible internal game state to visible audiovisual phenomena. A logic's game state presentation maps the underlying game state into sensory phenomena to support that role. Every game takes its own approach to that mapping for each of its logics, fulfilling their respective communicative roles—exactly as every game has its own concrete implementations of operational logics' abstract processes. Just as one can recognize the abstract operations of resource transactions independently of their concrete implementation in different games, it is possible to discuss approaches to game state presentation that are common across games. Resource logics, for example, are often presented by placing numbers next to icons and/or textual labels, or animating a floating number near the resource pool whose quantity of contained resources has just changed.

Every logic can be identified with a somewhat open-ended set of presentation strategies, although these necessarily depend on the literacies of the target audience. Often, these strategies may overlap in some ways with those of other logics. As an example, the two-dimensional combat game *Worms* expresses both its collision and physics logics by drawing all simulated characters and terrain on the same plane, so that visual overlaps and perceived movement reflect simulated collisions and physical dynamics. When characters in *Worms* are injured, the amount of damage is displayed as a floating number originating at the character's

head and rising upwards while fading out. The resource logic leverages some of the same *channels* used by the collision and physics logics' game state presentation.

Ultimately, game state presentation can only be defined completely when logics are composed into a game. It is the composition proper that provides the set of communication channels used to fulfill the communicative role according to the appropriate presentation strategies. I save a complete discussion of how operational logics compose for Chapter 4.

2.3 Why operational logics?

I have explained *what* operational logics are, but it remains to be shown *why* they are appropriate for the project of this dissertation: describing *what games are made of* in a way that is useful both to people and machines. It will become clear in Chapters 4 and 7 that operational logics effectively specify games at a suitable level of detail. But why are operational logics *useful* to people and to machines? There are three aspects of operational logics theory which make them suitable for this purpose:

1. Operational logics *unify simulation and communication;*

2. Operational logics are *composable;* and

3. Operational logics are *countable.*

An operational logics account of *Call of Duty* (for example) must address both how the simulation functions (e.g., different types and locations of damage) and how this relates to the qualities of objects' animations and sound effects (e.g., different animation and sound

assets). The interpretation would break down if the animations played a few seconds *after* the collision and resulting health loss, or if bullets shot at one player passed right through while another player received damage as normal. A complete operational logics reading leaves no audiovisual phenomenon or aspect of the simulation unexplained.

This is evidently useful on the interpretation side: an analyst can address more of a game than they could before, especially the *relations* between different elements of the game. Interpreters can explain how game mechanics are built out of and communicated through the operators—the authorial affordances—provided by operational logics, address similarities between mechanics, and trace how the terms of one logic (e.g., resources) flow through the system and influence others (e.g., collisions or linking) in a rigorous, unified framework, and I do so in Chapter 4.

From the definitional side, tools like Game Maker, Unity, or Twine *center* or *privilege* different combinations of logics, and their internal schemata likewise commit to certain relationships between what's drawn on screen and what's simulated in the system. Moreover one can hypothesize *new* game-making tools that use different or unusual logics or combinations thereof. These tools can even take the form of game generators whose knowledge representations encode particular operational logics (as in Gemini [242]). If game developers were to build games in terms of operational logics (as I advocate in Chapter 7), that would yield natural hooks for analysis: one could mechanize techniques like proceduralist readings or accessibility measures and have the rules of the game accessible in a useful and usable format.

Part of why this is possible is that operational logics *compose* together. The events that might happen after a collision can come from any logic, and likewise the conditions under

which a resource transaction is allowed. These low-level compositions roughly correspond to *game mechanics,* as shown earlier.

Logics also compose in more complex ways. Consider a game like *Diablo* and especially its inventory and crafting systems. I consider the notion of a *system* in more detail in Chapter 4; it is another undertheorized concept which players and designers intuitively understand but take as obvious enough that no one has explored the general idea extensively. *Diablo* has a simple crafting system based on having sufficient quantities of different inputs which are then used up to produce some number of outputs. Its inventory is a grid on which stacks of objects are placed, an abstract model of a sack in which five potions take up the same space as a single potion while five swords take up five times the space of one. The crafting system must take this combined numeric and spatial representation of the inventory, project away the space (leaving only numbers), and then ensure that the inventory after removing the inputs has enough space to contain the outputs. This is a subtle and nontrivial mapping which can be accounted for using operational logics and logical specifications without resorting either to natural language explanations or to programming language code.

Finally, operational logics can be organized such that the number of logics (rather, the number of *families* of logics) can be counted—and, importantly, that this number is relatively small. Chapter 3 explains the process through which I split apart categories of logics according to their communicative role and contains a provisional catalog which covers a broad variety of games. This is one of the key contributions of my dissertation, because this catalog amounts to a handbook to using operational logics in critical analysis or in devising game-making tools.

In Chapter 8, I take this contribution further, giving two concrete, computational in-

carnations of operational logics. This admits *modular game definition languages,* supporting *game generator generators* which could pick compositions of logics and then produce game generators for that particular design space. This project continues into Part 9 where each family of logics is associated with a *formal logic* or theory which has similar authorial affordances and specification power; this yields an approach to automatically model-checking and verifying game designs written using each of these logics in a modular way.

2.4 What aren't OLs?

What is or isn't a logic is discussed in greater detail in Chapter 3, and the ways in which logics combine—or, indeed, might try to stand in for each other—is handled in 4, but here I can distinguish two broad categories of things around games which are not directly addressed by operational logics.

2.4.1 Player models

What I say about AI agents goes doubly for human players. Operational logics does make certain predictions about—or at least explains itself in terms of—cognitive processes occurring in game players, but these are not validated experimentally. Even beyond the fundamental perceptual activities taking place in recognizing game phenomena and forming mental models of game rules, questions of why players play games, social play, differences between players, player skill, and so on are all left open by operational logics and this dissertation. Again, this is an intentional move: operational logics are expressly about what happens at the

interface between games and humans, and it makes sense to leave some mystery on the human side even as I operationalize the game side further.

One can, however, leverage operational logics to inform player modeling! Like AI "players," humans interacting with a game engage with the operational logics at work, and the models they form of how the game operates are the basis of their interaction with the game system; for example, if players cannot identify the logics at work in a game they have no effective agency. The resolution of the conflict between the player's beliefs about the game and the logics it truly supports is the foundation of playable model theory [165]. Moreover, there are places where player modeling could incorporate information about the game to classify or describe players more precisely, moving from *model-free* approaches like Gamalyzer [200] and game-specific approaches like PaSSAGE [253] to the operational logics-founded player modeling incorporated in Gemini [242]: what might a player choose to do and what must a player do, and how can the player's preferences be characterized?

2.4.2 Cultural knowledge

Players consider more than just cause and effect relationships when playing games. Operational logics advance game studies by establishing a coherent account of how players interpret such causal relationships and understand microworlds at a base level, but as Treanor points out they halt at the level of *cultural knowledge* [261]. From an operational logics standpoint, an abstract game of circles and triangles where the triangles cause the circles to disappear is equivalent to the same game with rocks and jackhammers (or dinner plates and straws), as long as the collision logic is consistent with the appearance of the objects. While a player might

know that rocks are hard and jackhammers break rocks, they likely have no such relationship schema for plates and straws; they might have trouble explaining why it is that the straw breaks the plate, but they would not be confused as to the fact that it is the collision which causes the disappearance of the plate.

Unsuitable choices of graphical appearance can stand in the way of the logic's interpretation (for example, illegible fonts, inscrutable symbols, or character graphics which are difficult to distinguish or have poor contrast against the background), and to some extent this can be due to missing cultural knowledge; but in general operational logics do not consider whether, for example, a hazard looks dangerous or not. That said, a player might not think to attempt any interaction between straws and plates if they are not primed to suppose that such an interaction will be meaningful.

It is interesting to explore how far operational logics interpretations can be driven without considering cultural signifiers, or considering them only abstractly. A strength of the operational logics approach is that it makes no commitments to the form of this cultural knowledge, but gives a way to explain when and why audiovisual phenomena take place, what obligations they must fulfill in order to be interpretable as communicating particular logics, and how they relate to each other and to the simulated world of the game. Chapter 5 goes into more detail on this topic, grounding Treanor's proceduralist readings within the refined framework of operational logics devised in this part of the dissertation.

Chapter 3

Cataloging Operational Logics

The most commonly cited operational logics in the literature are collision, resource, and linking logics. Additionally, terms like *textual logics* and *graphical logics* were included in the first formulations of operational logics. Readers of earlier texts on operational logics may have concluded that these five terms comprise the whole universe of operational logics. This uncertainty can be resolved by stating that the purpose of *this* catalog is to identify broad families of closely related logics. "Collision logic" is not a single, monolithic entity, but a family of operational logics which all possess sufficient communicative roles, abstract operations, etc to function as collision logics. "Physics logics" could address momentum, gravity, wind-shear, or other physical behaviors; each pairing of the operation simulating that system with the human-visible phenomena which communicate it could be seen as a tiny operational logic in the family of physics logics, working in concert to form one specific larger physics logic (or, equivalently, the larger physics logic could be seen as comprising a superset of the abstract operations and game state presentations of the constituent physical behaviors). Likewise, "graphical logics" are

exactly those logics whose game state presentations are primarily graphical, and by convention refers to the composition of collision, physics, entity-state, and simple resource logics. [1]

Mateas and Wardrip-Fruin exemplify their definition of operational logics with a *random event resource management logic* which merely parameterizes a resource logic with random chance [166]; this seems to immediately open the door to an unbounded number of logics, all on equal footing with each other. This reduces the power of operational logics as an underpinning for game mechanics, because it seems that nearly every mechanic could be phrased as having a corresponding special-cased operational logic. It is more useful to separate the *random event* part from the *resource management* part, yielding a resource logic and a *chance logic;* this also removes the dependency on another specific logic's presence. While it is certainly possible to delineate an unbounded number of logics, I prefer to focus on *families* of logics, generally grouped by their abstract operations and communicative roles. While the number of *concrete* logics is limited only by human imagination, the number of *families* is potentially amenable to enumeration.

Guided by the utility of *graphical logics* as an umbrella term to characterize what is the same and what is different between arcade games and other games featuring colliding characters, I cataloged the main families of operational logics by examining particular games and genres [188]; the extant list of logics is given in Table 3.1 and the catalog at the time of writing is reproduced in Section 3.4. Note that this excludes, for now, an account of operational logics around *AI behaviors;* I explore these in Section 3.3.

[1] *Portions of this chapter originally appeared in "Refining Operational Logics" [198] and "Combat in Games" [192].*

Camera	Chance	Collision
Control	Entity-State	Game Mode
Linking	Persistence	Physics
Progression	Recombinatory	Resource
Selection	Spatial Matching	Temporal Matching

Table 3.1: The major families of operational logics

3.1 Building a catalog by explaining game phenomena

I assembled this catalog by playing and analyzing games, accounting for high-level concepts like *power-ups* or *turn scheduling* in terms of existing and candidate operational logics. I considered both the abstract operations that could be used to implement a concept and the communicative roles that must be fulfilled to obtain a satisfactory reading. This approach mirrors that taken in [192] to identify how models are built up from groups of operational logics' abstract operations.

I chose a set of around two dozen home console games from 1984-1998, with a few additions to include some rhythm games. For the initial set of logics in the catalog, I tried to address a breadth of loose (and admittedly ill-defined and overlapping) genre categories: action, fighting, puzzle, rhythm, role-playing, simulation, sports, and strategy. Games were selected for their notability and for diversity within genre groupings. The full list is available alongside the catalog [188]; these specific games were not intended to form any kind of canon or authoritative resource, only to act as a starting point for seeding the catalog. Generally, I identified logics by going from perception down to presumed abstract operations: reasoning from observed behavior towards an underlying process. As an example, I mirror the process I followed in [192] for *Street Fighter 2* and *Final Fantasy,* and also show such a decomposition

for *SimCity.*

3.1.1 *Street Fighter 2*

Fighting games focus nearly all their systems and instantial assets on presenting a martial (and supernatural) arts duel, and *Street Fighter 2* is an ur-example of the form.

My first objective when playing *Street Fighter 2* is to explain what the players see immediately when starting a match (Figure 3.1). First and foremost are the large and luxuriously animated fighters which move in continuous 2D space with the corresponding player's joystick (strong evidence for control and agency given by control logics, physics, and entity-state logics). These burly martial artists square off against each other from a respectful distance; their faces match the portraits seen at the character selection screen, and their poses make it clear that they intend to do battle, so one can be certain that the two players respectively control the characters they selected before. Brightness, size, and color contrast these fighters, and the ground on which they walk, against a background with which players cannot interact; this suggests a foreground and background for a collision logic with a ground plane. Although there are no visible walls or other obstacles in the stage, characters are prevented from moving beyond its left or right edge. I also see at the top of the screen two yellow bars with each fighter's name underneath: player one's bar on the left where player one's fighter stands, and player two's on the right.

A player may notice that walking away from her opponent does not change the direction her own fighter is facing. This entity-state logic cue gives the sense that the two are enemies and reinforces the interpretation that they are enacting a martial arts duel. Characters are given additional weight, and their adversarial relationship is further emphasized, by the fact

Figure 3.1: *Street Fighter 2 at the beginning of a match.*

that they cannot walk through each other (thanks to the same collision logic that provides the ground plane). The fighters may jump, and indeed may jump over each other—turning around to maintain eye contact in the process—which suggests a world where physics and entity-state logics control the movement of objects on the screen.

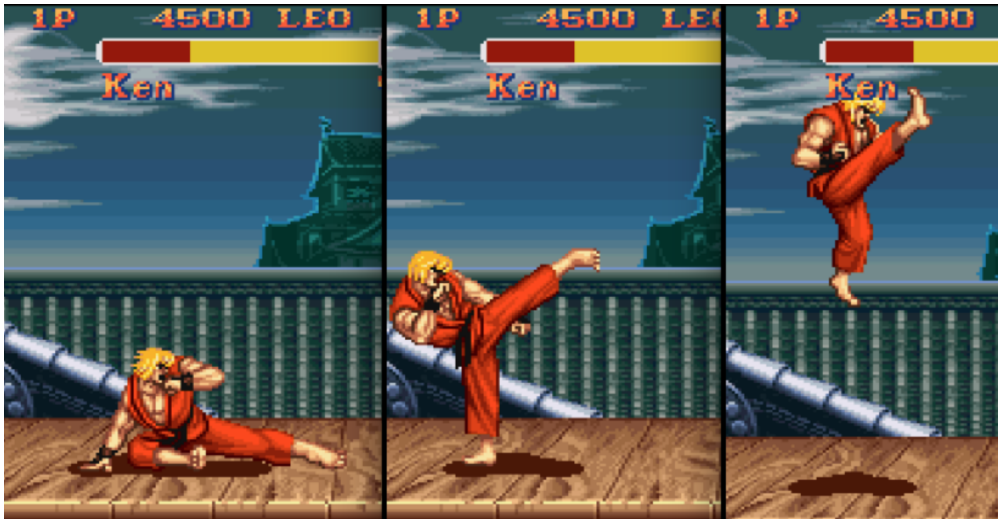

Figure 3.2: Attack state machines in *Street Fighter 2*.

Experimenting with the six buttons next to each joystick sends that player's fighter

into a flurry of animations which look like punches and kicks—the upper buttons punch and the lower buttons kick, connecting the physical controls to the characters' bodies through control and entity-state logics (as in Figure 3.2). Different fighters have different punches and kicks as appropriate for their size and martial arts school. If the attacker is close enough to their opponent during a punching or kicking animation, the latter plays an animation superficially indicating pain in response (the entity states evidently switch out the collision volumes of the characters through their different instantial assets), and one of the yellow bars shrinks, revealing a red bar underneath (Figure 3.3). I can therefore interpret this yellow bar as indicating a current value out of a maximum defined by the red bar: a graphical representation of a resource logic. While it does not use a numeric representation like the timer, I know that there is arithmetic happening somewhere!

Figure 3.3: Damage in *Street Fighter 2*.

The transactions of the resource logic are themselves triggered by the collision and entity-state logics associated with punches and kicks and, specifically, the reactions of the attacked characters. If this latter connection were absent—if punches and kicks did nothing, or if health decreased for no visible reason when buttons were pressed—it would be difficult to argue that the martial artists are fighting.

Further reinforcing the fiction of a fighter being injured, any attacks a combatant is performing when hit are canceled, and the player's buttons and joystick are non-responsive for a brief interval following each hit; in fact, characters also ignore joystick inputs while attacking. Something similar happens when two attacks of comparable strength connect with each other: both fighters are hurt, or else both attacks are neutralized without damaging either party. "Hit-stunned" and "attacking" are time-bounded states operating in entity-state logics, constraining the available inputs of the control logic and showing special animations to give impressions of kinesthetics, bodily weight, and even attack strength.

If a fighter is backing away from their opponent during an attack, they put up their arms to block (entering a blocking state), as in Figure 3.4. I know this functions as a block because when hit their health bar decreases less, the hit stun is shorter, and the defender is much less likely to be knocked flat out (resource and entity-state logics at work). The counter to blocking is throwing, activated by pressing a punch and kick button simultaneously (a "whole body" move). Blocking will not protect against throws, which is consistent with cultural knowledge that throws involve grappling rather than percussive force; throws also drop the thrown character to the ground.

Jumping and crouching (holding down on the joystick) can both be used to dodge

Figure 3.4: Blocking in *Street Fighter 2*.

attacks by their involvement with the collision logic. These distinct entity-states also change the function of every attack button. This is consistent with a notion of combat stances borrowed from martial arts, and gives more predictable access to the three heights at which characters are vulnerable: the feet, the body, and the head.

The *Street Fighter* series in particular is known for character-specific special moves. If a curious or well-informed player inputs joystick directions and attack buttons in certain pre-ordained, time-sensitive patterns (which vary from fighter to fighter), their fighter performs none of the individual actions entered, but rather plays a custom animation with an unusual combat behavior: flinging a fireball across the screen, flying across the stage like a torpedo, delivering a punishing spinning uppercut, or shocking adjacent opponents with electricity (Figure 3.5). This deployment of a temporal pattern matching logic makes it clear that the fighters are world-class martial artists extensively trained in secret techniques that transcend mere brawling with fists

Figure 3.5: Special moves in *Street Fighter 2*.

and feet. Only players who are both in the know *and* meet a threshold of manual dexterity can perform these moves (intentionally), just as the characters themselves have inherited the collected knowledge of their respective schools.

Between the two yellow and red bars at the top of the screen sit the letters 'KO' and a timer counting down. In the context of a martial arts fight, this plainly means "knockout" and the timer is a limit on the length of the match—I know that the fight will end either with one fighter knocked out or because the time runs out. The yellow bars shrink *towards* the bar, suggesting that, if a yellow bar empties, that fighter is out cold and the match is over. I also imagine that whoever has the longer yellow bar when time runs out wins by default, which is consistent with my understanding of technical wins in formal martial arts. This reading implies that the match functions as a state machine: sometimes the game is inside of a round, sometimes it is between rounds, and sometimes it concludes the set, and under various conditions (timeouts, knockouts,

Figure 3.6: The end of a match in *Street Fighter 2*.

and interstitial cutscenes) it transitions between these states (this is called a game mode logic). This is confirmed through play, and indeed neither player can move after the timer runs out. This ties the transitions of the match's state logic to the resource logic of the yellow bars, each of which I could call "health" or "stamina" by its relation to being knocked out.

To sum up, I see these logics at work in *Street Fighter 2:*

- Collision, physics, and entity-state logics suggest two dueling fighters in an arena
- Entity-state, physics, and collision logics help read movement as advancing, positioning, and dodging
- Collision, entity-state, and temporal pattern matching logics communicate physical, kinesthetic actions with different attacks using the body in different ways
- Collision and entity-state logics determine success of attacks

- Entity-state and resource logics work together to express fighters' reactions to being hit
- Game mode and resource logics structure matches

3.1.2 *Final Fantasy*

Combat, exploration, narrative progression, and character development comprise the four pillars of *Final Fantasy* and its successors; these are arranged by a game mode logic that situates each distinct set of operational logic engagements. In this chapter, I will focus on the game's battle scenes; I will explore several modes of its sequel *Final Fantasy IV* in chapter 7.

The early console role-playing game (RPG) *Final Fantasy* cemented genre conventions of party combat that were inspired by earlier games including tabletop RPGs. Whereas fighting games feature real-time physics, collision, and entity-state logics, combat in computer RPGs mainly takes place via resource and chance logics, often with turn-taking provided by control logics. Working game-by-game like this, I build up a list of ontological notions like collision, shops, characters, combat, and camera movement which is enacted differently in different games. I can also determine whether a particular notion is *fundamental* in the sense which is useful to the catalog.

As in my reading of *Street Fighter 2,* I begin by explaining the initial state of the combat screen (Figure 3.7). On the right I see the sprites of the four heroes selected by the player at the beginning of the game, arrayed vertically and all facing left; the first is slightly to the left of the others. They are on a black field surrounded by a white rectangle whose upper quarter has an environmental background appropriate to wherever the fight began (e.g., a forest or a dungeon). These cues suggest that the heroes are in the same space, and that it is connected

Figure 3.7: A combat scene in *Final Fantasy.*

to their position on the world map (in other words that I am looking in detail at a small portion of a large map, since the sprites are larger here than they were on the world map; this functions like a linking logic). Their rigid lineup amounts to a battle formation, and the first hero—stepping ahead of the rest—is in a distinguished position.

The screen has several other boxes, and by now I can recognize them as communicating something like overlapping windows from a game mode logic: some of them occlude the edges of others. Each of the windows to the right of the heroes contains the name of a character, the glyph "HP," and a number. These are laid out in the same vertical order as the heroes, suggesting (along with the names) that each is bound somehow to the corresponding character.

The largest window is on the left and contains sprites that vary from fight to fight. They are larger than the heroes and more intricately illustrated, always face right, and look like frightening monsters. These are all cultural cues implying threat and conflict. This box also

has the environmental features used in the heroes' window. This shows that the monsters are in the same general area as the heroes, but the window boundaries and width of the black fields suggest a substantial distance between the groups.

Below the monsters is a box with one or more words. Some experimentation shows that these vary with the monsters encountered and vanish when all the monsters with the same sprite are defeated, so over time the player understands them as the names of types of monsters. Tying the visibility of two graphical features together—the sprite and the name—is a basic cue for logics that communicate using graphical channels.

The last window shown when starting a fight is a grid of fantasy-combat actions: "Fight," "[Cast] Magic," and so on. A cartoon finger points at "Fight," and moving the directional pad moves the finger from word to word. By analogy to graphical adventure games, I can guess that some model of abstract commands is active here. The position of the hand—a symbol of agency—is tied by a selection logic to a choice of actions which will be enacted by some other logic (in this case, control, chance, and resource) after the selection is confirmed. Instead of LOOK or TAKE, my verbs are mostly violence against the enemies on the left. Now I will briefly examine each type of action.

Pressing the A button while "Fight" is selected moves the hand over to the enemies (Figure 3.8). The same logic that denoted selecting an action is now used to select a target. Pressing the B button moves the hand back, canceling the initial choice (a game mode logic at work). Pressing the A button also moves the hand back to the action list, but the first character moves backwards to the right and the second steps forward—but the "Fight" action has not yet taken place! This strongly suggests that I have merely committed to an action for the first hero

Figure 3.8: Selecting the "Fight" action in *Final Fantasy.*

and am now making a similar decision for the second hero. This impression is reinforced by the observation that pressing B now moves the second character back rightwards and the first left again. State and graphical logics combine to give form to the idea of selecting an action for each hero before resolving those actions, a convention shared with some pre-digital role playing games.

Selecting "Magic" causes another window to pop up and moves the hand there. This window overlaps the others, suggesting hierarchy: I am now choosing a specific spell such as "Fire" or "Cure." Not all heroes have the same choices here—the martial artist (recognized by his headband) knows no magic, and the spells of the white and black mage are disjoint (Figure 3.9). Individual spells are purchased and assigned to characters one by one, so even two black mages may have different options. These constitute a per-character magical inventory, engaging a resource logic where each hero "has" certain spells stored in the abstract space of

Figure 3.9: Magical "inventories" in *Final Fantasy.*

their memory.

Merely knowing a spell is not enough to select ("cast") it. Magic comes in tiered power levels; magic users can only cast spells of a given level a few times before they run out of "magic points" at that level and must replenish them by sleeping. These limitations on magic are given by a resource logic layered over the inventory of spells, surfaced by numbers at the left of each row (tier) of spells in the menu. Once a spell is chosen, it is directed—using the pointing hand—at an enemy (for single-target offensive spells), an ally (for single-target defensive or healing spells), or to the entire enemy or ally party (for group-targeted spells).

"Drink" gives a choice of healing potion from the party's shared inventory and uses it on the current character (Figure 3.10). This inventory has a set number of slots occupied by stackable and non-stackable items; stackable items may fit up to 99 to a slot (a resource logic with aggregated, commodified resources). "Item" permits the use of individual heroes' carried

Figure 3.10: Other action commands in *Final Fantasy.*

equipment as items in battle. Most have no effect but some function akin to spells, without the magic point restriction. Each hero has four slots for weapons and four slots for armor (another instance of a resource logic without aggregation). "Run" takes no target and indicates an intent to flee the battle on behalf of the entire party; each hero who tries to run gives the whole group a chance to escape.

Figure 3.11: Combat resolution in *Final Fantasy.*

After all the actions are selected, the combat resolution phase begins. Allies and enemies respectively (in a randomly determined order provided by a chance logic) animate and flash to indicate acting and reacting as more windows pop up over the screen—multiplexing

several logics over the communication channel provided in the game mode logic (Figure 3.11). The attack animations and hit reactions are a simpler version of what I see in *Street Fighter 2* and form a simplistic entity-state logic. These message windows appear in consistent locations, with one indicating the initiator of an action, another the (current) target, and still others showing the effects. During the fight, the numbers in the right hand windows will decrease when the corresponding heroes are attacked, communicating resource transactions over hit points (character health). It can be assumed that enemies have a similar resource, and when it is exhausted the enemy vanishes (and subsequent attacks against them are useless) or the hero collapses, unable to select or perform any actions. When an entire team is defeated (or the heroes successfully flee), the battle ends in victory or defeat—the game mode logic at work again (Figure 3.12).

Figure 3.12: Possible conclusions of combat in *Final Fantasy.*

An attack may miss, hit, or critically hit, determined by the relative statistics of the

attacker and defender; I read those outcomes textually and they are consistent with observed changes to characters' hit points. Basic attacks (the "Fight" command) can hit multiple times based on the implied speed and the accuracy statistic of the attacker: the dexterous ninja and master martial artist strike the most times, the knight hits fewer times but for more damage per strike, and the frail mages attack the slowest (here, the resource logic connects to cultural knowledge of physical prowess). The resulting damage depends on these statistics in yet different ways (and the specific attacks: for example, some spells hurt only undead monsters), and the statistics in turn depend on the equipment of the heroes involved. The systems of equations governing the resource logic of hit points are the central feature of RPG combat, and there are many branches, conditions, and modulations contingent on mostly-invisible numbers. The abstract and non-spatial modeling of strikes here and the substantial role of random numbers is a strong differentiator between combat in *Final Fantasy* and combat in *Street Fighter 2,* with the former being much less predictable and more chaotic.

Besides reducing hit points, a variety of other effects can occur in battle (mainly through spells or monster abilities). Hit points of living characters can be restored by healing spells or potions; a dead character can be revived (with a few hit points) by certain specialized spells. Characters can also be made stronger or faster; be prevented from acting by magical sleep, paralysis, or petrification; take gradual damage from poison; be prevented from casting spells by having their voices silenced; and gain resistances or immunities to specific elements. Some of these effects are resource augmentations and others belong to entity-state logic. The multiplicity of modifiers that combatants can obtain is another distinguishing feature of RPG combat.

In summary, these are the logics at work in *Final Fantasy* battles:

- A linking logic connects the abstract game world and the abstract battle arena
- Selection, game mode, and entity-state logics interoperate to describe the flow of combat in phases: selecting fantasy-themed actions and targets for each hero, and then resolving those actions in a loop
- Resource and entity-state logics constrain the space of possible actions
- Resource, chance, and entity-state logics determine actions' success
- Resource, chance, game mode, and entity-state logics calculate and communicate the effects of combat actions
- Game mode, entity-state, and resource logics structure combat as a whole, determining the final outcome

3.1.3 *SimCity*

Maxis's *SimCity* has been well-explored in the literature from both technical and political perspectives [134, 101]. In this section, I want to examine it from a phenomenological perspective as in the examples from combat above. What do I see coming out of the game, and what does that suggest about which logics are present? Here I focus on the Super Nintendo iteration of the game.

Once I am in the main play mode of *SimCity,* I observe a naturalistic terrain of dirt, forests, and water, as well as icons arranged as a menu palette; I also see a yellowish square

Figure 3.13: Cursor movement and actions in *SimCity.*

above the terrain (figure 3.13). The terrain suggests I have a simulated space in which collision between objects might play a role. The icons indicate that some selection logic is at work, since the picture of the bulldozer is highlighted yellow. The yellow square moves around when I press directional buttons (thanks to a control logic), and if I happen to move that cursor into the *menu* area it turns into a pointing finger; this is a key strategy used to indicate a selection logic of actions. Selecting a menu icon near the top of the screen has a different behavior, hiding the *tool palette* and switching into another type of menu; I'll explore more of these later.

After some experimentation I may press the confirm button on the controller and then some forests might be knocked down; this lends credence to the hypothesis that the icons (at least the icons in the *tool palette*) are indicating what action is currently selected, and that the cursor decides where to apply the action. When I *Bull Doze Area,* the number by the *$* indicator near the top of the screen decreases by one. I notice that bulldozing costs one dollar per bulldozed tile. The meaning of the other number—the zero with the person next to it—is yet unclear, as is the significance of the *RCI* bars. Both of these representations—decimal numbers next to icons and bars which grow or shrink—are typical of resource logics.

While the main function of the directional buttons is to control the cursor's movement,

Figure 3.14: Camera movement in *SimCity.*

pressing a button on the controller pops up a graphic of four arrows and an additional view on the world; in this mode, the directional buttons *scroll* the visible part of the world (figure 3.14), clearly indicating that a camera logic is at work.

Figure 3.15: Construction and development in *SimCity.*

The core of *SimCity* is building and developing a city, replacing the naturalistic terrain with a man-made one (figure 3.15). Using the toolbox palette to select different con-

structions, the player can place power plants, train tracks and roads, electrical wires, industrial/commercial/residential zones, and more. This uses up the money resource as I discovered during my bulldozing spree. Most buildings may not overlap (suggesting a collision logic at work), although there are some exceptions (electrical wires may cross roads and train tracks, for example). There are dozens of rules governing the behavior of the buildings and their effects in the game, but for a few key examples consider: buildings which are adjacent to either a powered building (or wire) or a power plant are powered; buildings are mutually reachable by transit if a transit building has an uninterrupted path between them; and so on.

Over time, several things happen: the date in the top-left corner changes, for one, and the industrial, commercial, and residential zones I built fill up with buildings. Commercial and industrial buildings take up an entire zone, while residential buildings begin as a perimeter around the central *R* (later turning into a full-block building like the others). As residential zones fill up, the number by the icon of the person—the town's population—increases. I discover through play that the RCI bars indicate need (or surplus) of the residential zones, commercial zones (representing internal trade), and industrial zones (representing external trade).

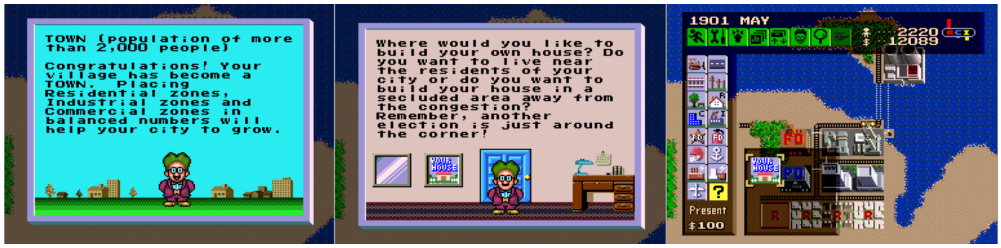

Figure 3.16: Progression logics at work in *SimCity*.

At a local level, *SimCity* is played in terms of its collision logics and resources like

money; but progress in the game hinges on reaching global population thresholds. For example, when the village reaches 2,000 citizens it earns the designation of *TOWN;* it will never be downgraded and moreover this transition earns the player the ability to place their own house in the game world (figure 3.16). This building can only be built once and has a few special properties. Most importantly for this discussion is that it (and other special buildings which are made available at different times) signifies the presence of a progression logic.

Figure 3.17: *SimCity's* diagnostic map.

In order to reach higher and higher thresholds, the player must build a dense metropolis balancing cost-of-living, crime, pollution, and other considerations. The player has several tools to accomplish this task, including a variety of maps which begin to reveal the essential nature of the simulation (figure 3.17). The pollution map, for example, shows how much smog there is in different parts of the city; these are localized mainly on industrial zones or coal power plants and spread, as do property values, crime, electricity, and every other quantity in *SimCity,* according to cellular automata rules. Cellular automata are a decentralized AI formalism where each cell has some quantities which it may transfer to adjacent cells according to particular rules

or under certain circumstances. As a model, they combine resource logics with linking logics (defining which cells are *neighbors* of a given cell) and are used widely in simulations.

Figure 3.18: Planning and menus in *SimCity*.

Moving up beyond the short-term decision-making provided by the maps, the player can consult public opinion to determine what problems are facing the town (figure 3.18). This menu (displayed, again, thanks to a game mode logic) summarizes some internal variables of the simulation's resource logics for the player's analysis. Finally, besides geographical considerations of what buildings to place where, the player is responsible for determining their city's fiscal policy: what the tax rate on citizens is and how much money should be allocated to transportation, police, and fire departments. Higher taxes depress population growth but make more money available for building, and decreased funding for utilities and public safety decrease their effectiveness and even cause roads and train tracks to decay over time.

To sum up, *SimCity* makes use of the following key operational logics:

- Camera logics show different parts of the game world;
- Control and selection logics give players tools to build with;

- Resource logics (together with linking logics) govern everything from taxation to population to pollution and crime;
- Collision logics prevent overlapping buildings and work as input to the automata rules (a case could also be made that these are spatial matching logics at work);
- Progression logics give the player goals to reach towards and rewards for reaching them;
- Game mode logics permit working with different aspects of the simulation's state via different interfaces;
- Persistence logics allow players to save their cities and come back another time.

3.2 Excluding candidate logics

Where possible, I tried to explain observed phenomena in terms of previously-known logics; if these explanations were awkward or did not reflect both the abstract process *and* the communicative role of the candidate logic, I acquiesced and formalized it as a new logic. While a logic or combination of logics can suffice to implement an abstract process of another logic, this can feel abstract and allegorical even in the best case. For example, a full resource pool inhibits the addition of new resources in a similar way as a collision logic inhibits the movement of objects into occupied space; but this blockage is hard to interpret successfully as *collision,* being more adequately interpreted as *saturation* or *exceeding capacity*. To detect a redundancy, fulfilling the abstract process is important; but I only have a truly redundant set if the communicative role—broadly construed—can be readily expressed by other logics.

It is not, however, the case that every notion one might wish to communicate to players has a corresponding operational logic. I set aside the primarily culturally-informed concepts which support interpretative frameworks such as proceduralist readings along with the complex playable models which are evidently built out of multiple logics: combat, for example, or cooking. After this step, I still had to apply some selection process to determine if a concept was or was not backed by a novel logic.

I wanted, as much as possible, a *parsimonious* catalog. I would prefer to work with a collision logic, but not a logic of space; a physics logic, but not a logic of driving or running; a resource logic, but not a logic of growth. While collision logics may be used to model space, and physics logics may be used to communicate running, and resource increase may be used to model personal growth, the latter set of concepts do not qualify as *primary* or *fundamental* logics: one can find other logics that model space, running, or growth equally well using completely different game state presentations and abstract processes. The "growth" of an object increasing in size is quite different from the "growth" of a character's strength statistic increasing. While one could imagine a category of *growth logics* which include collision logics, resource logics, linking logics, et cetera, this is best left as a grouping by possible interpretive affordances (similarly, *graphical* and *textual logics* are groupings of related logics).

Consider, for example, how *vision* works in three games: *The Legend of Zelda: A Link to the Past, DOOM,* and *Metal Gear Solid*. In the first and third, one generally sees a rectangular region around the avatar which stops at linked room boundaries; this region slides around as one moves, like a camera on a rig might. *DOOM,* on the other hand, provides a first-person view from the avatar's perspective; the camera does move along with the avatar's movements, but in

a much more direct and immediate way, as if it were hand-held or helmet-mounted.

These games model several distinct kinds of sight. Besides the rectangular area in view of the camera, *Zelda* has some dark rooms which are lit locally by the hero's lantern and globally by lighting wall or floor sconces; the former, local light functions as a kind of polygonal overlay, whereas the latter acts more like a timed resource whose quantity controls illumination. Moreover, enemies in *Zelda* tend to ignore Link until he comes close or enters a region of space just ahead of their current position and orientation. Enemies also have a kind of sight more akin to the lantern's light than to the panning camera. This type of vision is a defining characteristic of *Metal Gear Solid,* and avoiding being seen is a key consideration during play. Functionally, what an enemy can see (or not) is more like *DOOM's* perspective camera than it is like the panning camera used in scrolling through the space of a level.

It is clear that vision is a concept that games can and do model in a variety of ways. One can imagine many implementations (colliding with vision volumes, being on the other side of a closed or open door, being in shadows or in a spotlight, drawing or not drawing a sprite) and ways to communicate that status (changes in character behavior or background music). There is, however, a key distinction to be drawn between, on the one hand, the communicative role of indicating that agents may or may not see other agents in particular situations; and on the other hand, communicating to a player that *they* are seeing part of a larger environment.

I could hypothesize a single *vision* logic, but I have seen that this is too open-ended and does not have one clear communicative role. Then, why not have both a *vision* logic and a *camera* logic, separating out the role of showing who can see what and the role of showing the player a sub-region of the level? Neither candidate logic depends significantly on cultural

knowledge, which is promising; but the former is readily explained in terms of e.g., collision or resource logics, while the latter does not have such a clean decomposition. A moving camera could be expressed, in some cases, as a stage which is moving around under the lens; but it is strange to think of the *DOOM* avatar as a stationary object in a rapidly-moving jumble of walls. I am therefore forced to posit a *camera logic,* one of whose uses might even be the modeling of vision (e.g., in a multiplayer game).

To generalize from this example, I ask two questions when attempting to explain a modeled concept and determine whether it justifies the introduction of a new operational logic: First, does this candidate logic have a distinctive role? Second, does the candidate logic admit a usefully concrete implementation? Splitting out cameras from other types of vision is an example of distinguishing based on the first question; but what does it mean to ask if a candidate logic has an implementable or usefully concrete abstract process?

Games often have *scheduling:* for example, the turn order of a role-playing game or the alternation between players in a turn-taking game. But the abstract process this suggests—"determine who goes next and whose turn it is, and give the current player control"—is at once too generic and too specific. A good operational logic has a process which is abstract but puts constraints on possible implementations, and at the same time does not over-constrain its set of abstract operations and limit the contexts in which the logic can be used (recall that abstract operations describe the sorts of game state transitions that the logic enables). A scheduling logic phrased as above excludes the possibility of simultaneous turns, and if it expands any further to incorporate that it morphs into a version of the more general *control logic* ("different entities are controlled by different inputs at different times"). Scheduling is not an operational logic;

instead it is generally (but not necessarily) built up from resource and control logics.

For an example of applying both considerations, consider menus in games. Their abstract process is "select from some possibly hierarchical options and trigger actions based on those options" and their communicative role is that the player has many options of which they may choose one. These are already so close to each other that something seems amiss; moreover, the candidate *menu logic* seems to step on both *selection logics* (governing which of a set of items is currently designated as selected) and control logics, overlapping with them while also overly specializing them to the case of menus. Games also have many different kinds of menus: I could press a keyboard key to activate a menu item in one, or move a cursor to select and then press a key to confirm in another, or move the player character into one or another doorway or region of space to activate a menu item in a third.

This breadth of implementations (in some cases engaging totally different logics) shows that menus are not likely to have their own operational logic; or, rather, that the explanatory power of menu logics is poor. The candidate "menu logic" fails to have a distinctive role.

These questions also allow for distinguishing, for example, game-mode logics (characterized by state machines) from entity-state logics (also characterized by state machines). Although their abstract processes are similar in the sense that both transition some entity through a state transition system, they have completely distinct communicative roles.

3.3 Cataloging AI Logics

While operational logics readily describe how a game communicates simulated phenomena to players, it is harder to account for the complete behaviors of complex enemies. Something like *Super Mario's* Koopa Troopa which changes direction when bumping into a wall is easy to explain in terms of collision and physics logics as a kind of law of nature, but the behavior of *Pac-Man's* ghosts is trickier. Some seek out a point in front of or behind Pac-Man, some try to stay away from Pac-Man; pathfinding is awkward to situate in the treatment of operational logics shown so far.

On the other hand, Wardrip-Fruin's work points to notions of *navigation logics* or *volition logics,* through which AI agents decide where to go and what to do in a way that players understand. In the past I have felt that these are more appropriately considered as *playable models* of decision-making, but it is certainly true that games have particular conventions for communicating AI decision-making to players and this might point towards a set of families of AI logics, akin to recombinatory logics.

AI design and game design are notoriously difficult to tease apart, but in my work I have found it useful to consider the role of operational logics as specifying what an agent *might* do, while external controllers like players or AI brains determine what the agent *will* do. In part this is because, eventually, AI behavior is arbitrary machine code. While higher-level descriptions of character behavior may be possible (for example, as temporal logic formulae, behavior trees, or state machines), game AI is often extremely *ad hoc* and game specific. The actuators and sensors used by AI agents must certainly be definable in terms of operational

logics, and the remaining question is whether some analogue of control logic is appropriate for AI *brains.*

For now, I suspect a similar approach to the one described in this chapter must be applied to the behaviors of AI characters. I believe that logic families like *pathfinding logic* and *ranked-choice logic* are certain to turn up through such an investigation. If it is the case that entity-state logics suffice to address character state machines and even formalisms like behavior trees, pathfinding and ranked-choice logics should give a good account of the nondeterminism that for a player would be resolved via control logic. So, the Goomba is a simple machine without any need for choice-ranking or pathfinding; the *Pac-Man* ghost might add pathfinding but makes no dynamic choices about prioritizing one action or another; and the *Prom Week* character does choice ranking in the context of a resource logic but engages no other logics in particular. All of these are communicated to the player and become part of the game they play, manipulating the pathing of enemies or forcing them to oscillate between choices as a matter of strategy.

3.4 The complete catalog

With a rationale, a process for gathering examples of logics, and criteria for inclusion and exclusion laid out, all that remains is to construct and show a catalog of operational logics. This is just one such catalog and it is sliced along the axis of *communicative role;* one can imagine other catalogs focusing on game state presentation, communication channels, or abstract processes, or on the logics which enable certain mechanics or which are used towards

certain aesthetic or interpretive aims. I believe this approach strikes a useful balance between too many and too few logics and that the way it is organized admits effective use in compositional game definition, game interpretation, and the automated analysis of games.

3.4.1 Camera Logics

3.4.1.1 Communicative role

What the player sees is one of potentially several viewports onto a game world, and our viewport can pan, zoom, rotate, etc. On-screen and off-screen parts of the world might act differently, and the player might see several viewports at once.

3.4.1.2 Abstract process

Moving or altering the shape or position of a camera's projection; mapping this onto other surfaces, e.g., a sub-region of the game screen.

3.4.1.3 Abstract operations

- Alter the world position of the camera or other aspects of its transform (e.g., pan, zoom, rotate)
  - Potentially performing some camera wipe or other transition
- Choose whether or not to draw certain objects on certain cameras

3.4.1.4 Presentation

- Show the viewport on the game screen where the player can see it

- Project the viewport onto something in world space

3.4.1.5 Required concepts

- World coordinate space

3.4.1.6 Provided concepts

- Camera views

3.4.2 Chance Logics

3.4.2.1 Communicative role

Events can play out in unexpected (“random”) ways.

3.4.2.2 Abstract process

Sampling from stationary or nonstationary event sources, along with analysis of probabilities, expected values, etc.

3.4.2.3 Abstract operations

- Check/alter the likelihood of an event occurring.
- Cause an event to happen (perhaps from a pool of events) with some likelihood.
- Make a random draw from a stationary or nonstationary source

3.4.2.4 Presentation

- Show probabilities like success rates as textual percentages or as discrete symbols summarizing their probability
- Show expected values as ranges between minimum and maximum possible values
- Highlight extremely unlikely outcomes with visual effects
- For nonstationary sampling as in sampling without replacement (e.g., from a shuffled deck of cards), show the upcoming picks or the remaining size of the bag

3.4.2.5 Required concepts

- Types of random events

3.4.2.6 Provided concepts

- Chances, for example success rates or expected damage value

3.4.3 Collision Logics

3.4.3.1 Communicative role

An illusion of physical space is provided by the fact that some game objects occlude the movement of others.

3.4.3.2 Abstract process

Detection of overlaps between subsets of entities and/or regions of space and the automatic triggering of reactions when such overlaps occur.

3.4.3.3 Abstract operations

- (Alter/Check) which entities could be said to collide with which other entities or regions of space (e.g., collision layers or flags, or hurt vs hit boxes)
- Check if two entities or regions are overlapping or touching
- (Alter/Check) the region of space taken up by an entity (e.g., grow or shrink an entity or change its collision polygon)
- Enumerate the entities or regions overlapping or touching an entity or region
- Separate the positions of two or more entities or regions such that they touch but do not intersect
- Determine how far an entity could move towards another entity or region without touching or causing an overlap
- Whenever a combination of the above checks becomes true (or remains true, or becomes false), perform some abstract operation of this or another operational logic involving the objects considered in those checks

3.4.3.4 Presentation

- Shapes or images for each entity or region of space, whose extents and appearance correspond to the extents of the corresponding entity, and whose positions on screen correspond to their positions in space
- Sound effects or visual effects indicating collision when particular objects or re- gions of space begin/continue/cease to touch or overlap
- Textual descriptors detailing the sizes or shapes of entities and whether they are touching or overlapping
- The presence or absence of textual descriptors of entities when describing a region of space
- Textual messages describing the event when objects begin/continue/cease to touch or overlap

3.4.3.5 Required concepts

- Entities and their positions
- Space

3.4.3.6 Provided concepts

- Collision
- Overlapping

- The extents of objects in space

3.4.4 Control Logics

3.4.4.1 Communicative role

Different entities are controlled by different inputs at different times.

3.4.4.2 Abstract process

Map button inputs, AI intentions, network socket messages, etc onto high-level game actions, possibly grouped onto specific loci of control like characters according to the event source.

3.4.4.3 Abstract operations

- Change which character the player (or another event source) is controlling
- Change the mapping between low-level inputs and high-level actions
- Determine the available high-level actions at a given time for a given actor

3.4.4.4 Presentation

- Show a graphical indicator like a triangle or colored circle around the character being controlled (perhaps mapping colors to event sources).
- List the available moves for an actor in a menu overlay.
- Show a picture of e.g., a keyboard or a controller with labeled lines illustrating which buttons perform which high-level actions.

3.4.4.5 Required concepts

- High-level actions
- Event sources/actors
- Loci of control (e.g., game entities)

3.4.4.6 Provided concepts

- Available actions for a given actor
- Which actor controls which locus
- Which actors are present/active

3.4.5 Entity-State Logics

3.4.5.1 Communicative role

Game entities act in different ways or have different capabilities at different times, intrinsic to each such entity.

3.4.5.2 Abstract process

Governing the finite, discrete states of a set of game characters or other entities. Updating states when necessary. Conditioning the operators of other logics on entities' discrete states.

3.4.5.3 Abstract operations

- (Alter/Check) the discrete state of an entity according to a given (fixed) transition system.

- Whenever an entity's discrete state changes, perform some operators of this or another logic.
- Synchronize state changes between several entities.

3.4.5.4 Presentation

- Change an entity's sprite, animation, or visual effects according to state.
- Textual or icon labels/descriptors to indicate state.
- Audio/visual/text effects on state change.

3.4.5.5 Required concepts

- Entities
- Condition predicates for making transitions

3.4.5.6 Provided concepts

- Entity states
- State transitions and related events

3.4.5.7 Notes

Note that an entity may be attached to several concurrent transition systems, and these must be composed somehow. In this document, I use "discrete state" to refer to an entity's overall,

combined state, e.g., “jumping while ducking” or “poisoned, hasted, and confused”. Concurrent Hierarchical State Machines provide one possible semantics.

Also note that resource logics may also involve discrete state (e.g., number of health units), but this seems somewhat ambiguous with entity-states (e.g., current set of power-ups; in a Metroid example, whether the Varia suit upgrade has been obtained). The key guidelines for whether such a piece of state is a resource or an entity-state are:

1. If the statistic “feels” more categorical than numeric (even if an ordering could be given), entity-state is likely more appropriate.

2. If the statistic is numeric and is ever “spent”, or if it acts as a cap on a numeric resource which is spent, or if it otherwise engages with a resource economy in a similar way, it is likely a resource and not an entity-state.

In particular, RPG character statistics are usually best thought of as resources, whereas status effects are more like entity-states.

3.4.6 Game Mode Logics

3.4.6.1 Communicative role

The game switches between different screens in which different controls, information, and actions are available.

3.4.6.2 Abstract process

Set up, activate, suspend, and clean up different game modes (potentially in parallel) like title screens, save/load screens, navigation versus menu or battle modes, etc.

3.4.6.3 Abstract operations

- Activate a new game mode, optionally deactivating the old mode
- Determine the currently active game modes

3.4.6.4 Presentation

- Change the visual layout of the screen, the background, or some text to illustrate the active mode
- Play a transition visual or sound effect to highlight a change in mode

3.4.6.5 Required concepts

- A set of game modes and conditions under which they might change

3.4.6.6 Provided concepts

- Which game modes are currently active

3.4.6.7 Notes

This will often work in concert with persistence logic so that while the player is in the in-game menu or pause screen nothing changes in the world.

3.4.7 Linking Logics

3.4.7.1 Communicative role

Some things, in some context, are related or connected to each other.

3.4.7.2 Abstract process

Maintain, enumerate, and follow/activate directed connections between concepts.

3.4.7.3 Abstract operations

- Determine/alter which links are available potentially according to operators of other logics
- Perform some action when a link is activated or traversed
- Enumerate or perform some action on all the objects linked to another object

3.4.7.4 Presentation

- Show a laid-out graph of links between concepts.
- Highlight links which are available to traverse/activate or hide those which are unavailable
- Scroll smoothly when moving between connected spaces, if connections have a “side”
- Play a screen transition when moving between connected spaces

3.4.7.5 Required concepts

- Things to be linked together
- Types of links and attributes of those links

3.4.7.6 Provided concepts

- Networks of linked objects
- Current "positions" in those networks

3.4.7.7 Notes

This can be used for spatial links between rooms, for dialog trees (navigating an abstract space), for character progression systems like Final Fantasy X's Sphere Grid, and for a wide variety of other uses.

3.4.8 Persistence Logics

3.4.8.1 Communicative role

Some things in the world stay the same, for example across sessions, between rooms, or when scrolled offscreen; while others change or reset.

3.4.8.2 Abstract process

Determining which entities, assets, and variables are in a particular scope (e.g., saved-game scope, current-level scope, visible-area scope) and persisting or resetting those as the scope

changes.

3.4.8.3 Abstract operations

- Determine or alter which persistence scopes are active.
- Set, clear, or restore the facts associated with a particular scope.
- Trigger an action of another logic when scopes change, are saved, or are restored.

3.4.8.4 Presentation

- Give audiovisual feedback when scopes change or when variables have been reset.
- When the player is about to leave a scope, give textual descriptions of what data will be preserved and what will be lost.

3.4.8.5 Required concepts

- Abstract objects and facts governed by scoping rules.
- A set of possible scopes.

3.4.8.6 Provided concepts

- Active and inactive scopes
- Save and reset events

3.4.8.7 Notes

This is generally how a game addresses concepts like “bullets disappear when they go off-screen” or “remember opened treasure chests and defeated bosses but not regular enemy health/positions”.

3.4.9 Physics Logics

3.4.9.1 Communicative role

Physical laws govern the movement of some in-game entities.

3.4.9.2 Abstract process

Update and honor objects’ physical properties like position, velocity, density, etc., according to physical laws integrated over time.

3.4.9.3 Abstract operations

- Alter the physical properties of objects (mass, moment of inertia, coefficients of friction, etc).
- Apply impulses, continuous forces, heat, etc., to objects.
- Integrate physics forward (or rewind it backwards) for one or more objects.
- Determine an object’s physical quantities based on terms from some other logic (e.g., character-state, resource).

3.4.9.4 Presentation

- Map physical quantities like energy or mass onto visual qualities of the objects
- Show changes in physical quantities with visual or sound effects

3.4.9.5 Required concepts

- Game entities, including their positions, shapes, and extents
- Physical properties of entities and the world

3.4.9.6 Provided concepts

- Physical quantities and behaviors

3.4.9.7 Notes

This is nearly always in simulated continuous time and space.

3.4.10 Progression Logics

3.4.10.1 Communicative role

The content of a game and its behavior can change as the player makes progress through various sequences of goals.

3.4.10.2 Abstract process

Track and check measures of progress (potentially on multiple fronts).

3.4.10.3 Abstract operations

- Update or reset progression on some front, or track some flag.
- Check whether the player has progressed past some threshold, or whether something should be available.
- Parameterize operations of other logics based on progression.

3.4.10.4 Presentation

- Pop up an achievement or other notification (e.g., a title card) when game progress takes place
- Show story events in a logbook or as a network of events
- Write text so that non-player characters refer to earlier stages of progression

3.4.10.5 Required concepts

- Types of progress

3.4.10.6 Provided concepts

- Current status on progress measures

3.4.10.7 Notes

This can be used for things like quests and chains of quests, for in-engine cutscenes and scripted events, and to gate some content based on other content.

3.4.11 Recombinatory Logics

3.4.11.1 Communicative role

Some things are built out of smaller (recognizable) things.

3.4.11.2 Abstract process

From a fixed set of (possibly parameterized) atoms, build a structured object (a room, a dungeon, a description, a magic item) according to some rules and constraints (possibly conditioned on operators of other logics) with some optional post-processing.

3.4.11.3 Abstract operations

- Generate an object or behavior out of some set of atoms with some parameters
- Enumerate valid outputs for some given constraints

3.4.11.4 Presentation

- For text which is filled in a "mad libs" style, underline or otherwise highlight the dynamic part of the text
- Show a seed value identifying the generated object

3.4.11.5 Required concepts

- Atoms to be recombined
- Conditions governing valid combinations

3.4.11.6 Provided concepts

- Object/behavior generators
- Parameters and measures associated with generated objects

3.4.11.7 Notes

These can be used for procedural content generation or for monster formations or even for monster combat scripting, text generation, etc.

Diablo rooms come from a fixed set and are connected arbitrarily through doorways; Rogue rooms come from a fixed set and are connected by gluing passages between them; Markov-chain text uses fixed characters and chooses the next according to the current character and some probabilities encoded in a Bayes network; Eliza produces a new string by picking a template based on whether key words were found in the input string by a pattern matching logic, with the templates parameterized by rooted or transformed versions of the input to the pattern matching logic.

These can roughly be characterized in terms of how complex the choice structure is, how many or how fine-grained the intermediate structures produced by the operators are, and how much post-processing is done to ensure consistency or otherwise "fix up" the generated stuff. I suspect that constraint-solver-based methods would live in here too.

3.4.12 Resource Logics

3.4.12.1 Communicative role

Generic or specific resources can be created, destroyed, converted, or transferred between abstract or concrete locations.

3.4.12.2 Abstract process

Creating, destroying, and exchanging (usually) discrete quantities of generic or specific resources in or between abstract or concrete locations on demand or over time; triggering other actions when these transactions take place.

3.4.12.3 Abstract operations

- Check if a transaction involving some units of certain resources is possible for some given locations.
- Perform a resource transaction.
- Perform an action of another logic when a transaction occurs.
- Perform an action of another logic when the quantity of resources within a location crosses a threshold.

3.4.12.4 Communicative strategies

- Resource types are icons or text, quantities are numeric labels, locations are distinct regions of the screen (Or, in textual logics, separate lines or sections of the text)

- Resource types and quantities have intertwined representation, e.g., ten units of resource R appear as ten identical grouped icons or text labels communicating R; again, distinct regions of the screen express location (Or, in textual logics, separate lines or sections of the text)
  - This refers to e.g., Zelda “hearts” if the “capacity” of the location is also shown; this could also apply to health bars or other indicators where the length or color of a bar is a proxy for resource quantity
- As in the previous case, but different quantities of the same resource map onto different iconic representations
- Transactions may be described as lists of prerequisite resources and quantities (as above) and lists of result resources and quantities (again as above, but not necessarily in the same way). The interface elements affording the execution of that transaction may be dimmed if the transaction is not allowed, or the individual prerequisite resources if they are not present, or the result resource indicators may be dimmed. (Where dimming may be broadly construed as affording non-interaction)
- An attempt to perform a disallowed transaction may be met with graphical, textual, or aural feedback
- When a transaction occurs, text labels indicating the resource and quantity (via color, size, text content, etc) may appear for a time on screen (perhaps near the locations the transaction took place) or more permanently in a log, possibly along with aural feedback

or animations

- Locations with spatial semantics may be communicated as appropriate for those semantics, with resources assigned individually or in groups to "physical" objects in those spaces. Common examples include "inventories" in adventure and especially role playing games. In any event the communication is somewhat deferred to the other logics which provide the spatial model; often this spatial view is not used in (for example) crafting systems of the same game, where aggregate counts are used instead

3.4.12.5 Required concepts

- Resource types
- Locations for resources
- A way to enumerate and count resources in those locations, e.g., to map a Diablo inventory onto a crafting system
- Ways to remove resources from and add resources to locations

3.4.12.6 Provided concepts

- Resource transactions (available, allowed, and disallowed)
- Resource quantities and thresholds

3.4.12.7 Notes

In many cases there will be only one location, or resources of a given type will only be created, etc. The generic/specific language used here distinguishes e.g., "Minerals" from having more than one "Copper Sword." The two types of locations are ones which cannot be physically located in the game (abstract) and those which can (concrete).

Cellular automata are a special case, especially since they are often communicated by mapping cells/locations onto physical space in a metric way, and quantities onto a dimension of space/color/etc

3.4.13 Selection Logics

3.4.13.1 Communicative role

The player can mark a subset of items out of a larger set as "selected", and there might be several currently active selections with different semantics.

3.4.13.2 Abstract process

Remember sets of selected objects out of larger sets, and pass around or enumerate those sets for other logics.

3.4.13.3 Abstract operations

- Add or remove an object from a selection
- List the objects in a selection, or which could be selected

- Perform some action when objects are selected or deselected
- Constrain the maximum or minimum size of a selection

3.4.13.4 Presentation

- Highlight the selected item graphically, by putting an icon next to it, or by altering or annotating its text label
- When a selection changes, play a visual or sound effect or have the selected/deselected item play an animation

3.4.13.5 Required concepts

- Things that can be selected and types of selections

3.4.13.6 Provided concepts

- Selected object identifiers and measures on selection sets

3.4.13.7 Notes

These are often used for menus or for RTS control (interacting with control logic), but also show up in crafting systems and many other places. Sometimes there are several selection categories which can be active at once, e.g., “the item slot the player selected for ‘from’ and the item slot the player is about to select for ‘to”’.

3.4.14 Spatial Matching Logics

3.4.14.1 Communicative role

Arranging objects in certain ways can cause things to happen.

3.4.14.2 Abstract process

Analyze the positions of objects and trigger events when they enter or leave particular layout patterns.

3.4.14.3 Abstract operations

- Determine whether a set of objects is arranged in a particular pattern
- Perform some action when a pattern is matched
- “Click” or automatically align objects into a particular arrangement

3.4.14.4 Presentation

- Highlight objects which partially or fully satisfy a pattern with visual effects

3.4.14.5 Required concepts

- Things to be analyzed for pattern matching
- A space in which those things are laid out

3.4.14.6 Provided concepts

- Matched patterns
- Slots in which objects can be arranged

3.4.14.7 Notes

In a point-and-click or parser adventure game, situations like “use HAMMER on ROCK” or “put FISH in OCEAN” could be seen as cases of spatial matching. This also covers puzzle games like Tetris or Bejeweled as well as certain puzzles in RPG dungeons and potentially the minigames seen in e.g., golf games.

Some crafting systems also use spatial pattern matching logics.

3.4.15 Temporal Matching Logics

3.4.15.1 Communicative role

Some sequences of events can have different effects when they occur with particular timing or in a particular order.

3.4.15.2 Abstract process

Observe sequences of events to recognize when a pattern is matched and trigger an action when this occurs.

3.4.15.3 Abstract operations

- Trigger an event when a temporal pattern is matched

- Determine which, if any, temporal patterns are currently under consideration/partially matched
- Determine how well a new event matches any pattern currently under consideration

3.4.15.4 Presentation

- Give the player feedback on how well their input lines up with the temporal pattern
- Provide visual or audio feedback when a temporal pattern is satisfied
- Show the sequence of actions under consideration to the player in a textual or pictorial representation

3.4.15.5 Required concepts

- Types of events to be matched
- Streams of events

3.4.15.6 Provided concepts

- Matched patterns
- Timing quality

3.4.15.7 Notes

Music/rhythm games are often based on temporal pattern-matching logics, as are time-sensitive mini-games and special moves in fighting games.

Chapter 4

Composing Operational Logics

At least since the first distinction was drawn between collision and resource logics, Wardrip-Fruin and others have contemplated the question of whether and how multiple logics in a game fit together. Mateas and Wardrip-Fruin wrote that there were low- and high-level operational logics, with the latter being strictly built out of the former [166], and asserted that game mechanics and game systems were built from such simple and compound operational logics. Treanor *et al.* explicitly distinguished logics from rhetorical argumentation and illustrated how the latter could be justified in terms of the former [259]. My collaborators and I showed an example of how multiple logics could form a playable model when taken together [192]. These works established *that* logics compose and *for what purposes,* but not *in what ways.* Since my view of operational logics relies on strong boundaries between logics, it is important to explore in detail how distinct logics relate to each other and how they fit together. I have argued that operational logics do not combine in a simple hierarchy; in fact, they compose in a variety of cross-cutting, overlapping, highly contextual, and complex ways [198]. In this chapter I ex-

plore that line of reasoning more deeply and deploy it to explain a variety of concepts from game studies and game ontology. [1]

4.1 Three Types of Composition

There are three main points of contact or sharing between logics: communication channels, operational integration, and structural synthesis. Every composition of multiple logics (e.g., a game, genre, or game-making tool) must define the semantics of each of these interfaces. Even if the same logics are deployed, different choices for this glue can result in very different games.

For example, a side-scrolling beat-em-up may use a collision logic to determine damage, an entity-state logic to determine character behaviors, and a resource logic to track the health of each character. Here, the display of characters' health bars over their heads (as opposed to putting them in corners of the screen) is a shared *communication channel* (the character's embodied sprite); the triggering of resource transactions in response to collisions is an *operational integration;* and the precise, frame-by-frame data-flow and mapping between a character's position in the world, its current entity-state, and the positions of its hit-boxes and hurt-boxes (damaging and vulnerable collision areas) is a natural example of *structural synthesis.*

The composition of operational logics opens up four key questions:

1. How do the logics fulfill their communicative role, i.e., how do they enact their game state presentation?

[1] *Portions of this chapter originally appeared in "Refining Operational Logics" [198].*

2. Where logics overlap and depend on each other, often in cyclic ways, how is this condition expressed in the rules and resolved at runtime?

3. What is the game state, exactly, and how is that initialized and managed across time?

4. When logics demand different parts of the game state, or have different sorts of inputs and outputs or world models, how is this mapping done?

Question one is addressed by communication channels; the second is the subject of operational integration. The last two concerns are the domain of structural synthesis.

4.1.1 Communication Channels

The simplest way for logics to overlap is by sharing *communication channels.* This can be seen as a kind of semantic multiplexing, where affordances offered by one logic can be leveraged by another. Of course, operational logics must be defined independently of their possible relations to other logics, so it falls on the composition of logics to define how one logic's communicative role and game state presentation strategy are fulfilled in the context of another's.

Games with distinct characters or sprites, e.g., from entity-state or collision logics, commonly share the space around the sprite as a channel. Regardless of how the sprites are positioned in the world, additional regions of the screen might be allocated for individual characters (for example the status displays in a fighting or role-playing game). Channels can also be duplicated or parameterized: games with split-screen multiplayer provide several distinct copies of the same configuration of channels (one for each player). Ultimately, what pixels get drawn

to the screen (or which tokens are positioned around a board) has to be defined somewhere, and that display has to incorporate information from every logic. This type of composition is always present, even if the individual logics interact in no other ways.

Compared to operational logics, graphical and textual design are relatively well-understood. All the laws of composition and information design apply, and a complete discussion of how a designer may choose to enact the game state presentation of a logic is outside the scope of this project. Suffice it to say that each logic has its own universe of discourse—the colliders of a collision logic, the resource quantities and types of a resource logic—and it falls to designers and programmers to decide upon and realize a game's eventual sensory output, consistent with the communicative roles and game state presentation of the involved operational logics. These two aspects of operational logics are a bridge to these other, well-established disciplines.

4.1.2 Operational Integration

Sometimes, an operational logic may need to make a decision based on a fact determined in another logic, or it may need to trigger an action drawn from its own operators or those of another logic. Examples include a character who is injured when collisions occur, an enemy that changes behavior when the player comes close, or a resource transaction which is only available when a specially timed sequence of button inputs occurs. I call this kind of composition *operational* in the sense that it has to do with combining the abstract operations of several logics.

The logics of the catalog in Chapter 3 include, in their abstract operations, phrases

like "trigger an operation of this or another logic." These are explicit sites of composition where logics can integrate their operators together.

One way to think about this type of composition is that every logic defines a set of logical predicates and a set of actions: questions that can be asked about the logic's view of the game world and changes in the world that this logic is competent to make. Operational integrations are a type of composition that can be defined almost exclusively at this level, mostly agnostic of the underlying implementation of the integration. While their concrete semantics—how data about character positions are used to determine whether character behavior changes should take place—are up to the implementation, they at least *have* a clear implementation-independent abstract semantics. Communication channels and structural synthesis, on the other hand, admit much less space between the implementation and the specification.

This operational integration might be implemented by a blackboard where the programmatic systems implementing logics write their calculated facts and desired actions, through object-oriented messaging and observation, via forward-chaining inference, or by an event propagation mechanism. If the game specifies, for example, an ordering among logics—that movements are resolved before collisions, and the collision handling code is processed before determining the applications of a spatial pattern matching logic—then shared variables could suffice to implement the operational integration.

4.1.3 Structural Synthesis

I have accounted for how operational logics' concepts are presented to players via (possibly shared) communication channels and how mechanics can be built out of several logics

through operational integrations. But I have glossed over how concepts like characters and space—which are required and provided by a variety of logics—are eventually resolved and unified. This is different from referring to a black-box predicate (are two characters touching?) or triggering an opaque action (perform a resource transaction): an entity-state logic and a physics logic need to somehow refer to the same set of characters. *Structural synthesis* defines how game state flows between operational logics, how it is mapped between their different internal representations, and so on.

In the catalog entries of the preceding chapter, one pair of fields has so far gone unexplained: required and provided concepts. This is a kind of ontological free variable, a placeholder for terms which can't be defined within the logic itself or which could be converted into the native terms of other logics (in the style of modular action languages [156]). Consider, for example, graphical logic games comprising collision, physics, entity-state, and resource logics. A *game character* is not just a collision volume, nor just a point mass, nor just a state machine, nor just a resource pool. The same character *identity* has to be consistent across all of these logics. The physics logic repositions the point mass subject to the constraints of the collider, and with velocity and acceleration determined by the state machine; if a projectile hits the collider, the corresponding resource pool must have health resources removed; and so on. Some of these integrations are operational, but all assume an external *mapping* to place them into the same conceptual framework.

Crafting systems in games are often designed as simple resource transactions: take three of item A and combine them with two of item B to make one of item C. When inventories are also described in terms of item types and quantities, the mapping seems natural: a move-

ment from one resource pool to another. On the other hand, many games feature inventories with a spatial embedding—multiple packs and satchels, with items taking up different numbers and arrangements of slots, where items may form stacks of bounded size—implicating e.g., a collision or linking logic alongside resource logics in the determination of what items are available for crafting. In cases like these, the resource logic behind the crafting system must have a way of counting resources of a given type in a given abstract location, and a way of inserting any newly created or transferred resources into an abstract location. The resource transaction should be disallowed, for example, if the pack has no room for an object with the product's shape. This mapping is (and must be) specific to the concrete composition in question, and it is a question of structural data-flow.

The key distinction between a structural synthesis and an operational integration is that from the player's perspective, structural syntheses introduce new *atomic units* or new ontological concepts, whereas operational integrations reuse and recombine those concepts in ways which are evident to players. In the crafting system example, the idea that the supply of an ingredient is intrinsically connected to the number of items in the inventory is a structural synthesis, whereas using up an item (like a recipe) to learn a new crafting transaction takes place via an operational integration—especially if there are other ways to learn recipes such as completing quests or gaining character levels.

How is the game state (opaque and inscrutable to individual logics and operators) projected out onto the terms of a particular logic or set of operations? And how are the effects of these operations merged back in to the global state? What is the game state, exactly, and how is it set up and managed across time? Structural synthesis provide *invariants* and govern these

flows of game state, as opposed to operational integrations which make use of and trigger some of these flows and changes.

These determinations must be made both in game engines and in individual games. If a character moves too far from an active enemy, that enemy might *despawn* or reset; from an operational integration standpoint it does not matter whether the enemy continues to exist in the world, but the actual game state might reuse the memory associated with that enemy for some other purpose. If persistence logics are used to determine what characters might be active for the purpose of low-level collision checking or state updating, that is an example of structural synthesis; so is the use of a linking logic to help determine which region of 2D space a camera is scrolling over. Operational integrations and communication channel sharing are, eventually, built on top of this structural synthesis. Chapter 7 explores the question of *defining* games in terms of operational logics in depth.

4.2 Reading games with operational logics

The preceding discussion mainly addresses the authorial affordances [165] of operational logics: How game designers can deploy logics to rhetorical or aesthetic ends, building up complex rule systems and communicative structures to explain those rules. Games, however, do not exist *per se* until they are played. A key strength of operational logics is that it simultaneously addresses both how designers conceive of a game and how players understand that game in play. I have already seen several examples of reading game phenomena through the lens of operational logics. Chapter 3 illustrates this approach in depth for the purposes of

cataloguing logics. At this point I can formalize the intuition behind how operational logics provide interpretive affordances to both human and automated players.

The underlying cognitive and behavioral assumptions behind operational logics are not validated by experiments or studies. Instead, one can use operational logics as a possible *model* of how players understand games and explore the consequences and predictions of this model; this dissertation stops at that point. It remains important future work to design and conduct studies capable of validating the cognitive assumptions behind the theory, but I hope that this at least establishes what a suitable ground theory of operational logics perception might need to satisfy. Operational logics do, at least, present a testable theory, which is a significant difference from many other approaches to game ontology.

The general shape of operational logics interpretation begins with the player's perception of audiovisual, tactile, or other sensory phenomena over a variety of communication channels. A player attempts to assemble these observations into coherent objects based on considerations like juxtaposition, textural differences, and animation; this stage elides a longer discussion of visual perception and literacies, and a complete discussion is outside the scope of this dissertation.

After the player successfully demultiplexes the shared communications channels, this account resumes at the process of learning possible explanations (for example, causes and effects) for these objects' behaviors. At this stage, a player's awareness of existing operational logics comes into play, informed by commonsense reasoning: noticing that two objects seem to occlude each others' movement, or that an animation that suggests an injury is concurrent with a reduction in a number with a label like *health* or *hit points.* This is an ontology fitting

problem that is the player-side dual of structural synthesis; it is generalized further in Chapter 6. One possible explanation is that players perform some relational learning in schema which are provided by the individual operational logics: for example, that a numerical resource could in some way reflect the number of a given type of object in the world.

Finally, the player lifts these structural observations into the realm of operational integrations by determining, through experimentation and observation, how logics are engaged to enact and explain game rules. This might take place through forming and testing theories of possible game rules in terms of the expected activations of operational logics.

While the above account is largely speculative, it does offer some promising directions for validating the underlying assumptions of OL theory. We can test the theory by presenting games which we would suppose have similar logics and determining whether players agree; whether changing the visual appearance of colliders in a way that OL theory would predict ruins the interpretation of collision indeed does so; or whether players with varying levels of game literacy are capable of understanding low-level cause-and-effect relationships under different configurations of operational logics. More importantly for my purposes it is naturally mechanizable for projects in automated game understanding (Chapter 13).

4.3 Game Ontology Revisited

With these definitions in hand, one is equipped to explain a variety of notions in game studies in terms of compositions of operational logics. This exercise both validates the framework given above and provides a solid foundation for concepts that have traditionally been

only loosely defined. In the remainder of this chapter I survey a variety of ways games have been organized and described in the past; in the following chapters 5 and 6 I examine in detail how operational logics has supported two particular ways of *interpreting* games.

4.3.1 Rules

Because the subject of what constitutes a *game* has produced myriad dissertations on its own, I will focus here on the discussion of formal elements which seem to recur in various definitions of *games.* This does assume that a game is an activity with formally defined rules or other intentional elements, but I believe the discussion here should be portable to a variety of definitions of games. Notably, operational logics are not restricted to games: they should function similarly in any interactive form. Moreover they are not limited to digital or computational media. Physical games (such as board games) certainly define their own processes and have their own techniques for communicating their inner workings, but these can leverage the physical world as an implementation substrate whereas digital games must implement every aspect down to the level of putting pixels on a screen. In general operational logics theory is most robustly explored in the domain of digital games, and that will be the focus of this section as well.

Salen and Zimmerman synthesize eight definitions of games like so: "A game is a *system* in which *players* engage in an *artificial conflict,* defined by *rules,* that results in a *quantifiable outcome*." [211] Their term *system* itself comprises *objects, attributes, internal relationships* among objects, and an *environment.* They use system to include idealized formal systems, experiential systems activated by interaction between (for example) players, and

cultural systems; I will focus on the first category here.

System objects are defined, in an operational logics regime, by the entities which respective logics address: collision and physics logics require some simulated bodies, entity-state logics need characters to which state machines are attached, resource logics work on specified resources and pools, and so on. These are synchronized and fit together via structural synthesis. System object attributes are also implied (or imposed) by the logics and the way the designer has arranged, for example, the starting conditions of the game or the specific operational integrations that enact the game mechanics. Salen and Zimmerman describe internal relationships as something akin to a current world state, which is defined in terms of the ontology given by the available logics and their structural synthesis (a fuller exploration of game state will be given in 7). Finally, the environment in which the system objects interact is the game in play—the dynamic behaviors of the given logics, exposed along the communicative channels engaged in the work.

With *system* out of the way, I can now develop the operational logics connection to the remainder of Salen and Zimmerman's definition of *games.* Leaving *rules* for last, I can address the elements of their definition like so: I have already asserted that *players* exist just outside of the domain of discourse of operational logics, perceiving their operations through the communicative channels and hypothesizing specific structural syntheses and operational integrations to guide their play. Operational logics thereby provide the bridge between the reality of players and the *artifice* of the game. *Conflict* can be enacted in games in a variety of ways, and emerges from the interactions of operational logics with expectations about what players or other agents ought to do to achieve goals (as described in [242]). *Quantifiable outcomes* come in

several forms, depending on whether they represent quantifiable progress towards a goal or the final conclusion of a game. Like conflict, these can appear naturally from resource, collision, or other logics and the dynamics their arrangement (and interaction with the player) induces; outcomes which end the game or switch it into different interaction modes are generally encoded in *game mode logics.*

That leaves *rules,* "the structure out of which play emerges." [211] Salen and Zimmerman initially distinguish the formal rules from the audiovisual or other thematic elements that communicate these rules to players—for example, asserting that altering the suits on a deck of cards would yield the same rules but a different experience of play—but explain that this "creates an artificial separation between the structure of a game and players' experience of the structure." Rules in this sense are similar to *mechanics,* left for the next section. They immediately resolve this uncomfortable separation by describing three *kinds* of rules: operational, constituative (the "formal rules" from before), and implicit rules. Operational logics, conveniently, are used to define (via operational integrations) the *operational* rules, which lie above the realm of formal mathematical equivalences (the domain of *constituative* rules, where *Tic-Tac-Toe* is the same as *3-to-15* in which players take turns picking numbers trying to form a sum of 15). In digital games, Salen and Zimmerman explicitly connect the internal behaviors to external indications of those behaviors in operational rules, making an important step beyond the conventional framing of game mechanics. *Implicit* rules, on the other hand, are outside the domain of operational logics; these are mainly social rules of good conduct implied by manners and the physical necessities of playing the game.

4.3.2 Game Mechanics

I have already shown that game mechanics are built out of and communicated through pieces provided by operational logics, most often through operational integration and communication channel sharing. Two mechanics like "Lose ten health when you touch a wall" or "Regain full health when you touch a powerup" share a similar template: "When you touch something, something else might happen." This is characteristic of the operational integration of collision and resource logics. Addressing the essentially infinite space of possible mechanics by carefully combining from a fixed set of operational logics gives valuable constraints on automated game analysis, game generation, and indeed game design itself. Moreover, describing mechanics in terms of operational logics admits a more nuanced analysis of games' constitutive rules than using mechanics directly.

In many cases, game mechanics are actually relatively complex, specifically-situated constructs. Even the names used for mechanics often point in quite different directions, depending on the logics that support the mechanics and the models within which they are situated. For example, consider the mechanic of "jumping." In a game such as Christopher Strachey's 1951 checkers game *M.U.C. Draughts* (arguably the first video game) the jumping mechanic is quite different from that in a game like David Crane's 1982 *Pitfall!* In Strachey's checkers game space is implemented (as is traditional with a checkers board) as a set of fixed, non-overlapping spaces. This is wedded with an implementation of time (again, as is traditional in checkers) divided into alternating, non-overlapping turns. This combination of discrete space and time is common not only in video game emulations of traditional board and card games, but also in

genres such as strategy games, simulation games, and pattern-matching games. In Strachey's game the jumping mechanic enables a move, on one's turn, from one discrete space to another, ignoring the intervening space.

Pitfall, on the other hand, encodes space and time in ways players experience as continuous, rather than discrete, as in many of the most influential video games. In such games the player's experience is that she can take arbitrary positions (in the reachable areas of the game space) and usually move at any time she is playing (rather than waiting for a discrete turn). These games use collision, physics, and entity-state logics, in the context of continuous time and space, to support a rather different mechanic: jumping as an activity that the player character can engage in at any time he is not already in the act (the state) of jumping, with the player being uncertain whether the jump will reach the area of continuous-seeming space for which she aims.

Operational logics give a different way to think about and analyze game mechanics, to better capture similarities and tease out differences across games. Through this lens, it is clear that in many ways *Zelda 2* and *Bionic Commando* are highly similar games, as are *Metroid* and *Super Mario Bros.*—each pairing has the property that a game engine for one readily admits constructing the other. This may also give a more useful way to address concepts like game genre or game systems.

4.4 Conclusion

A primary motivation for using operational logics in this work was their compositionality. This chapter has explored how operational logics compose and what one gains by speaking in terms of their composition: a *lingua franca* for discussing a broad variety of concepts in and around games with a common grammar. While operational logics do stop short of making claims about certain aspects of play (player modeling, AI behavior, and cultural knowledge) at least their interface with these areas is narrow and well-understood. If it is indeed the case that everything happening *inside* the game, with the exception of perhaps AI, is due to operational logics, this provides a tool with considerable analytical power—and, as it turns out, a special utility for *defining* games as well (which I explore in Chapter 7).

Chapter 5

In Depth: Proceduralist Readings

Proceduralist readings are a manual proof technique for determining whether a given *interpretation* of a game or game phenomenon is supported both by its instantial assets (e.g., images and sound effects) and by the rules of its simulation (i.e., the behavior of game entities) [259]. A *meaning derivation,* the product of a proceduralist reading, is a tree of observations drawn from witnessed game phenomena and a reader's cultural knowledge grouped together and linked by conclusions regarding a hypothesized underlying process or rhetorical claim.

Proceduralist readings have been rooted in operational logics from their initial conception, and were developed specifically to address questions of procedural rhetoric and communication in graphical logic games. While the extant treatments of proceduralist readings have evident ontological commitments to operational logics theory, this chapter expands on and further formalizes the foundation of proceduralist readings both to exercise and help to validate this dissertation's development of operational logics.

Assume a game with two objects (A and B), and if A ever collides with B then B is

removed from the screen. What does this interaction mean? In other words, how can a player interpret it and assign a semantics to the observation that B disappears when it touches A? Is A *eating* B? Is A *picking up* B? Is B *hiding from* A? It depends in large part on what A and B look like. If A is a human face and B is a hamburger, then an interpretation of eating seems likely. If A is a torpedo and B is a submarine, then the player might say A is *destroying* B; Fig. 5.1 shows a meaning derivation justifying this claim. [1] If A is a circular saw and B is a plank of wood, the player could accept that A is destroying B but might be surprised not to see two smaller planks of wood afterwards.

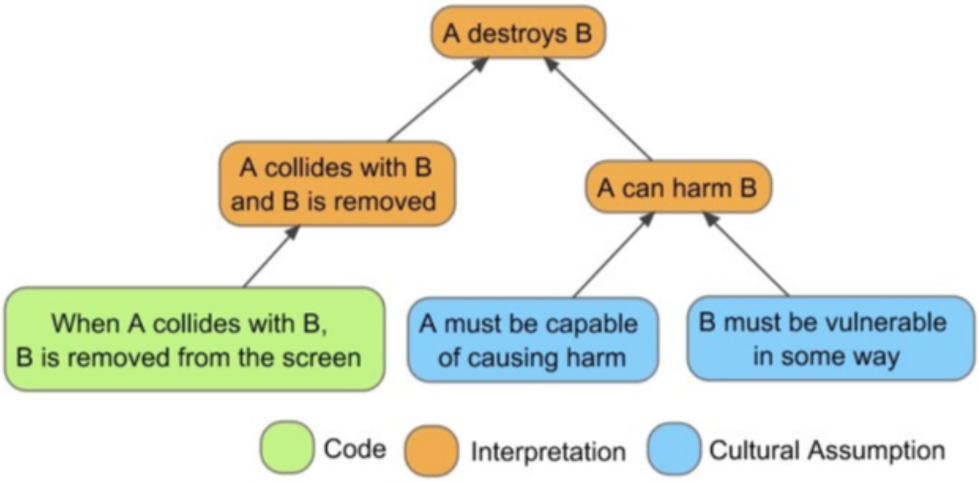

Figure 5.1: A reading of "A destroys B", reproduced from [261].

Proceduralist readings are mechanizable in the sense that two people can, given the same set of assumptions, come up with the same conclusions or at least agree on whether a reading is reasonable (an opportunity formalized in [164]). But this process's inputs are not

[1] Mark Nelson's work on semantic microgame generation is another important precursor, and he also uses a graphical notation—though his is centered on conceptual relationships rather than a tree of inferences [180].

very well defined. What defines the set of possible interpretations? What refines the set of applicable cultural assumptions? How does one go up the tree from low-level observations to higher-level assemblages of how things work and how they look? Operational logics yield a satisfactory answer to some of these questions, and their boundaries show where more work remains to be done.

5.1 Proceduralist Readings via Operational Logics

In the preceding chapters I have shown several examples of causally interpreting game phenomena as interactions between operational logics. Through proceduralist readings I can leverage this technique to interpret human-scale concepts that lie in the realm of cultural knowledge. Treanor's own work has showcased graphical logics-based readings of games [258], so in this chapter I focus on the deployment of other logics. In general, operational logics provide the domain of predicates to be used in readings (in his example, *CollisionEval(A,B)*) while cultural knowledge extends these predicates with interpretations (e.g., that it is in and of itself bad for *money* to collide with *fire*). As I have shown in my work on automated game design learning [242], a system can automatically draw conclusions about certain interactions being good or bad *with respect to winning the game,* but this is different from the implications provided by cultural knowledge.

Spatial pattern-matching logics, like collision logics, govern the spatial relationships between objects. Unlike collision logics, these can trigger actions when objects enter particular spatial relationships: three-in-a-row, connected-within-some-tolerance, line-of-sight, and so on.

Droqen's *Block Faker* is in many ways like *Sokoban,* with an avatar colliding with boxes and pushing them around (see Figure 5.2). There are two key differences: first, the level is won when the avatar enters a green square; second, and more importantly, lining up three or more of the same-colored block in a horizontal or vertical row eliminates the row. This unambiguously engages spatial pattern-matching logics (whereas regular *Sokoban* is mainly a collision logic game).

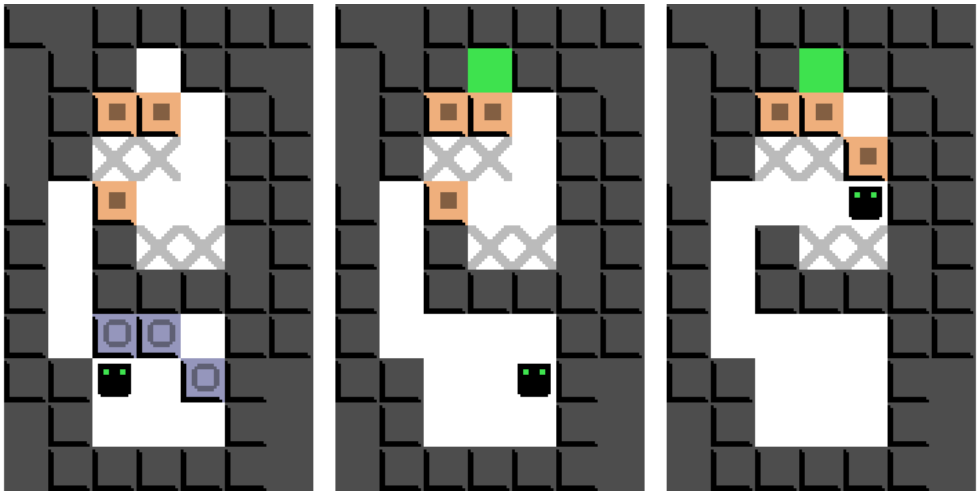

Figure 5.2: *Block Faker,* a game featuring collision and spatial pattern-matching logics.

Making a match, broadly interpreted, is the primary operator of spatial pattern-matching logics. Throughout his publications, Treanor's approach to proceduralist readings is generally written at this abstract level [260, 258, 259]: any type of collision between objects is collapsed into one predicate determining its valence (Summerville et al have a more nuanced interpretation [242] but the simpler one serves fine for this example). The possible interpretations for matching a cross-shaped block of objects versus a horizontal or vertical line are quite similar to those of collision, in the sense that any particular type of match can be good for the player or

bad for the player. In *Block Faker,* the result of matching is that the matched objects disappear; this can be helpful if the objects were blocking the player's path, or unhelpful if the player were trying to bring one of the matched blocks somewhere else. *Block Faker* is an abstract puzzle game; if its art assets were more representational, I could bring in cultural knowledge in the same way as Treanor did in his study of BurgerTime [257]: if the matched objects seemed like ingredients and the match caused a food item to appear, then I could argue that cooking was taking place. As it is, I can only say that matching can be good or bad, that the black object is an avatar, that the green square is a goal, and that reaching the goal is good.

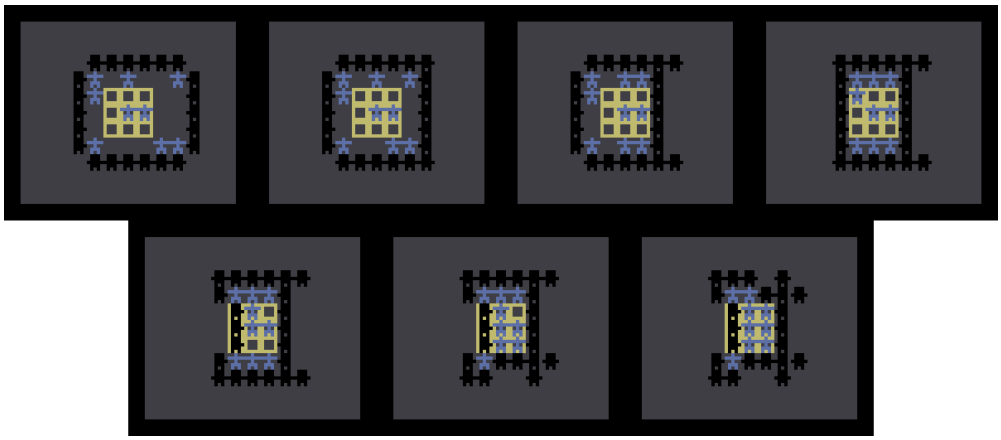

Figure 5.3: Stephen Lavelle's *Kettle,* a game featuring collision, sophisticated control, and spatial pattern-matching logics.

In *Kettle,* the player controls the black-clad riot police around the perimeter of the board. Pressing left moves the police along the right edge leftwards by one step if possible, pushing blue protestors in the same direction if they are not blocked by police on the other side; pressing up moves the police along the bottom edge upwards according to the same rules; and so on (see Fig. 5.3). When all the protestors are lined up on all the yellow squares, the game is

won. Police never move backwards, so the game is effectively over if any police officer is on a yellow square.

Setting aside cultural knowledge for a moment and looking purely at the game's formal systems, one might infer that matching up blue characters with yellow squares is good (since it leads to winning the game). The player also understands that they control all of the black-colored characters (so they are sympathetic to the player) and that matching a black-colored character on a yellow square is bad. The consequences when objects collide are either that some objects are moved along with the mover or that the mover is prevented from moving, and the latter has a negative valence in the circumstance where the player had hoped the particular moving object should indeed have moved. This is as far as a pure operational logics reading can go, although it is clear where the *hooks* on cultural knowledge might be hung could be: the game's name refers to a controversial police tactic; it is therefore evident which characters are police and which are protestors, and while the player knows that they control the police it does not seem good for the protestors to be grouped together (in contrast, if the blue characters represented sheep and the dark characters were shepherds, this at least gives the reading of *protection*). Cultural knowledge can act as a tie-breaker which disambiguates possible interpretations of relationships between objects.

5.2 Cultural Knowledge and Learnability

Just as cultural knowledge refines the possible interpretations of a game, the specific ways operational logics are used in a game put *constraints* on the ways that cultural knowledge

can be used to make proceduralist readings arguments.

As I said in Chapter 2, "Unsuitable choices of graphical appearance can stand in the way of the logic's interpretation." If A touching B causes A to lose health, then B should look dissimilar to an object C that causes A to gain health. A game where some blue spikes hurt the player and other blue spikes heal them would be extremely difficult to learn to play. Even if two objects are visually distinguishable, cultural knowledge around their appearance must also suggest a reasonable probability of behaving differently from each other. It would seem wrong for green spikes to harm the player and blue spikes to heal the player; there is no way to interpret these relationships besides referencing these specific abstract and arbitrary color choices. If the green spikes were hamburgers and the blue spikes were vegetables, one could potentially make an argument about vegetarianism, and there would at least be some modicum of consistency to the readings if the theme of *meat-eating is bad* were maintained throughout the game.

Games that aim to be learnable and interpretable should support players in developing mental models of the game rules so that players can predict what will happen when performing a new interaction. Chapter 6 expands on this in greater detail.

5.3 Reasonable Readings

Treanor briefly describes the *validity* of a reading as a measure of its *consistency:* a reading is less valid if it ignores contradictory information (e.g., hypothesized game rules, observed phenomena, or cultural knowledge) and it is more valid if it addresses more of the game situation [259]. Elsewhere he characterizes a reasonable reading as avoiding large interpretive

leaps that, likewise, fail to address important aspects of the game experience.

While fully accounting for the size of an interpretive leap is outside the bounds of operational logics, having an enumerated set of logics and their operators at least gives a set of possible low- and mid-level inferences and information about how these might engage with cultural knowledge. This allows for calculating valid interpretations of game rules and dynamics and verifying whether the higher order interpretations given by human readers conflict with any of these mechanically derivable readings. In fact, at least some readings can be performed automatically, as shown in the Gemini game generator project [164].

Gemini's goal is to allow non-designers to control, in a constraint satisfaction framework, what possible interpretations a game must (or must not) have and what mechanics it must (or must not) have. A key area where this is useful is in games about controversial topics—the guarantees afforded by constraint satisfaction (in particular, answer set programming) allow a system to both generate games widely and ensure that the generated games do not accidentally contradict the designer's own viewpoint.

Gemini uses human-provided inference rules to determine whether, for example, the player controls some in-game outcome or whether two entities can harm each other (this reading is, again, modulo cultural knowledge). Gemini game rules are all operational integrations of collision, physics, resource, and control logics, with any given rule mixing and matching some number of operators from each logic. The logics are supported in the underlying game language, although they are not reified *per se:* there are possible preconditions and postconditions involving, say, resources, but no notion of resource logics proper.

Moreover, no changes to the communication channel sharing or structural synthesis

of these logics are possible; Gemini generates games with entities spawned from a small set of (generated) types, resource pools that are either independent from or connected to specific entity instances, and a mapping between resources and colors that may be painted onto the game's stage by entities. Two of the three axes of compositionality across operational logics are therefore fixed, and the third (operational integrations) is implicit in the mechanism for generating rules. One consequence of this is that readings (e.g., that X has a goal of doing Y) must be written in redundant forms: once, for example, for the case where Y is a goal of increasing a resource, and again for the case where Y is a goal of increasing the amount of a color drawn on the screen—even though the structural synthesis of resources and color should yield this automatically.

Some interpretations certainly require multiple rules along different lines of reasoning (X harms Y if X reduces Y's health, or if X immediately causes Y to be deleted, or if X causes Y to bounce backwards violently, or perhaps if X reduces a resource Y needs to achieve its goal), but at a certain level of abstraction these should eventually be able to share some common structure. At the very least, an explicit account of logics, their operators, and the way they can be combined would yield a way to check the coverage of a reading ("there is no way to infer harm in the absence of resource logics, is that as you expected?") as well as possible ways to improve generated games in the future. If Gemini had the notion of feedback, for example, which is vital to operational logics but mostly absent from the system as-is, it could ensure that every rule is effectively communicated to players across one or more channels, and then this communication could be the foundation of readings rather than a priori knowledge of the game's rules. In chapter 8, I explore a version of Gemini that makes explicit the sites of

operational logic composition and the operators available to each logic.

Proceduralist readings, grounded implicitly in operational logics, have inspired several successful projects in game generation and interpretation. This chapter has reframed proceduralist readings in terms of this dissertation's theory development, and in the process has opened new opportunities for improvement in systems like Gemini. While addressing cultural knowledge (and interpretative leaps) remains important future work, this chapter neatly ties up the lower-level aspects of proceduralist interpretation. Since allowed inference steps on meaning diagrams can be seen as axioms of operational logics theories, it is clear how to extend proceduralist readings to address additional logics and game phenomena in a clean and orthogonal way.

Chapter 6

In Depth: Playable Models

The game studies part of this dissertation addresses three main tools for analyzing and designing games: operational logics, proceduralist readings, and playable models. Of these, playable model theory has been explored only obliquely in the literature, being confined mainly to a handful of slides from various presentations given by my advisor Michael Mateas. I made an attempt at pinning down what exactly playable models are in my paper on combat in games [192]; the present chapter refines this explanation and situates it a little more cleanly with respect to proceduralist readings. [1]

Playable model theory draws from Mateas's earlier work on agency and expressive AI [165, 270]. *Model* here is used in two senses: first, a computational system can *model* some real-world phenomenon or system. This use of model is used to mean a simulation or other simplified version of a complex process. The second sense in which playable model theory refers to models is in the sense of *mental model:* the interactor's beliefs or hypotheses about

[1] *Portions of this chapter originally appeared in "Refining Operational Logics" [198] and "Combat in Games" [192].*

the way in which the computational model (or simulation) operates, or the interactor's beliefs about the real-world phenomenon or system being simulated. *Playable models* differ from general computational models in that they support the interactive and gradual exploration of a simulation model so that an interactor can develop a mental model which provides them with effective agency in the system. Playable models also differ from games in general because they ought to allow a player to *transfer in* knowledge the player already possesses and, potentially, admit transferring *out* new knowledge produced in play.

By way of example, a Turing machine simulator which takes a text file as input is not a playable model of computation on its own, even though it enacts a model of computation. To become playable it must be paired with mechanisms to, for example, visualize the machine's operation and history, to support the writing of stored programs within the simulation itself, to illustrate mappings between high level programming languages and Turing machine representations. The use of the system in the context of the larger interactive work should be capable of teaching the system's principles.

From the other direction, a game in which the player can hold a button to make their avatar move faster could be *read* as having running in the sense of the previous chapter. But it would not function as a playable model of running since running is not simulated at a suitable level of fidelity to admit transfer in of a player's knowledge of running (besides that it is faster than walking), nor to admit transfer out (there is no clear analogue of the *go faster* button on one's body). If running in this hypothetical game used up a renewable resource with a name like *fatigue,* the model would be slightly more playable; but without incorporating aspects of physicality into the interaction it is difficult to see how a game could faithfully model running.

Bennett Foddy's *QWOP,* which could be said to model the biomechanics of running via physics logics, is an interesting example here—in a sense it requires the player to train their fingers to *run* using what they know about human locomotion. On the other hand, a game could very effectively model track-and-field *coaching* or *training;* these map naturally onto optimization problems and one can imagine a player skilled at track-and-field coaching would have an advantage in playing such a game, or at least that they would know what should be possible and how things ought to work; such a player would also be in an excellent position to *critique* the game's simulation.

6.1 Building Playable Models

How can a designer construct a playable model for some real-world system? How can they ensure that they produce a system which both models a concept and makes that model playable? Simulations are built out of rules, and these rules are enacted by computation; but playability requires that these rules are communicated in a way that helps players understand the operations of the system. From an authorial standpoint, operational logics provide the building blocks of readable rules, and for the player their literacies in operational logics give them a *ladder* which they can climb to understand the model.

Players interact with a game by perceiving its output phenomena and providing inputs on various devices (e.g., controllers). In order to know what to do next, what the game's goals are, what the player's own goals are, and in general in order to participate in the player-game system, the player must *simulate* the game's operation in their own mind: they must be able to

make informed guesses about the system's operation. The version of the game rules that the player holds in their head is called their *mental model* of the game.

There is a tension between playable models and proceduralist readings. Every model a player holds *is* a reading—that the game is *about* something and that thing works in a certain way. As established above, not every reading indicates a truly playable model, and in fact not every reading about a game is correct with respect to the game's true rules.

Playable models describe abstract processes similarly to operational logics, but at a much higher level of abstraction—these specifications have too many possible implementations. This may be because they refer to cultural knowledge (as in the case of *cooking* or *romance*) or it may be because they could be realized by a variety of combinations of operational logics (as in what Wardrip-Fruin calls *navigation,* which could be obtained through a discrete linking logic or through physics and collision logics in concert). A model can be satisfied or induced by many distinct arrangements of operational logics with instantial assets and interpretive affordances that even come from other models; but this always terminates eventually in the operations of specific logics and in instantial assets. In order to participate in a game effectively at all, players must be aware of what operational logics are present. What distinguishes playable models from proceduralist readings is that not all readings will improve a player's performance at a game, whereas acquiring the model makes a player *better* at the goals the game suggests.

Another way of discussing playable models is to say that the player proceeds through a process of *progressive refinement* of increasingly more precise and accurate readings until they synchronize their own mental model with the model encoded in the system. A provisional interpretation or reading might be inconsistent with the game's rules or inconsistent with the

real-world system, but it suggests to the player certain interactions to try out to learn more in a targeted way and resolve these inconsistencies. This is the key element that makes playable models *playable:* not only that they are comprehensible, but that incomplete or incorrect understandings suggest natural experiments to perform or observations to make that would improve the player's understanding (or give the player a way to critique the simulation in relation to the modeled concept).

How do playable models offer this support for experimentation and learning? Operational logics give a framework for players to make sense of the game's low level operations and to compare the actual effects of an in-game action with what the player expected should have happened. The player can then use their powers of perception, lateral thinking, existing knowledge, or good guessing to come up with explanations for the discrepancy. The trivial interpretation of some in-game action is that its outcome is *random;* if a game does not communicate its internal operations well enough, or if the player is not paying attention to the right things, this is the only possible interpretation. As a player tries out alternatives, or repeats a trial, they might come to learn which aspects of the interaction or of the surrounding game state have an influence on the outcome—but they can only do this if the state is presented to the player somehow (even if indirectly).

As a thought experiment to pursue this idea further, let's imagine two players of a game: the *perfect player* who has perfect perception of game state, perfect ability to execute intended actions using the input devices, and a perfect mental model of the game's rules; and the *uninformed player* who provides random inputs at random times, ignoring the system's output completely. Certainly, the uninformed player is incapable of understanding playable models and

thus no model is playable for them. To the perfect player, every cause and effect relationship is immediately apprehended; their deep knowledge of computational systems means that even when the game is trying to appear random by using a pseudo-random number generator, the player recognizes the algorithm at work and the seed value and anticipates the result perfectly. To this player, all simulations are playable in the sense that the player can form and execute plans—even a weather simulation spreadsheet—but the fine grained level of detail prevents accepting *abstraction* which is necessary to attach cultural knowledge to the system's behavior and thus produce a playable *model.* Put another way, if the weather in the game randomly either rains or doesn't rain with some probability, this roughly comports with a human player's experience with weather in a commonsense way; but if the weather is always rainy if you walk an even number of steps before going to bed, then this ceases to model weather and becomes a quirk of the game system. Playable models therefore interestingly connect cultural knowledge with players' mental models of game systems.

Playable models, like other inferences made by proceduralist readings, imply certain constraints on the system and its instantial assets; these can be satisfied by different compositions of operational logics (different along all three of the axes of structural syntheses, communication channel sharing, and operational integrations). The remainder of this chapter explores some examples of playable models to illustrate key concepts.

6.1.1 Space

Operational logics can be applied to represent space in a variety of ways. Text adventure games use *linking logics* connecting textual passages which can be read as describing

distinct rooms; a player's sense of place is that they are in a world of discrete spaces connected by discrete links, and that other objects likewise reside in this world. Often these worlds have a second deployment of linking logics in modeling *containment:* that just as a person or monster *is-in-a* room, a bag *is-in-a* person and a rock *is-in-a* bag. Space in these games is a hierarchy of containment and movement between spaces is essentially being taken out of one receptacle and put into another, or moved along tubes which connect ports of one receptacle to those of another. These games go to great lengths to make clear to the player where the exits to the room are, what the items and creatures in the room are, and so on. A player knows from common sense that they cannot, for example, put a bag into itself or *go up stairs* in a room that never mentions a staircase. They also know that, typically, stuff which is not in the room they are in cannot interact with them or be interacted with. These games connect schematic understandings of space to what the simulation supports, and when they are shocking or surprising it is often by violating or complicating their model of space (e.g., by making their simulated space obviously non-Euclidean or by mixing up spatial navigation and some other linked structure).

Early computer role-playing games like *Rogue* are, in many respects, similar: the world is made of floors or zones which are linked by special entrance and exit tiles like staircases. Creatures and players can carry items which themselves might contain other items, but these two types of space are roughly consistent. The key difference from textual adventure games comes from the inclusion of *collision logics:* within a single zone a creature's movement is not constrained by a linking structure but by the presence or absence of *walls* that block their path. Likewise, any creature which the player can see might, for instance, launch a projectile which gradually moves in a straight line through the simulated space. This type of space has

no physics and remains discrete, but does have an explicit local geography (in addition to the global topology of linking-logic spaces). Players know that touching something can cause an interaction and that solid things prevent movement.

Arcade games work with yet another model of space that deprioritizes linking logics but brings physics logics to bear. I will discuss how they model physics in the next section, but even their model of space is quite distinct from the previous examples. In arcade games (for example, *Asteroids*), space is continuous and not discrete. The concept of *narrowly avoiding* something comes into play, which is mostly absent from coarsely discrete models of space, and players' commonsense knowledge of physical space can be applied. Some kind of automatic or continuous-feeling motion seems important to give a model of space which goes beyond topology and geography and into the virtual sensation described by Swink [247]. One can imagine a game with discrete (i.e., non-physical) stepped movement at the pixel level, and compare this to a game with 16x16 square tile-based movement; in the move-by-pixels game, the fact that parts of objects can collide with parts of other objects (which is not true of the 16x16 game) induces substantially different relations between objects and between the player and objects, complicating the model of space. In the version with turn-by-turn pixel-by-pixel movement there is a sensation of *carefully fitting into* instead of the feeling of narrowly avoiding a collision that comes from physics. This sense of carefully aligning does not trigger proprioception but does seem to implicate spatial relations more strongly than a coarser discretization does (and also suggests objects with irregular shapes as an authorial affordance).

6.1.2 Physics

Recall that a physics logic governs the smooth movement of objects in simulated continuous space. Can a game that does not use physics logics have a playable model of physics? It is useful to separate concepts like physical systems and phenomena (the domain of playable models) from concepts like the integration of systems of differential equations (the domain of operational logics). Consider the PuzzleScript game *Midas,* which models gravity: blocks touched by the avatar turn to gold and break apart, and players can form plans to make staircases from these falling blocks to overcome obstacles (figure 6.1). This is a discrete-time game where blocks make discontinuous and instantaneous jumps downwards. Still, a player can apply commonsense knowledge of gravity and develop an intuition for breaking and stacking, so a simple playable model of physics is present. Physics (especially fluid dynamics) is also modeled in games like *Dwarf Fortress,* and a player can both learn from these games and apply their existing knowledge of fluids to them. The lack of continuous movement means that physics logics are, however, absent; physics are implemented with a spatial pattern matching logic!

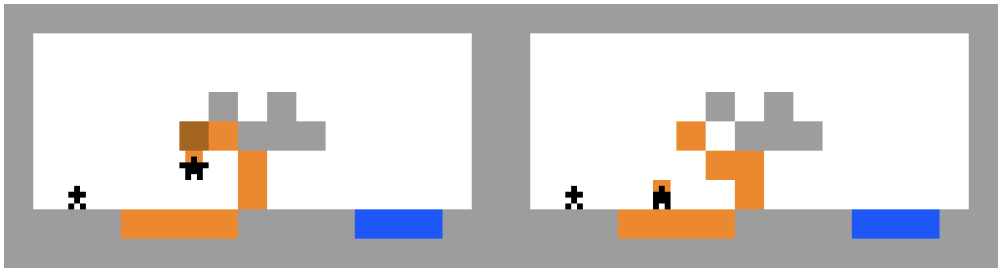

Figure 6.1: *Midas,* a game that models gravity without physics logics.

In my advancement I argued that if the player perceives physics, there is a physics

logic; the author may see different affordances and even logics than the player. Unfortunately, this argument to ignore the implementing substrate breaks down in two directions. First, I would like to be able to describe how such an author can idiomatically build levels with physical objects, and ignoring the substrate does not help there. Second, I would like to understand cases like the movement of a character in *Dragon Quest* or *Final Fantasy,* where continuous animation suggests physics but the rules of the simulation constrain movement to tile boundaries.

In the past I have used terms like *trivial model of space* or *trivial physics logic,* further confusing the distinction between logic and model. It is better to say, for example, that a collision logic can induce a model of space because it satisfies modeling obligations that lead a player to believe objects exist in a physical space, as opposed to a non-diegetic space like user interface elements. A physics logic induces a physics model of (e.g. Newtonian) mechanics and continuous, *physical* movement referring to players' experience with springs or rag dolls. Two logics that seem to *depend on* a shared model of space or persistent entities must be *consistent* with respect to those models (and the role of the compositional glue is to map out this consistency).

When someone implements gravity in PuzzleScript they are *modeling* physics without using a physics logic. Could one suppose the game has a physics logic and analyze the game based on that? Perhaps, but that is a different implementation with different runtime behavior. In this sense only models may be trivial, and not logics. The movement in *Dragon Quest* strongly models discrete space via its collision logic and that is how players read it, while weaker interpretations (or interpretations of weaker/simpler models) of continuous space, velocity, and so on can be gleaned from the animations of the entity-state logic. These interpretations give trivial

models which are less playable—they do not support the interactive and gradual exploration of a simulation model.

6.1.3 Cooking

What about more complex systems like combat, cooking, or politics, which may combine resource logics, spatial logics, and others? How do players read a game as representative of such a higher-order system? Certainly, cultural knowledge plays a large role, as recognized in Treanor, Schweizer, Bogost, and Mateas's proceduralist readings [259]: A set of game rules and assets describes cooking if *ingredients* in physical proximity are *prepared* (often using *tools*) and combined into a food *product.* Knowledge of common ingredients, cooking techniques, and foodstuffs can be attached to in-game phenomena such as sprites and game-mechanical reenactments of activities such as shaking a pan.

This assignment of game concepts to cultural concepts—ingredients, preparation processes, tools, products—is mandatory interpretive legwork that must always appear when the cultural frame of cooking is invoked. The concrete game concepts must meet certain requirements but are allowed to vary considerably: for example, ingredients must change form during preparation and be destroyed during combination, but ingredients and products could be resources of a resource logic or shapes in a collision logic. Their spatial collocation could mean touching in a collision logic, their presence in the same inventory bag, or the accumulation of enough resource units corresponding to each ingredient.

Meeting these requirements requires additional interpretive steps, grounded by the respective operational logics deployed to satisfy them. The combination of ingredients might

be a conversion of two units of flour for one of bread in a resource logic, or the graphical collision of a bun and a patty after which both vanish and a complete hamburger appears. If the ingredients are still present in their original form, or if no product results, cooking has not taken place—one can only read those behaviors as bugs or as the mere *theme* of cooking divorced from its essential procedures. Any claim that a game models cooking is incomplete if it fails to define the ingredients, tools, preparations, and products in terms of the game's lower level logics in a way that is consistent with some cultural knowledge of cooking. Thus, at a broader level of analysis, playable models compose structuring information with varying operational logics to support the player in incremental exploration, intention formation, and interpretation.

BurgerTime is an informative example here. As Treanor points out, it is difficult to read outside knowledge of cooking into *BurgerTime* [257], or to learn anything from play that transfers effectively to cooking. On the other hand, reading the game as a model of being a *chef* is a metaphorical jump that admits (slightly) more coherent translation between the real-world concepts of picking, seasoning, and grouping ingredients and the movement of enemies on burger-laden girders. Expert play of *BurgerTime* depends on dropping the ready-to-hand but uninformative model of *cooking* and trading it for a more subtle model of *chefery.*

6.1.4 Scheduling

If I am arguing that movement in games is generally achieved through modeling some type of space, is there a difference between sets of game mechanics and playable models? The key here is in specificity: a game mechanic is a concrete engagement of operational logics, while a model of, say, discrete space can be satisfied by several distinct combinations of mechanics.

The way Mario jumps in the game might be called Mario's *jumping mechanic,* but this in fact involves multiple mechanics (gravity, collision, powerups, and so on) working in concert; it is clearer to say that the game *models jumping* using physics, collision, and entity-state logics arranged in the particular mechanics used by *Super Mario Bros.*

Consider turn-taking mechanics; take *Chess* and *Final Fantasy Tactics* as examples. These could be seen as modeling *scheduling,* since the turn order is well-defined and the players make decisions to optimize aspects of the schedule (e.g., *zugzwang* moves in chess or manipulating turn order in *Final Fantasy Tactics*). The turn-taking in chess could potentially be realized as a game-mode logic—a state machine—but *Final Fantasy Tactics* has to engage multiple logics to discretize the give-and-take of a small-scale fantasy army skirmish. A digital version of chess might communicate whose turn it is by some UI label (e.g., "White's Turn"), and any white piece would become clickable; in *Final Fantasy Tactics,* turn status is communicated by moving the camera to bring the active unit in view, putting two icons over its head, and showing a menu of actions (see figure 6.2).

What logic provides the model of scheduling in *Final Fantasy Tactics?* It may be that an entity-state logic puts the unit into an active state, but then the menu (a game mode?) and camera movement are still unexplained; this also doesn't account for the game's delayed actions, which are scheduled in a similar way to (and in fact, on the same agenda as) units. The connection of the menu options to the behavior of the unit is also unexplained when considering only entity-state logics. Chess has the same kind of problem: game-mode logics usually communicate via different *screens* with a process that puts different logics and modes of interaction at work in each state, but this is not what happens in chess.

Figure 6.2: Displaying the current unit in *Final Fantasy Tactics.*

I addressed scheduling briefly in *Combat in Games,* considering the differences between *Chrono Trigger* and *Final Fantasy* in terms of how the resource logics of combat could be changed to incorporate "turn ordering" [192]. So, if turn ordering and scheduling are hard to account for in terms of individual logics, must there be such a thing as a scheduling logic? Because scheduling is so game- and genre-specific, it is probably unspecifiable as an operational logic; but it can be handled neatly by saying that scheduling can be modeled using various combinations of operational logics.

Scheduling may be extremely simple, in which case a single entity-state or game-mode logic suffices; it can also be more complex and involve resource and other logics. In *either case,* a key way that scheduling is made playable is by connecting the idea of the *currently scheduled entity* to a control logic; *who* may control *what, how,* and *when.* Control logics communicate via UI elements or decorations, by camera focus, by sprite animations, command

prompts, and so on. Of course, scheduling could also be communicated to the player by the movements of level elements or other entities in a game, and the player might not be subject to the scheduler at all! In such cases, the player needs a different way to interact with the schedule for the model of scheduling to be playable.

Control and resource logics working together yield a more complete explanation of the scheduler of *Final Fantasy Tactics.* Note that the activation, animations, and effects of delayed actions like spells are still unaccounted for without bringing in still other logics. Models of scheduling, compared to control logics, are much more abstract and could be realized without respect to units (e.g., to schedule upcoming events in a resource management simulation). They do not need much cultural knowledge, but having some can help players guess what will happen—for example, in *Final Fantasy Tactics,* some units have a higher *Speed* statistic and so intuitively a player can guess their turn might come up more often. Finally, it is worth noting that, like the *BurgerTime* player's pre-existing experience with cooking and burgers, a novice *Final Fantasy Tactics* player already comes equipped with a superficial model of the scheduler. Unlike *BurgerTime,* this model becomes more sophisticated but is not totally discarded as their knowledge of the game increases, until eventually it is understood at the level of resource transactions instead of (or in addition to) the passage of time. This is another argument that scheduling is a model rather than a logic: it can be deployed effectively without being fully understood.

6.1.5 Growth

Growth, or the idea that a living thing can become larger, stronger, or otherwise more mature over time, is modeled across a variety of game genres. In arcade games, something getting bigger is generally implemented in the collision logics with corresponding changes in physics to sell the illusion of size increases. Simulation games will use increasing resources and decreasing relative rates of resource transactions to model the growth of cities, with the understanding that larger systems have more inertia. The connection between role-playing games and *bildungsroman* is well-attested in game studies [120].

What all of these models have in common is that there is something which grows, that this growth changes some relations between that thing and the environment, and that the player either causes or responds to the growth. It could be the growth of resources, as in the role-playing game where a character's strength or wisdom increases through play, and this enables them to defeat stronger enemies; or metaphorical growth where the character unlocks a hidden ability during a story sequence where the character accepts some responsibility or experiences some trial; it could be the increasing hysteresis of a system like a simulated city that has slower feedback mechanisms as it increases in complexity; or it could be the physical growth of a plant or animal, increasing in simulated size.

6.1.6 Combat

In *Combat in Games,* I explored what requirements a game must fulfill to read as *having combat* [192]. Briefly, to be readable as having combat a situation must feature multiple combatants (sometimes on teams) in a shared space: the arena or the battle screen. These actors

must aim to do violence to each other and may perform nonviolent actions like movements or beneficial spells to enable future violence or to prevent, mitigate, or undo violence committed upon them or their allies.

Violence here is broadly construed and is not limited to health damage: forcibly moving an opponent, hindering its ability to act, and stopping it from preventing, mitigating, or undoing violence are all violent acts because they are all non-consensual. Some actions could have both nonviolent and violent components (e.g., a parry which protects the defender and stuns the attacker, as opposed to a dodge), and some could be contextually violent (e.g., bumper cars just move around, but some movements impinge on other actors' personal autonomy).

While not all actions performed in combat are violent, every combat actor must aim to do violence or support the violence of its teammates or else it is a noncombatant. This immediately distinguishes combat from attacking inanimate objects or the defenseless: combat requires a degree of symmetry and a mutual agreement to fight. Any model of combat must support a claim that violence is being done by all involved sides. This exchange of violent (and sometimes nonviolent) actions is realized by the real-time graphical, state, and collision logics of *Street Fighter 2* and the control, entity-state, and resource logics of *Final Fantasy.*

I can pin down the exchange of combat-related actions a little more firmly: At a given time, combatants have certain actions available and may select one to activate, possibly with parameters such as one or more targets; this action conditionally succeeds; the degree of its success is likewise conditional; and the target or targets (and possibly the initiator as well) react in response to it. Each of these is a hole into which different compositions of operational logics can be slotted, but every set of phenomena that players interpret as representing combat must

suggest linkages—however trivial—satisfying each component.

Finally, combat must come to an end, either altogether or just for certain unlucky combatants. Entity-state logics are often deployed alongside resource logics of health or points for this purpose, but occasionally a fighter can be disqualified for leaving or being forced outside of the combat space (collision logics at work). In general, violent actions will try to bring about a bad end for enemy actors and nonviolent actions will facilitate that violence, prevent that bad outcome, or produce a good outcome for allies. This accounts for battles whose aim is a rout as well as those fought for points.

To faithfully model combat in this sense—a sense which accounts for substantial variation across games without being uselessly broad—a game must provide:

- (1) a space in which (2) multiple agents (3) exchange violent (and possibly nonviolent) actions with each other;
- (4) a way to decide which actions an agent may perform, (5) whether they succeed, and (6) to what extent;
- (7) observable effects for all combat actions which are eventually visible at least to the initiator and target(s);
- and (8) circumstances under which agents enter or exit the fight and (9) for the combat itself to terminate.

Many of *these* concepts—space, agents, violence, and so on—are themselves the subject of various models which can independently be realized in a variety of ways. Players

effectively build up larger models out of smaller ones, but eventually each individual model is made by combining outside cultural knowledge with the operators and feedback mechanisms provided by operational logics.

Chapter 7

Defining Games via Operational Logics

So far, I have defined and explored operational logics, both individually and in terms of how they compose in various ways to build mechanics. I have also examined and *read* games in terms of the logics they deploy and the higher level structures implied by those readings. Now, the reader is equipped to *write* games—to specify their designs—as compositions of operational logics. To specify a game, one must specify exactly what the structural syntheses, shared communication channels, and operational integrations are; moreover, game-specific definitions like the graphs of linking logics and the high-level actions and input mappings of control logics must be provided.

In this chapter I will define three games in terms of how they engage operational logics. This completes the theoretical foundation of the second part of this dissertation: if I have made operational logics and their compositions concrete enough that I can use them to fairly completely describe games, that readies the way for modeling languages and formalisms that capture some significant aspects of a game's design. Any individual game is quite large, so

in most cases I will limit my attention to particular game modes or game systems rather than address every entity, character, level, ability of each game.

7.1 *Galaga*

Figure 7.1: The four main game modes of *Galaga:* attract-mode, stage-start, within-stage, and game-over.

Like many arcade games, Namco's *Galaga* is structured by a game mode logic that transitions between attract-mode, stage-start, within-stage, and game-over states (figure 7.1). Cycles between stage-start and within-stage occur repeatedly as players complete levels or lose lives (in the multi-player case), and the game-over state loops back to the attract mode. The game transitions from attract-mode to stage-start if the P1 or P2 start buttons are pressed while one or two credits are available respectively; credits are obtained by inserting quarters into the physical machine. The game transitions to game-over if all the players run out of their respective *ships* resource. Players start with two ships and gain new ones by achieving thresholds in points, a resource obtained by defeating enemies; players lose ships when their ship gets collided with by an enemy or missile or is captured. After a short delay, the attract-mode resumes.

The attract-mode itself is similar to the within-stage mode but non-interactive, playing

canned sequences of actions and text interstitials. It introduces no new mechanics or systems so I will leave it aside for now. During the stage-start and within-stage modes, a control logic governs the turn-taking scheme, with the physical buttons sometimes controlling player one's ship and sometimes player two's ship; the controlled agent is flipped when the current player changes, and one could consider this to be a specialized version of the game mode logic in the two-player case.

The background is a continuously scrolling starfield, but the lack of reference points and the relative stability of the player character and other agents does not suggest that the game has a camera scrolling through a large space, so there is no need to invoke a camera logic. As an arcade game, one supposes that collision logics will play a large role. In any state where the player's ship is visible, the player can move left or right; this movement is smooth, suggesting a physics logic (and indeed, the smooth movements of enemies in wide arcs confirms this). The fact that enemies arrive in groups and then join a formation can be obtained through the control and physics logics, along with per-enemy entity-state machines to transition them between following a fixed path, joining up with a formation, and swooping down towards the player (firing shots on the way). Pressing the game's Fire button spawns a missile sprite on the player's ship which proceeds upwards at a constant speed—but if there is more than one missile on the screen, the player cannot fire any more until at least one of them hits the edge of the screen or an enemy.

Certain enemy ships (from the uppermost rank of opponents) can emit a different sort of projectile: blue and white arcs which, on touching the player, *capture* their ship and carry it up to the top rank. If the player has only one ship left this ends the game, but if the player

has additional ships then defeating the enemy holding the ship will put the player in control of *two* ships, simultaneously firing missiles. Both of these are vulnerable to attacks, and if both are destroyed then play resumes as normal when the player's ship is destroyed. This process is illustrated in figure 7.2.

Figure 7.2: Ship capture in *Galaga.*

After defeating all the enemies in the stage, the player progresses to the next stage via a stage-start mode; every few stages they play a bonus *Challenging Stage* in which enemies approach in arcing waves but never attack, and defeating a complete wave gives extra bonus points. The player is evaluated at the end of the stage and graded on how many enemies they hit. Statistics are also shown to the player on game-over (figure 7.3). Levels gradually become harder and new enemy patterns are introduced over time, with shorter time intervals between attacks and other tweaks to the system, until the player inevitably loses all their ships.

To sum up, *Galaga* involves game mode, collision, physics, control, entity-state, progression, and resource logics. It is a useful example since its structure is fairly simple but in the particulars its engagements of operational logics can become quite complex. *Galaga* is an archetypal arcade game, and while some games in the genre add concepts like size-changing

Figure 7.3: *Galaga's* game over statistics.

characters, linked rooms, or per-character resources (e.g., health), its particular structural synthesis of operational logics is typical of the form.

To sketch out the compositions of logics in *Galaga,* one can examine the game's core concepts: the overall structure of lives and levels; and the player and enemy characters and their behavior. The operational integrations were mostly given above in the discussion of *Galaga's* mechanics, so here I will just address the structural syntheses and communication channel sharing. In the world of *Galaga,* the game mode logic forms an outer loop alongside the progression logic of stages, with resources governing its transitions: credits, lives, points, and the implicit resource of remaining undefeated enemies in each level. This gives an overall structure shared with many arcade games, fed by quarters and extended by player skill. The inner loop of *Galaga* ties a set of colliding bodies to corresponding physics entities and entity-state machines, assigns point values to each enemy, and connects AI behaviors to enemy agents and the cabinet's controls to player actions; the progression logic is also hooked up to the AI behaviors and physics. This synthesis is also extremely common among arcade games and action games in general. As

for communication channel sharing, some (like the positions of entities) are implicitly shared given the structural syntheses above, while others are defined on a case-by-case basis. Points, for example, are scored either by defeating enemies (in which case the new points are displayed where the enemy used to be) or by earning bonuses in the Challenging Stages (displayed in an after-stage report); the player's remaining ships are communicated using ship icons and the current level is communicated using level mark icons, and the total score is given in a textual display at the top of the screen.

7.2 *Super Mario Bros. 3*

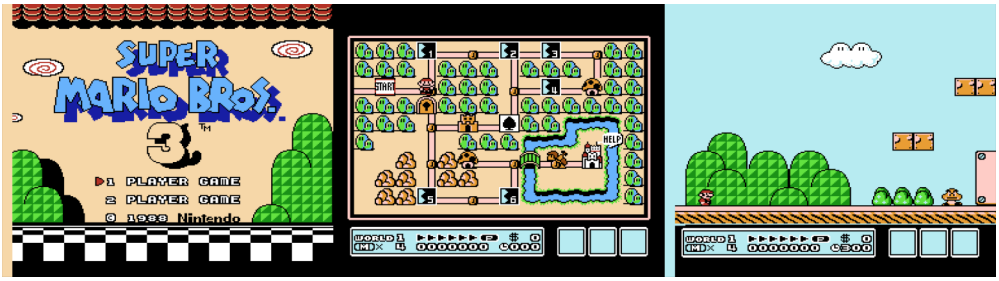

Figure 7.4: The main game modes of *Super Mario Bros. 3.*

Originally released in 1988, *Super Mario Bros. 3* (*Mario 3*) features two primary game modes (besides the attract mode, which I'll ignore in this and subsequent examples): world map and within-level modes (figure 7.4). The within-level mode is, structurally, quite similar to the original *Super Mario Bros.* It did offer some new operational integrations in terms of additional game mechanics, but the key difference is in its camera logic: *Super Mario Bros. 3* extended the camera logic of its predecessor (which only scrolled the viewport horizontally and

in only one direction) with smooth scrolling in both horizontal and vertical dimensions, even supporting simultaneous diagonal scrolling—a feat previously unachievable on the Nintendo Entertainment System, and made possible only by extra RAM embedded in the game cartridge. The main viewport of the camera logic is also smaller than earlier *Mario* games, placing a tall static menu at the bottom of the screen (this has the pleasant side-effect of reducing the number of tiles that need to be rewritten during horizontal scrolling).

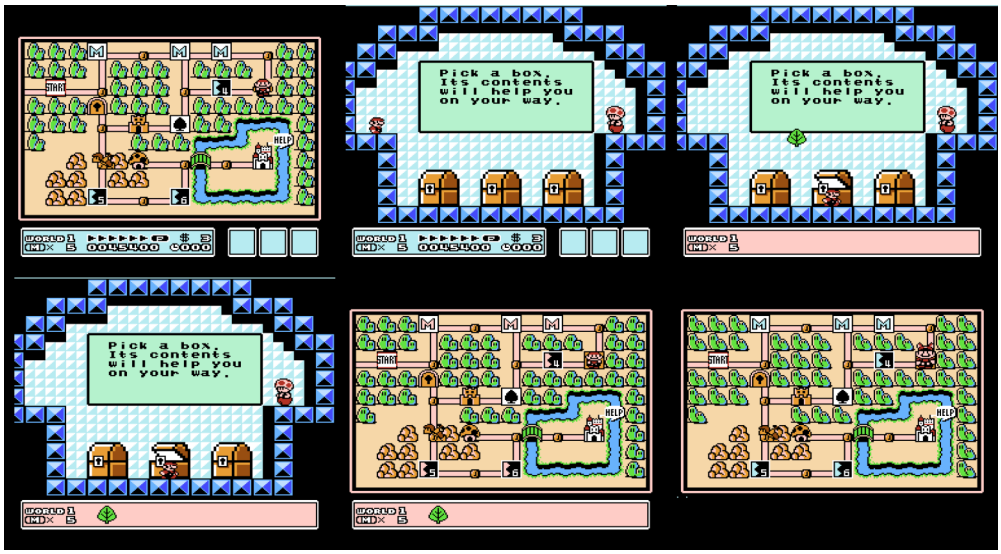

Figure 7.5: Toad House level and inventory in *Super Mario Bros. 3.*

Between levels, there are brand-new structural syntheses including an inventory system to which items can be added and used during play (figure 7.5 shows the acquisition and use of a leaf), a card-collecting and matching system (figure 7.6), and new deployments of linking logics. *Mario 3* expanded the scope of the platformer genre to encompass a link-based overworld, with links unlocked by completing stages at particular nodes; while even the first *Super Mario Bros.* used links extensively both within stages (via pipes and vines) and between stages

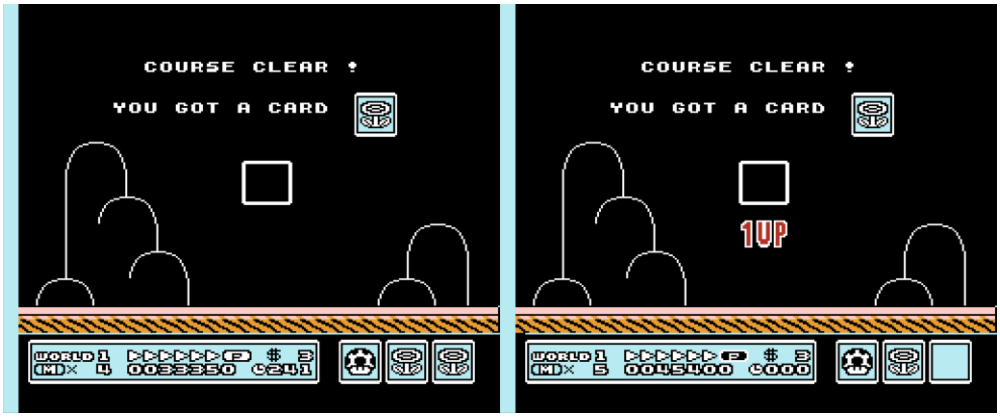

Figure 7.6: Card collection in *Super Mario Bros. 3.*

(via warp pipes), *Mario 3* gave players more control over the navigation of this linked structure. At the beginning of the game, as well as whenever the player loses a life or clears a stage, the player can select which of the currently available levels they will tackle next. The player often has a choice of which level to play, and can even skip some levels entirely; links can also pass through levels and emerge elsewhere in the world (figure 7.7).

Figure 7.7: Advanced linking structure in *Super Mario Bros. 3.*

Besides links within the current world, the player can obtain warp whistles (a reference to *The Legend of Zelda*) which bring them to a particular component of the Warp Zone depending on the world in which they used the whistle item (figure 7.8). The warp whistle is

therefore a limited-use resource which admits traversing the invisible links between each world and the corresponding part of the warp zone. There are also special *enemy levels*—for example, the Hammer Brothers which appear in the first, second, and fourth images of figure 7.7. These levels move from one spot on the linked graph to another adjacent spot whenever the player attempts a stage, and the player must clear or bypass them to make progress. Another special case is found where items like the hammer can be used to destroy obstacles and find secrets or skip difficult levels, opening up links that were previously closed. Linking logics are deployed in several ways in *Mario 3* to give the impression of a huge simulated world; players can even customize their experience to some extent by selecting which links they will follow and therefore which levels they will play.

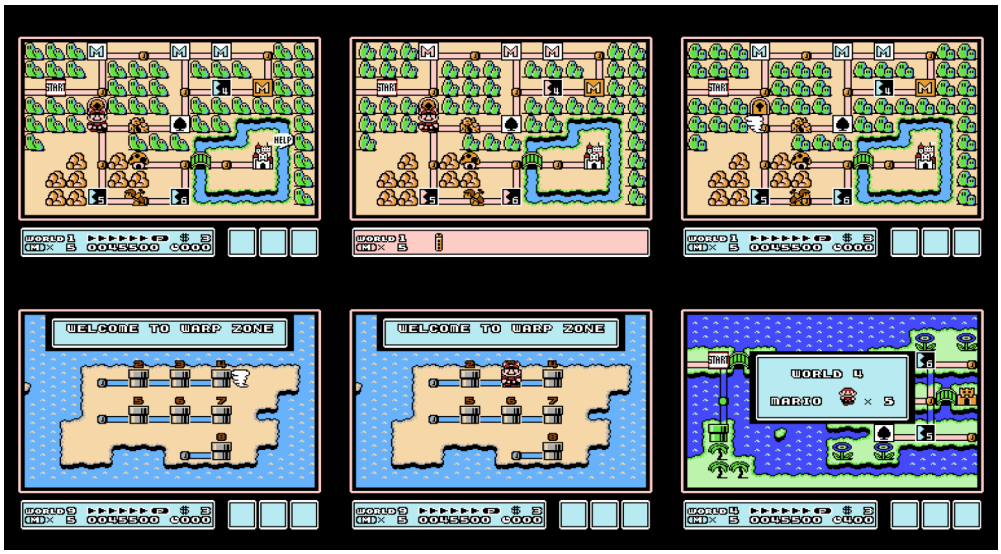

Figure 7.8: The use of the warp whistle in *Super Mario Bros. 3.*

Like other platformer games, the main play mode of *Mario 3* foregrounds control, collision, physics, camera, and entity-state logics, with some engagement of resource and linking

logics. A persistence logic maintains player state (lives, points, coins, and the inventory) across levels and handles the respawning of enemies (but not powerup-laden question mark blocks) when they are scrolled offscreen and then back onto the screen. The world map is all about linking logics, with resource logics (e.g., the inventory) used as an additional means of connecting the two game modes. Finally, occasional one-off modes like the slot-machine minigame (figure 7.9) use pattern-matching and chance logics in a limited scope.

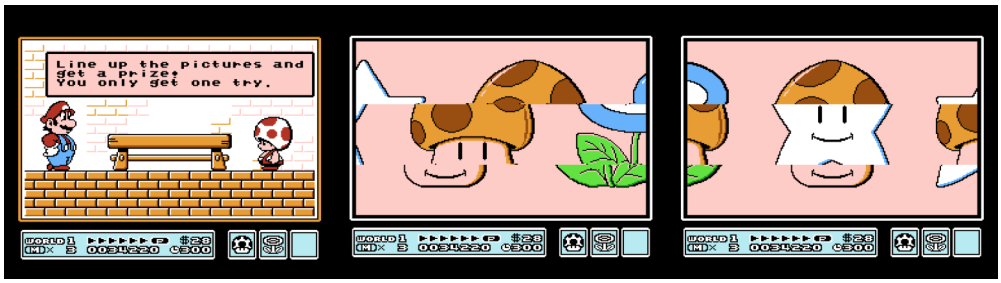

Figure 7.9: Chance house in *Super Mario Bros. 3.*

Mario 3 composes its logics in several primary ways. As in other graphical logic games, *Mario 3* synchronizes the respective entities of collision, physics, and entity-state logics. Resources like player lives are used to govern switches between game modes as is the composition of control and linking logics of the overworld map. The key structural synthesis of *Mario 3* is to connect the overworld linking logic with the individual levels used in the play mode. Communication channel sharing is largely similar to *Galaga,* with the resource logic state communicated using a dedicated screen area; one important difference is that the player can open and close the inventory menu while on the world map, which means that different aspects of the resource state are shown at different times. Information about the player's location in the larger linked space is also given in that dedicated menu area; the menu serves as an impor-

tant channel for game information that the player only needs to glance at occasionally. In fact, even aspects of the game's physics and entity-state logics are communicated using the menu. The chevrons and *P* icon fill up as Mario accelerates, and when Mario is at his top speed the *P* icon flashes; jumping at this time is extra powerful, especially when Mario is in the Racoon or Tanuki state—this tells the player when Mario will be able to fly by flapping his tail. A detailed, pseudocode-level description of *Mario 3* will be given in Chapter 8.1.

7.3 *Final Fantasy IV*

Figure 7.10: Major game modes of *Final Fantasy IV* (localized in the United States as *Final Fantasy II*).

Final Fantasy IV (*FF4*), released in 1991 and localized in the United States as *Final Fantasy II,* is a role-playing game for the *Super Nintendo Entertainment System.* In many ways it is quite typical of its genre, prominently featuring game-mode, progression, and resource logics, although its combat mode innovated on its predecessors by incorporating real-time el-

ements. Role-playing games tend to lean more heavily on distinct game modes than arcade or action games; to support their extremely long play times they need additional ways to modularize the game's content and provide variety from moment to moment. Progression mechanics and diverse (and mutually distinctive) game modes provide the leverage to achieve this design goal alongside the narrative arc of the game's instantial assets and written dialog, giving the player a sense of variety throughout dozens of hours of doing largely the same thing over and over again (figure 7.10).

FF4 features four main modes of interaction, ignoring the title screen and other meta-game modes:

- Cutscenes, during which the player has no significant input and all the game's systems are under automatic control of the progression logic;
- Navigation, where the player can move and see the world according to collision, linking, and camera logics;
- Menus, which use the directional pad and confirm/cancel buttons to move a cursor through various hierarchical sets of mainly control- and resource-logic actions;
- and Battle, where resource, control, and chance logics are mapped onto character animations.

Notably, cutscene mode is not an additional *game mode* in the sense of game mode logics, since cutscenes can incorporate any navigation or battle actions and can smoothly interrupt the middle of battles; in fact, such scripted actions in- and out-of-battle are the primary

Figure 7.11: In *FF4's* cutscenes, progression logics drive scripted control of all the game's modes.

delivery mechanism for the game's story content (figure 7.11). Cutscenes are developed as a structural synthesis of progression, control, camera, collision, and other logics which puts the latter types at the full control of pre-written *scripts.* In this sense they have a lot in common with the attract mode and arcing enemy paths of *Galaga,* with the key difference being that nearly any interaction—walking onto a specific tile, talking with a character, reducing an enemy's health past some threshold, or even falling in battle (which would normally end the game) can trigger a cutscene. The emotional impact of these special cases is strengthened since they incorporate the very operational integrations and structural syntheses with which the player is already familiar (movement, combat maneuvers and spells, and so on); if they were replaced with intertitles in the style of a silent film, it is fair to suppose that their impact would be reduced and the sense of agency (even if it is illusory) would be decreased.

Players spend most of their time with *FF4* in the game's navigation mode. As is typical for console role-playing games of the era, discrete collision logics with control logic-based directional movement on a fixed grid (in this case, 16x16 pixels) are supplemented with smooth animations between fixed positions, giving the illusion of a continuous spacetime without modeling it deeply. A camera logic has the whole screen as its viewport and scrolls to keep the

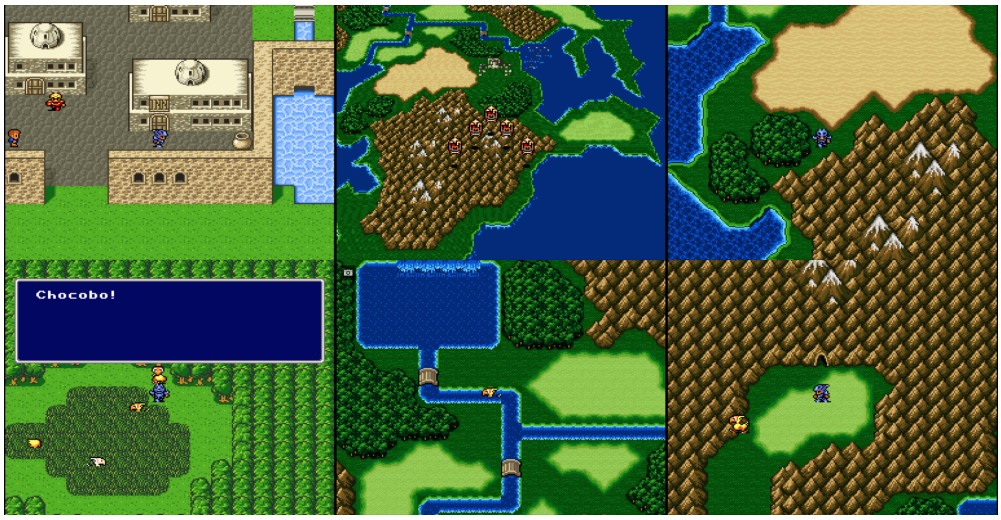

Figure 7.12: *FF4's* navigation mode features collision, camera, linking, and control logics.

player character roughly centered on the screen, and levels are padded with black space or other graphics that indicate inert and inaccessible backgrounds; the player is always seeing exactly one *room.* Certain tiles within a room are links that can be traversed to enter other rooms: doorways, teleportals, stairs, and so on. Note in the above figure that the player can obtain *vehicles* including horse-sized birds (which admit walking through shallow water) and, later, hovercrafts and airships, which give progressively greater locomotive capabilities—although vehicles must be dismounted in appropriate places before entering towns, dungeons, or other linked rooms.

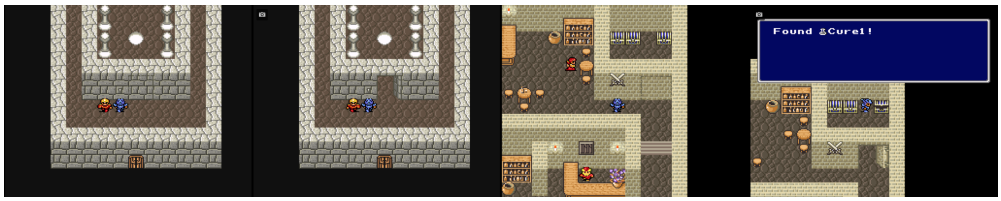

Figure 7.13: Switches in *FF4* are a special kind of interaction, like retreiving a chest or speaking with a character.

The player can also interact with other tiles and entities in the room by pressing the confirm button. This can introduce conversations with non-player characters, activate switches, or open treasure chests to retrieve an item inside (figure 7.13). In role-playing games and game engines (for example, *RPG Maker*), all such interactions are essentially of the same type: activations of progression scripts of the sort described earlier. Persistence logics give the difference in player experience between the NPC whose dialogue resets after every conversation, the switch and door which close whenever the player leaves the room, and the treasure chest which remains empty forever after being opened (or the cutscene which, once finished, never activates again).

Figure 7.14: *FF4* integrates the game's inventory with the navigation mode in special cases.

In certain special cases, the player is asked to use an item on the world map (figure 7.14). This is often employed with keys, medicines, or other plot items and communicates that the player should expect that an interaction *will be* available once they obtain some plot-related gadget. The choice of items is exactly the player's inventory, and using an item from here generally consumes it from the inventory.

The four key interconnected elements of console role-playing games are navigation, resource management, combat, and progression. In *FF4,* the main resources the player manages reside in their inventory (figure 7.15). The inventory has a limited number of slots, and each

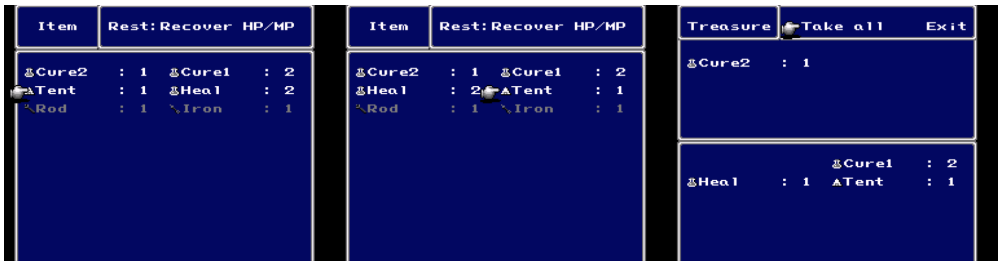

Figure 7.15: *FF4's* menus are the main site of out-of-battle resource transactions.

slot can hold one or more of some type of item up to a per-slot limit of 99. Each slot can be seen as a resource pool which is restricted to hold just one type of resource at a time, and any item obtained through shops, treasure chests, plot events, or battles is placed in an inventory slot.

Figure 7.16: Each character in *FF4* is associated with a set of equippable items.

Some items can be *equipped* by particular characters, who themselves have a set of five equipment slots (figure 7.16). Unlike the regular inventory slots, these are further restricted: each slot for each character can only hold certain sets of items, so (for example) there is no equipping a Tent as a helmet or a Carrot as a shield. Moreover, equipping certain weapons forbids the use of a shield in the other hand; these operational integrations of the resource logic

of items with that of character statistics is a key factor in players' long-term engagement with the game and a source of some strategic depth.

Figure 7.17: Some items are consumed when used in *FF4*.

Besides equipment, items are integrated with characters via *consumption* of healing or recovery items. When characters are injured in battle or use up magic points, these can be restored by certain single-use items. All of these nested menus are driven by control logics which give the menu structure, selection logics where particular items or targets are to be chosen, and resource logics which govern the preconditions and effects (to a small extent, chance logics also determine the exact effects). Apart from inventory, another key resource is *Gold Points* or *GP* which is used to purchase items and equipment in shops.

Character statistics can be checked and augmented in various ways within the game's menus, but the main mechanism for this type of change is the gain of experience points in battle (figure 7.18). In battle, character statistics and development determine the actions available via control and selection logics, and they influence these actions' effects in the resource and chance logics. Recombinatory logics drive the selection of which enemies are in the fight and what their abilities are; these can include special-cased enemy reactions like counterattacks (for example, the Needle ability which a variety of enemies use when attacked). No collision, physics, or

Figure 7.18: Combat is the primary locus of tactical decision-making in *FF4*.

linking logics are at work in this mode, and entity-state logics are present only in a very simple way (status effects like poison or incapacitation).

Figure 7.19: Experience points gained through combat in *FF4* are the main vehicle for character growth.

Each enemy is associated with a particular quantity of GP and experience points (XP), and at the end of combat these are distributed to the party. GP are given into the player's GP resource pool, while XP are split equally among those party members with non-zero HP at the

end of combat. When characters gain levels, their stats increase (the amount of increase is given by a combination of progression and chance logics); when crossing certain level thresholds they can also learn new magic spells (figure 7.19).

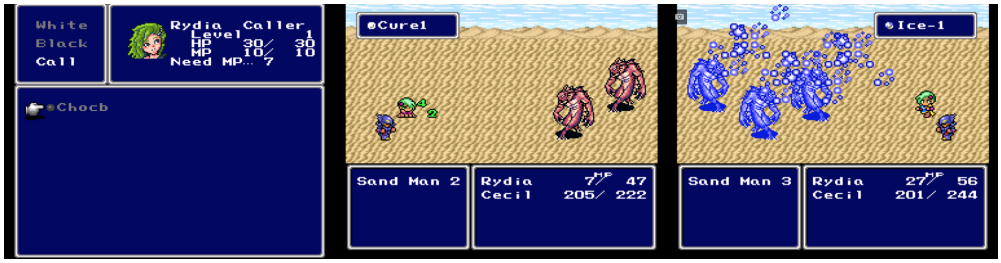

Figure 7.20: Characters in *FF4* differ mainly through their intrinsic statistics, equippable items, and learnable magic spells.

Only a few characters learn to use magic, and both obtained spells and the thresholds at which they are learned differ from character to character (the set of characters in the party changes according to the game's plot progression logic, although in updated versions of *FF4* for Game Boy Advance and other consoles a selection logic can eventually be used to pick party members in the plot's final stages). Some spells can be used in combat, others in menus only, and others in both modes. Spells can manipulate the player's position in the world or cause resource transactions, and in terms of the logics they involve are similar to items. Just as treasure chests, non-player-character interactions, and other navigation-mode operations tend to be built on the same structural syntheses and admit the same types of operational integrations, so too are item effects, spell effects, and enemy abilities in battle built of the same pieces. A key difference is that each magic spell is associated with a cost in magic points (MP), and characters must expend MP to cast spells. Another important distinction is that most spells can be targeted (via a selection logic) either at an individual enemy/character *or* an entire group of

enemies/characters (whether friends or foes).

Figure 7.21: Even without a collision logic, *FF4* has a notion of front and back ranks, as well as projectiles like arrows.

Resources used up in battle include not only characters' health and magic points, but also ammunition for weapons like bows and arrows. Arrow attacks have the property that, like magic spells, they are no weaker when used from the back rank rather than the front. Although the model of space in *FF4's* battle mode is simplistic, it does have the important feature that characters closer to the enemy group take more damage than those further back and can deal more melee damage. The party has five slots between which members can be swapped in the menus, and of which either three are front and two are back or vice-versa at the player's option. These determinations are made by the player and, while the ranks can be switched during battle, durable changes to rank composition and format are only made in the menus. It is not immediately clear which operational logics this mechanic is made of: certainly it has effects on the chance and resource logics of battle, and the choices of arrangements are made by control and selection logics, but the idea of putting characters into fixed slots with no significant spatial relationship to each other is hard to pin down. I think it is fair to characterize this as a kind of persistent selection (the player has chosen the front- and back-sets), since an interpretation of those slots as resource pools is awkward.

FF4 incorporates many operational logics and bridges between them using a combination of spatial linking, progression, persistence, resource, and game-mode logics. Each of these has its own affordances for the game's designers and carries its own significance for players. These combinations function as bridges because they are the main sites of structural synthesis across the game's diverse logics. Player characters carry a variety of resource pools and information with them across different game modes, and similar (though not identical) instantial assets tie together the character's representation in combat, navigation, and menus. During battle, the positions of characters and enemies on the screen are key sites of communication channel sharing for transient information like the amount of damage or healing done, or whether a character is currently in some special state like being poisoned; this information is also shared through the menus within and outside of combat. The association of items, statistics, and spells with individual characters is a key source of their characterization, and the idea that characters are each a locus of synthesis of diverse logics is a key design feature of role-playing games. The other key feature is the dominance of progression logics in governing the events of play both directly in cutscenes and indirectly through the available set of characters and per-character aspects of progression and resource logics; this is made possible by sophisticated persistence logics that help ensure the arrow of progress points only forwards as much as possible.

7.4 Games as Compositions of Logics

In this chapter I have detailed in three distinct cases specifically how logics are composed to produce complete games. This completes the work begun in chapter 4, and together with the exploration of *SimCity* and other games in chapter 3 confirms that operational logic theory is flexible enough to describe a range of games and concrete enough to translate into programming and modeling languages. Chapter 8 carries this through to two distinct computational models of operational logics: an operational semantics given via translation to a Python-like pseudo-code, and a logical semantics given via an Answer Set Programming encoding.

Chapter 8

Operationalizing Operational Logics

So far, I have treated operational logics mainly as a theoretical concept in the domain of game design. As it happens, this theory has a variety of coherent and complete realizations in technical and, to make an unavoidable pun, *operational* incarnations. This will yield the final theoretical pieces necessary for the second part of the dissertation. This chapter and the next (chapter 9) can be seen as the bridge between (game studies) theory and (computer science) applications. The remainder of the dissertation explores specific projects using knowledge representations gleaned from the preceding chapters and translated via these two chapters.

Every operational logic has an implementable abstract process, a communicative goal, and a strategy for presenting game state to players [166]. This chapter focuses mainly on the implementations of these processes; earlier chapters have discussed the other two elements of the definition. I explore two ways of using operational logics in computational systems: first, because operational logics must be implementable, I show how to obtain concrete implementations of specific logics; second, because the theory can be axiomatized as any other formal

theory can be, I give a characterization of families of operational logics and their compositions in answer set programming.

8.1 Implementing Game Designs

Game programs are characterized by tight loops over sets of interesting objects performing similar computations. Examples include iterating through all colliders to check overlaps or iterating through all enemies updating their AI behaviors. The code responsible for organizing each type of data and performing the computation is often called a *system:* a collision system, AI system, inventory system, dialogue system, and so on. Often, some state machine governs handoffs between different systems, and each system itself has some state to indicate what sort of processing it should perform on a given frame. This kind of structure is seen to separate concerns in a way which is relevant to game programs, and it has been reified in so-called entity-component systems under the umbrella project of data-oriented design [159].

One can imagine other approaches—an object-oriented style, perhaps, where each object performs its own collision checks or activates dialog boxes right after moving, for example. In such a framework, logics could be seen as providing pools of operations which are executed at various times. I prefer the presentation here which centralizes control, where each logic does all its respective processing all at once; it makes the roles and responsibilities of each logic clearer.

Game-making tools like Bitsy that are intentionally constrained to specific genres or types of games can yield highly user-friendly programming environments, reducing the dis-

tance between the experience of play and the experience of development. They achieve this by fixing particular structural syntheses and sets of possible operational integrations, and therefore constraining the space of authorial interventions. More general tools like GameMaker or Unity also commit to particular syntheses to varying degrees, but give an escape hatch in the form of a Turing-equivalent programming language. Unity, in fact, is currently experimenting with replacing its GameObject-and-component system with an entity-component system in the data-oriented style. The discussion in this section is therefore entity-component-inspired, and treats game programs as stateful stream processors from input sequences to output sequences.

I will proceed by outlining an implementation of *Super Mario Bros. 3* in the implementation style suggested by the theory of operational logics. This implementation will mirror the discussion in the previous chapter of the game's logics, structural syntheses, and operational integrations.

The following examples use a procedural programming language, but a functional translation can be readily obtained by treating the game system as a function from an input and a predecessor state to an output and a successor state. A *state* here is a configuration of the data that together define a situation or moment in the game: every character's position in the world, the player's progress, physical quantities, and so on. From an operational logics standpoint, each logic has some natural corresponding state and the overall composition may introduce new state to tie the logics together appropriately.

Here I consider a game state as a database of facts, or equivalently as a set of structures in memory; in a data-oriented style one might choose to associate each logic's natural types of state (the positions of colliders, velocity and acceleration of physics bodies, resource quantities

in pools, and so on) with a corresponding vector of values. In a procedural programming language, operational integrations could correspond to conditional branching and subroutine calls or to the insertion of events into a stream; structural syntheses correspond to invariants that are maintained either by updating state in multiple locations simultaneously or by determining one element of state from another as needed; and communication channel sharing corresponds to naming destinations for audiovisual output and sharing those names across multiple sites. Operational logics themselves describe the data processors, the *systems* in the entity-component jargon, which perform operations on data and schedule or otherwise determine aspects of the communication of these operations and the game state to players.

```
def update(state, input):
  controlUpdate(state, input)
  cameraUpdate(state)
  if state.gameMode == GameMode.Map:
    selectionUpdate(state)
  elif state.gameMode == GameMode.Play:
    entityStateUpdate(state)
    physicsUpdate(state)
    collisionUpdate(state)
  linkingUpdate(state)
  resourceUpdate(state)
  gameModeUpdate(state)
  persistenceUpdate(state)
```

Figure 8.1: Top-level update function in *Mario 3*.

The top-level update function of *Mario 3* might look like Listing 8.1. Input is handled by the control logic and then depending on the game mode different sets of logics are at work (in the map mode, a player selects items to use, while in the play mode control activates operators of entity-states, physics, and collision). Explicit case matching on the game mode is

not strictly necessary, since each logic itself can condition its activity on being in the right state, but it simplifies the presentation. The control logic's operations both read and modify the state, transforming button inputs into decisions about which high-level actions to activate: entering a level, activating an item from the inventory, jumping, moving right.

```
def collisionUpdate(state):
  # naive collision logic; assume every block and enemy has a body
  contacts = {[] for i in range(len(state.collision.bodies))}
  # fill contacts by iterating through each pair of bodies
  # a contact for a body is defined by a second body and a normal
  # ...
  state.collision.contacts = contacts

def physicsUpdate(state):
  for i in range(len(state.physics.bodies)):
    body = state.physics.bodies[i]
    body.velocity += body.acceleration*state.dt
    if collisionHasBlockingContact(state, body.collider,
                                   CollisionSides.LeftOrRight):
      body.velocity[0] = 0
    elif collisionHasBlockingContact(state, body.collider,
                                     CollisionSides.TopOrBottom):
      body.velocity[1] = 0
    state.collision.positions[i] += body.velocity
```

Figure 8.2: Collision and physics in *Mario 3*.

Collision, physics, and entity-states form the core of *Mario 3's* play mode, so it is natural to ask what they look like in this regime. Listing 8.2 shows a skeleton of the game's collision and physics logics, including the operational integration between physics and collision (clearing velocity values on solid collisions) and the structural synthesis that connects physics bodies and collision bodies via the spatial positions of the colliders. In this realization, I suppose that events which are conditional on collisions treat the current collision status as part

of the world state, as they might treat control inputs, and query these through functions like `collisionHasBlockingContact`; likewise, the game mode and entity states are predicates which can be queried elsewhere. One can also imagine a version where collision logics know which *questions* are going to be asked—which collisions are significant—and directly trigger effects. Later in this chapter I explore an approach to operational integrations and structural synthesis that sidesteps such questions.

```
def entityStateUpdate(state):
  state.entities.destroyed.clear()
  state.entities.created.clear()
  for goomba in state.entities.goombas:
    if goomba.state == GoombaStates.Alive:
      if collisionHasContact(state, goomba.collider,
                             EntityTypes.Mario,
                             CollisionSides.Top):
        goomba.state = GoombaStates.Dying
        goomba.timeInState = 0
      elif collisionHasBlockingContact(state, goomba.collider,
                                       CollisionSides.LeftOrRight):
        vx,vy = physicsGetVelocity(state, goomba.physicsBody)
        physicsSetVelocity(state, goomba.physicsBody, vx*-1, vy)
      else:
        goomba.timeInState += 1
    elif goomba.state == GoombaStates.Dying:
      physicsSetVelocity(state, goomba.physicsBody, 0, 0)
      if goomba.timeInState > 30:
        state.entities.destroyed.append(goomba)
      else:
        goomba.timeInState += 1
  for mario in state.entities.marios:
    # update mario similarly, considering physics,
    # collision, and control logics
    pass
  # and so on for other entities
  for (ctype, x, y) in state.entities.created:
    collisionCreateBody(state, ctype, x, y)
    physicsCreateBody(state)
    entityStateCreateMachine(state, ctype)
```

```
  for d in state.entities.destroyed:
    collisionDestroyBody(state, d.collider)
    physicsDestroyBody(state, d.physicsBody)
    entityStateDestroyMachine(state, d)
```

Figure 8.3: Entity-state logics in *Mario 3*.

In the present discussion of *Mario 3* I have used a mix of *pulling* (e.g., querying collision contacts) and *pushing* (e.g., setting a physics body's velocity) to phrase the game's operational integrations, with structural syntheses mainly defined by invariants of the state data type (e.g., that there is one collider and one physics body for each entity-state machine). Listing 8.3 illustrates how *Mario 3's* specific structural synthesis connecting physics, collision, and entity-state logics is defined, with the entity-state logic's update taking responsibility for adding new entities and their corresponding physics bodies and colliders. State transitions immediately cause effects in this regime, although they could just as well be queried in the same way as collision contacts to determine whether their effects should take place. The linking logic works in a similar way, taking ultimate responsibility for changing the game mode at the instant links are traversed.

```
def cameraUpdate(state):
  (cx,cy) = state.camera.position
  if state.gameMode == GameMode.Map:
      mapX, mapY = state.map.positions[state.map.currentLink]
      if outside((mapX,mapY), (cx,cy,screenWidth,heightWithoutMenus)):
        cameraTarget = # pick a camera target that
                       # will keep Mario onscreen
        cameraSpeed = mapScrollSpeed
  elif state.gameMode == GameMode.Play:
      marioPos = state.collision.positions[
        findFirstOfType(state.collision.types,
                        EntityTypes.Mario)
      ]
```

```
    level = state.levels[state.map.currentLink]
    if level.autoscrollPath:
      cameraTarget = nextPointOnPath(level.autoscrollPath, cx, cy)
      cameraSpeed = level.autoscrollPath.speed
    elif outside(marioPos, cameraDeadZone):
      cameraTarget = marioPos
      cameraSpeed = freeScrollSpeed
state.camera.position = smoothMoveCamera(
  (cx,cy),
  cameraTarget, cameraSpeed,
  # constrain the camera to the boundaries
  # the map or the current level
  state.camera.worldBounds
)
if state.camera.position != (cx,cy):
  state.camera.moving = true
```

Figure 8.4: Camera logics in *Mario 3*.

Finally, listing 8.4 describes the camera operations of *Mario 3* in the game's various modes. In the map mode, the camera scrolls only when Mario is about to move off the screen, while in the play mode the camera keeps Mario roughly in the center of the screen, scrolling instantaneously as necessary to keep him within a small dead zone. An exception is found in those levels which scroll on their own; the camera moves along a fixed path in these levels.

The overall pattern in the implementations above is that each logic satisfies its structural requirements by projecting relevant data out from the state—picking out its owned data and also mapping from e.g., entity-state machines to physics bodies—and then unprojecting changed data back into the state. Where logics are manipulating their own data, these changes are made directly; and where they manipulate the data of other logics, the changes work through action operators of those other logics (e.g., the Goomba's state machine forcing physics velocity changes). Passing the entire (mutable) state around to all of the logics at work is a simplifica-

tion for the present discussion; in practice one might give each logic an opportunity to define the projections and unprojections it needs to perform and the operational integrations between it and other logics, and come up with more constrained permissions for what data a logic can access. The *entity-component systems* described earlier in this chapter often use these types of mechanisms to express data dependencies [85], leading to more predictable memory access patterns and avoiding certain classes of bugs.

In programming languages it is customary to separate simulation from presentation because of the technical affordances of programming languages and platform interfaces with graphics and sound hardware. To achieve the game state presentation strategy of the operational logics used in *Mario 3,* it is straightforward to imagine corresponding `draw` routines for each logic shown above, or to use a mechanism that sends audiovisual output to an `out` parameter threaded through every function. The pseudocode assumes, for instance, that setting the camera bounds entails that only part of the currently visible content will be drawn, and indeed that the drawn content depends on the linking and game mode logics in the ways suggested by their communicative roles. In this regime, there are implicit *invariants* that connect the game state and events to its presentation, but in theory these commitments could be made explicit at the programming langauge level. The approach above should work for any game's composition of logics: decide on a data structure to encode the game state, implement each logic's process in terms of projecting-from and unprojecting-to that state (maintaining structural syntheses), define operational integrations by calling back and forth between logics, and present the game state appropriately for the logics at work.

8.2 Specifying Game Designs

The procedural and dataflow settings require firm commitments to how data are transformed and where operational integrations are located. This especially constrains structural syntheses—in the example above it was most straightforward to locate game entity creation and destruction in the code responsible for implementing the entity-state logic. Instead of this leader-follower structure, an alternative might be to broadcast to each individual logic that a new entity of the respective logic should be created, but this seems to make it harder to maintain guarantees about the connections between physics bodies, state machines, and collision volumes. Moving to a declarative, logical formulation allows for treating structural syntheses as first-class concepts, in particular for the case where a successful synthesis involves the synchronized integration of several logics under particular assumptions. This might be considered as *specifying* a game design. Recall that a complete game design includes a particular set of logics, structural syntheses, operational integrations, communication channels, and also instantial assets like game graphics and level configurations. These instantial assets are specified in terms provided by the systematic parts, in the same way that a high-level game engine gives an ontology in which specific games are described.

The Gemini project is the most recent approach to specifying a game design in logical terms [242]. Like the earlier BIPED and Ludocore systems [229], it uses Answer Set Prolog as a specification language. Also like BIPED, Gemini games can be judged against various criteria or transformed into game programs that humans can play. Gemini expands the remit of BIPED in two key ways: first, the scope of the judgement criteria is broadened beyond

specific playability queries (as outlined in chapter 5); and second, Gemini games themselves are generated as part of the answer set, rather than being given in advance as in BIPED.

The design representation used by Gemini is called Cygnus 2 [164]. As an Answer Set Prolog encoding, it is essentially a first-order logic production system with a few additional features. Each program describes a set of possible inferences over facts and rules; a *model* of a program is a unique and minimal set of inferred facts which are mutually consistent, with the curly-brace-wrapped *choice rules* acting as a key affordance for the generator. A fuller account of the use of Answer Set Prolog can be found in the preceding citations.

```
% entity types e1, e2; resources r1, r2
label(entity(e_1_XX_),dish).
label(entity(e_2_XX_),friend).
label(resource(r_1_XX_),satiation,write).
label(resource(r_2_XX_),appetite,private).
% ...
initialize(add(entity(e_1_XX_),scalar(1),location(middle,center))).
initialize(add(entity(e_2_XX_),scalar(1),location(top,left))).
initialize(add(entity(e_2_XX_),scalar(1),location(bottom,center))).
initialize(set_draggable(entity(e_1_XX_),true)).
% ...
% a game mechanic: if r2 > 1 and an e1 is touching an e2, ...
precondition(ge(resource(r_2_XX_),scalar(1)),outcome(o_4_XX_)).
precondition(overlaps(entity(e_1_XX_),entity(e_2_XX_),true),outcome(o_4_XX_)).
% decrease r2 by 10 and increase r1 by 12, then delete the involved e1
result(outcome(o_4_XX_),decrease(resource(r_2_XX_),scalar(10))).
result(outcome(o_4_XX_),increase(resource(r_1_XX_),scalar(12))).
result(outcome(o_4_XX_),delete(entity(e_1_XX_))).
% Plus proceduralist readings about this game
% ...
```

Figure 8.5: A fragment of a generated Gemini game specification.

Cygnus 2 is Game Maker-like in the sense that games are described in terms of entities which have a one-to-one correspondence with colliders, physics bodies, resource pools, and

so on; a stimulus-response style of game mechanics where conjunctions of conditions trigger set actions from a finite pool; and a few extra pieces like the ability for entities to draw and clear colored patches on the playfield (see 8.5 for a generated game and 8.6 for the constraints which led to the generation of that specification). As in Game Maker, this representation hard-codes a specific set of structural syntheses and communication channel sharing, along with some operational integrations. Game-specific differences emerge entirely in terms of the generated stimulus-response pairs and initialization conditions of the game world (including the number of distinct entity and resource types and allowed colors). A Gemini game is a high-level specification which can be put to a variety of purposes. It is an impressive project which goes further towards general game generation than previous works, although it is constrained to a particular set of graphical-logic games due to its underlying representation.

```
:- not reading(sharing,relation(entity(e(1)),entity(e(2)))).

is_consumed(0) :- precondition(overlaps(entity(e(1))),0),
         result(0,delete(entity(e(1)))),
         result(0,modify(increase,resource(r(1)))).
:- not is_consumed(_).
```

Figure 8.6: A fragment of a design intention file (partial specification) for a Gemini game.

8.2.1 Constellation

The main obstacle to generality for Gemini—and a key reason for extensive duplication in its proceduralist reading rules—is that it enforces the choice of a particular set of operational logics in a particular arrangement, but it does not treat these logics or their fundamental operations as first-class objects. Specific integrations of logics are considered as units,

which leads to awkward situations like separately defining feedback loop detection for resource growth, the amount of a particular color drawn on the playfield, and per-entity health. This leads to an explosion of base terms and generation rules *and* a lack of expressiveness, because the primitives are not appropriately orthogonal (see Listing 8.7).

```
% You can compare a resource against 0
{ precondition(compare((ge;le),RESOURCE),OUTCOME)
  : resource(RESOURCE), outcome(OUTCOME) }.
% You can compare a color against a resource
{ precondition(compare((ge;le),amount(Color),R),OUTCOME)
  : outcome(OUTCOME) }
  :- palette(Color), resource(R).
% You can compare a color against a fixed number
{ precondition(compare((ge;le),amount(Color),scalar(A)),OUTCOME)
  : outcome(OUTCOME), color_amount(A) }
  :- palette(Color).

% Resources can increase/decrease
{ result(OUTCOME,modify((increase;decrease),RESOURCE))
  : outcome(OUTCOME),resource(RESOURCE) }.
% Resources can increase/decrease by another resource's value
{ result(OUTCOME,modify((increase;decrease),RESOURCE,RESOURCE2))
  : outcome(OUTCOME),resource(RESOURCE),resource(RESOURCE2) }.
% Entity property values can increase/decrease
{ result(OUTCOME,modify((increase;decrease),property(Entity,Property)))
  : outcome(OUTCOME),entity(Entity),properties(Property) }.
```

Figure 8.7: Redundancies in some of Gemini's generation rules.

Note the repeated structure not only between different versions of resource comparison and increase/decrease, but also between the structure of the choice rules for preconditions and results. In Gemini, adding a new generable precondition requires adding a new choice rule for the `precondition` predicate, and adding a new generable result likewise requires adding new rules for `result`. The overall effect of this is to conflate the low-level grammar of genera-

ble Cygnus 2 causes and actions with Gemini-specific structures of precondition/result synthesis. This also means that the Gemini *execution* pipeline, which presently comprises a translation to JavaScript, has to implement each primitive of Cygnus 2 independently, and it is not always clear in practice which features of Cygnus 2 are accounted for within the preconditions/results framework and which features exist only as conventions or special forms in the execution framework (e.g., `initialize(set_draggable, ...)`).

One solution is to rephrase Gemini's foundations slightly; an alternative to Cygnus 2 with a cleaner decomposition of operational logics would resolve many of these problems. There are two key design commitments in Gemini: the idea that all operational integrations take the form of condition-reaction rules, and the synthesis of collision, physics, resource, and entity-state logics at the core of graphical-logic games. Listing 8.8 shows the schema for operational integrations that make up the core of Gemini, in terms of query operators and action operators from a pool of operational logics; these are at the core of *Constellation,* an OL-centric replacement for Cygnus 2. Gemini forbids generating any game rule with no precondition or with no result, and this guarantee is obtained by putting a lower bound on the choice rule. Interestingly, the encoding of Constellation in Answer Set Prolog immediately yields a machine-readable specification of the operational logic catalog from Chapter 3.

```
generated_outcome(o(M)) :-
  M = 1..N,
  generated_outcome_count(N).

1{precondition(Q, O)
  : logic(L), q_op(L, Q)} :-
  generated_outcome(O).

1{result(O, A)
```

```
  : logic(L), a_op(L, A)} :-
  generated_outcome(O).
```

Figure 8.8: Operational integrations in Gemini.

Listing 8.9 gives the structural syntheses of Gemini. This necessitates two specifications. The first is to map the domains of discourse across different logics using `entity` as a bridge: an entity corresponds to a collider, an entity-state machine, a physics body, and a health resource pool; a physics mode corresponds to an entity-state; and so on. The second goal is to maintain the invariants without which these correspondences make no sense: that entity-state and physics mode changes always come together, that a collider cannot be destroyed without also destroying the corresponding physics body and entity state machine, and so on. `birequisite` here is a convenience asserting that if either of two operations is part of a mechanic, the other must be as well. It is implemented as a constraint rule, rejecting generated games which do not uphold the birequisite.

```
collider(collision, E) :-
  entity(E).
physics_body(physics, E) :-
  entity(E).
resource_pool(resource, health(E)) :-
  entity(E).

physics_mode(physics, E, M) :-
  entity(E),
  entity_state(entity_state, E, M).
entity_state_machine(entity_state, E) :-
  entity(E).

birequisite(entity_state, set_entity_state(E, M),
            physics, set_physics_mode(E, M)) :-
  a_op(entity_state, set_entity_state(E, M)),
  a_op(physics, set_physics_mode(E, M)).
```

Figure 8.9: Gemini's structural syntheses, rephrased in terms of *Constellation.*

In this regime, an operational logic is defined as a set of facts and inference rules (see Listing 8.10), for example in a file named for that logic: `collision.lp`, `resource.lp`, and so on. The key elements include *query operators, action operators,* and some *domain predicates* describing the sorts of concepts that the logic requires. Logics may also declaratively state prerequisites and conflicts for each operator (for example, moving into a particular entity state requires that the entity is currently in a different state); it is up to the broader system to enforce these. Although the space of possible queries and actions is determined by each logic, their operational integrations are not: the stimulus-response rules of Gemini above represent just one way of performing these integrations.

```
logic(L) :- collision_logic(L).
distance(0).
loc(C) :- collider(C).
q_op(L, blocking(C1, C2, B)) :-
  collision_logic(L),
  collider(L, C1),
  collider(L, C2),
  bool(B).
a_op(L, set_blocking(C1, C2, B)) :-
  % ...
  .
q_op(L, at_loc(C, Loc, B)) :-
  % ...
  loc(L, Loc).
q_op(L, touching(C1, C2, B)) :-
  % ...
  .
% Move C to Loc
a_op(L, move_to(C, Loc)) :-
  % ...
  .
% Add/remove a collider of type C at Loc
a_op(L, add_collider(C, Loc)) :-
```

```
  % ...
  .
a_op(L, remove_collider(C, Loc)) :-
  % ...
  .
% Can't remove it if there isn't one there
prerequisite(L, remove_collider(C, Loc),
             L, at_loc(C, Loc, true)) :-
  % ...
  .
```

Figure 8.10: Collision logic as described by Constellation (abridged).

Examples of domain predicates include, for example, that a collision logic knows about such things as colliders and locations, that a physics logic knows about physics bodies and modes (sets of differential equations), that a resource logic knows about resource pools and their capacities, and so on. Structural syntheses are written in terms of these domain predicates. Gemini, in this phrasing, is just one game generator written on top of Constellation.

The operations and constraints in Constellation are abstract and qualitative as written, and do not yet go all the way to an executable game in the style of Ludocore—an interpreter or compiler has to provide implementations of the described operations. A transition semantics for Constellation is valuable future work and could support a variety of applications in game generation and specification. In fact, this could be appended to the existing operational logic theory files (`resource.lp`, `collision.lp`, etc), or it could take the place of `gemini.lp` in a hypothetical `ludocore.lp`. Supporting Ludocore-style symbolic execution of games would be helpful for supporting a broader variety of readings, unifying the capabilities of both systems, and could support code generation rather than relying on a powerful interpreter. Because the descriptions of logics are purely declarative, they can also include concepts like *feedback:*

which elements of the state and which operations are communicated to players, and along what channels; this could pave the way for even more sophisticated readings like checking whether game mechanics are effectively communicated to players, or whether two mechanics or aspects of game state are ambiguous.

Expressing low-level concrete operations like movement in continuous space can scale poorly in an Answer Set Prolog regime, especially if game behaviors must be considered over more than a few dozen time steps. While recently-added support for symbolic constraints in the Clingo answer set toolchain could be helpful, it is also natural to ask whether there are other approaches and tools that might be considered for more tightly defined problems of game design verification.

It would be ideal if a system could somehow capture both the exact semantics (and computational efficiency) of an implementation of a logic *and* the logical, declarative nature (and amenability to analysis) of the answer set encodings: specifications of game designs in terms of operational logics, but with efficient decision procedures. The second part of this dissertation achieves this goal by identifying, for each operational logic, a corresponding formal logic or mathematical formalism, and in so doing obtains this type of specification: *game design modulo theories,* by analogy to satisfiability modulo theories.

Part II

Operational Logics at Work

Chapter 9

Game Design Modulo Theories

At this point, I have developed a complete account of operational logics: their constituent definitions, a (partial) catalog of fundamental logics, and a description of the ways in which they compose, both at the level of abstract semantics and concrete implementations. Why go to the effort of constructing this conceptual apparatus? What is the benefit of an operational logics approach? In the first part of this dissertation, I have shown ways in which this refined understanding of operational logics is useful to human interpreters and designers. This portion of the dissertation focuses on the utility of this framework for computational systems. [1]

Operational logics are a useful basis for knowledge representation in and around games, mainly in the areas of game modeling and reverse-engineering. The key move in this area is to step away from considering games as bags of mechanics and towards viewing them as assemblages of abstract operations from diverse logics, with suitable formal mappings. Formal logic has clear applications to games—both the linear-logic-based Ceptre and the answer set

[1] *Portions of this chapter originally appeared in "Playspecs: Regular Expressions for Game Play Traces" [193], "Automated Game Design Learning" [196], and "Evaluating a Solver-Aided Puzzle Design Tool" [191].*

Camera	Interval logics and projective transforms
Chance	Bayesian graphical models
Entity-state	Networks of finite state machines
Collision	Qualitative/quantitative spatial constraints
Control	Action logics, event calculi
Game mode	Hierarchical finite state machines
Linking	Second-order logic over graphs, separation logic
Persistence	Temporal logic, fluent calculus
Physics	Switched systems of differential equations
Progression	Partial orders
Recombinatory	Production systems, automata
Resource	Multiset rewriting, numerical transition systems
Selection	Monadic second-order logic
Spatial Matching	Term rewriting
Temporal Matching	(Metric) temporal logic

Table 9.1: Key operational logics and formal correspondents

semantics [229, 233] have been used to great effect in design and prototyping. The remainder of this dissertation explores and expands upon a *mapping* between operational logics and well-understood formal logics (table 9.1), for potential integration in a logical framework like Satisfiability Modulo Theories [30]. This mapping aims to be as natural as possible, preserving in the modeling languages qualitative and human-relevant aspects of the communicative roles while finding concrete semantics for the abstract processes. This chapter explains the particular connections drawn in the table, while the following chapters look at compositions of these fundamental logics.

This has clear applications in game design verification: If one has a well-defined way to compose operational logics and a well-defined way to compose formal logics, one could potentially bridge these to obtain modular, compositional specification and verification of game designs. As an example of the benefits of this approach, I have made productive connections

between the theory of hybrid automata (a combination of finite state machines and switched systems of differential equations) [12] and graphical logic games [194]. This was made possible by identifying graphical logic games as a synthesis of entity-state, physics, collision, and control logics; the composition of their corresponding formal analogues is *exactly* hybrid automata. This insight led to work both in modeling languages (chapter 12) and, with collaborators, in reverse-engineering game mechanics [243] and recovering hybrid automata specifications from game characters [244] (chapter 15). Considering camera and linking logics produced work in automatically mapping game levels from observations of game play [197] (chapter 14).

There are also applications to AI general videogame playing: the GVG-AI competition specifically targets graphical logic games, and recent years' progress on general videogame playing has focused more on exploring causal and relational aspects of these games' collision logics [204]. I believe that an expanded operational logics approach could lead to even more generalized techniques, moving from the domain of specific mechanics into general combinations of abstract operations.

Operational logics are a versatile and powerful formalism addressing both perception and simulation. I hope that this work and the catalog of logics encourage their broader use both by scholars and by designers of game-making tools, modeling languages, and other computational systems. Games are more than just loose collections of homogeneous mechanics, and operational logics yield a clear alternative representation.

9.1 Describing games to computers

The question of how to describe a game to a computer—without resorting to arbitrary source code—has been of interest both to games researchers and to non-programmers who would like to make videogames of their own. There are many concrete examples of knowledge representations deployed by game engines: visual scripting languages; the broad use of navigation meshes and behavior trees in popular engines; entity-component architectures; keyframe animation; the idea of public variables as parameters; games composed of *levels* or scenes; prototype inheritance via prefabs; per-object finite state machines or state chart-like entity scripts; predicate-action triggers; linked passages; ASCII diagrams of levels; tables of game data; sets of tokens and cards on a board which are visible or controllable by certain players; and so on (as in tools like Unreal Engine, Unity, Game Maker, Twine, RPGMaker, PuzzleScript, ZZT, Zillions of Games, or Vassal). These are not just inert features of the underlying engines, but world-views that make certain classes of design central and easy to phrase and others peripheral and hard to phrase (an insight adapted from Agre [7]).

Put another way, to understand when a player's action in a Unity game might cause an object's position on the X axis to change, the reasonable hypotheses are shaped by the representational strategies Unity admits: Are any of this object's movements triggered by an animation? By physics interactions with another object? By a default character-controller? By a script on the object or yet another object? One had better hope it is not the latter—or if it is, that the script's influence on positions is contained within a standardized callback—otherwise, one is stuck analyzing arbitrary C# code, largely unconstrained and unassisted by the knowledge

representations Unity supports, with only the `transform.x` variable identifier as a guide.

Separately from commercial systems, I consider academic research projects in game specification and modeling. Of particular note is the resource exchange and feedback loop modeling tool Machinations and its more concrete incarnation Micromachinations, which can be directly incorporated into games [91, 265]. Machinations is a graphical modeling language and prototyping environment for game *economies.* In contrast to per-entity state machines, it takes a global view of the game as a network of locations through which resources flow, where flows between two locations can be conditioned on and vary with the contents of a third location; certain special node types can create or destroy resources. The Micromachinations dialect supplements the graphical language with a textual one and removes some sources of nondeterminism in the semantics. Directly modeling resource exchange mechanics supports communication and analysis of the dynamics of these systems; in effect, the knowledge representation constrains the set of mechanics specifically in order to ensure that the dynamics are comprehensible and testable.

The notion of explicit knowledge representations originated in the artificial intelligence community, and the AI and games community is another rich source of knowledge representations for games. Ensemble (née Comme il Faut) is a social simulation where autonomous agents act according to their opinions of each other and their own values and shared history [212]. Opinions can be Boolean (dating or not, embarrassed or not) or real-valued (Alice's Buddy score for Bob is 0.7); personal values mainly concern positive or negative feelings about cultural objects or concepts; and agents can refer to previous events when deciding their next action. Authors of Ensemble simulations define a *cast,* the schema of *opinions,* the set of cultural

concepts, the set of *social acts,* and *influence rules* which determine each character's inclination to perform each social act, predicated on all of the above factors. This is a particular view of social AI which is more specific than *interacting agents,* supporting both authors and readers (including automatic analyzers).

Beyond components and AI subsystems of games, there have been attempts at representing entire games in a formal modeling language. The Game Description Language (GDL) [251] targets both game-theoretic games and games-as-entertainment, considering turn-taking games of perfect information (a restriction relaxed by GDL-II, which admits incomplete information and indeterminism [250]) with potentially multiple players whose scores never decrease. Users define predicates and single-step transition relations in a variant of Datalog. The syntax and semantics of GDL make a strong suggestion that individual game rules should be relatively simple: it is awkward to define systems of rules that go through several intermediate states before settling down into a stable state (consider physics simulations with complex collision restitution) or require reasoning over hypotheticals (e.g., that a chess move is invalid if it would put the moving player's own king in danger). On the other hand, the constrained representation enables general game playing agents to directly reason about game rules in a common format without knowing the game in advance. GDL represents games as sets of transition relations which are conditioned on actions performed by players; once all players select an action, the successor state is calculated. A similar though less constrained approach is taken in Ludocore [230], which uses a more expressive underlying logic and an event calculus encoding to admit broader classes of games.

Another general language focuses on videogames; it is aptly named the VideoGame

Description Language (VGDL) [214]. Inspired by arcade games, it emphasizes physical space (smooth or discrete movement over a grid of tiles), characters occupying that space, and either real time or turn-taking movement. Also unlike GDL, game rules are more expressive in the sense that the distance between the rule definition syntax and the rule's sense (e.g., collision response, scoring, firing bullets) is decreased. On the other hand, the rule definition syntax is less expressive in absolute terms, since the syntax only permits including parametrized rules from a finite set. These rules either associate sprites with physical or control properties or select an action which should occur when two classes of sprite collide. In exchange for the loss of generality, there is a substantial gain in convenience and forward-simulation performance. Theoretically, the fact that all possible game rules share a similar information-rich structure might make it easier for general videogame playing agents to learn how to play and transfer knowledge across games, but this has not been a main focus of the general videogame playing competitions so far.

If GDL and VGDL define grammars of games, that suggests new games could be synthesized by enumerating valid strings from those languages. In a sense this is what game generators do, although for much more constrained grammars. Game generation and evaluation is a rich application area for knowledge representations of games, whether it's the succession of ontologies used in different versions of Angelina [68, 70, 71, 73] or the notion of the arcade "microgame" centered in Variations Forever [228]. Some generators focus on synthesizing mechanics and verifying that they meet certain criteria; Mechanic Miner [72], Lim's PuzzleScript rule generator [157], and Zook's work [277] are all excellent examples, and each commits to its own definition of what comprises a game mechanic. The game generator Ludi is a particu-

larly strong example of the genre, perhaps because its generative space is strongly constrained: it gradually evolves data structures encoding game rules in pursuit of interesting symmetric perfect-information board games of territory control [48]. The Ludi-generated game Yavalath is even commercially and critically successful, interesting to human players as well as to the search algorithm.

The above examples have focused on describing games to computers in an *implementable* way, mostly oriented around their execution semantics, but there is more to a game than the raw computation that it performs. Mateas emphasized the roles of *authorial* and *interpretive* affordances in expressive AI systems [165]. These give a useful lens on knowledge representations which are intended for humans. Authorial affordances, like the network threshold tests of Ensemble, message-passing between actors, or the registration of handlers for particular types of collisions give designers working within a system a language for expressing ideas. Authorial affordances induce the expressive space of a system, while their dual interpretive affordances offer readers or other interactors a way to make sense of the system's visible behavior. Interpretive affordances such as visual or textural similarities to known objects provide semantic hooks on which to hang an interpretation of a work.

But as shown in the first part of this dissertation, this zoo of game representations has the central flaw that every approach is one-off, *ad hoc,* and unique, with very little that can be transferred across systems. In the remainder of this dissertation, I show how operational logics can meaningfully contribute to this discourse just as they meaningfully contributed to unifying the vast landscape of ontological concerns in game design and analysis. This requires knowledge representations that *work* for concepts from operational logics, under the assumption

that one can weave these together to match the ways game designs weave operational logics together.

9.2 Operational Logics

The goal of this chapter is therefore to find, for each broad family of operational logics identified in chapter 3, a mathematical formalism which captures not only the abstract process of the logic but also is modeled in a way consistent with the logic's communicative goals. This can be used to build up tools like the ones in the previous section but from the foundation provided by operational logics. Since individual logics (for example, recombinatory logics) can be implemented in a broad variety of ways, it is likely that *several* formalisms could be a good fit for a particular logic. The key is to find theories which can naturally be composed together in a logical framework like constraint programming, satisfiability modulo theories, or probabilistic graphical models. I will proceed with each operational logic in alphabetical order.

9.2.1 Camera Logics

Camera logics have a relatively simple abstract process compared to the sophistication of their communicative role. The main operation of a camera logic is to map one or more regions of *world space* onto one or more regions of *screen space,* or in some cases to map from one world space region onto another (for example, for diegetic in-game cameras that display on in-world monitors).

Named regions in named spaces make up the central concept of camera logics. In 3D

worlds, a projection transform defines what is inside and outside the camera's view and maps that onto a flat rectangle; the respective spaces are easily defined by linear inequalities and the mapping is a linear transformation. In 2D worlds, transforms given by bounded intervals over X and Y coordinates suffice. In either case it is useful to name the various spaces that a region may inhabit (generally by treating each source and destination pair of *frames* respectively), and one can name the region by naming the points making up the region.

If one wants to know what objects are inside of a camera's viewport, one can check their positions with respect to the camera's visible region (a rectangular frustum or simple rectangle); if one wants to know whether two objects will visually overlap, one can apply the projection matrix to know the answer. The formalism, then, is linear algebra, and it admits the use of optimized and widely available linear programming solvers which readily integrate with other formalisms in frameworks like SMT.

9.2.2 Chance Logics

Chance logics are one mechanism for indeterminism in games (the other, AI decision-making, is outside the scope of operational logics at the time being). While games may implement randomized events in any number of *ad hoc* ways, to qualify as a chance logic the game must communicate its random choices well enough that a careful observer could determine a probabilistic model of the outcomes; this could mean showing a percentage chance-to-hit in a UI element, randomly varying the amount of damage caused by an attack within some range, or in general providing enough information through various channels that an attentive player can understand that there is some structure to the random events.

Players often misunderstand chance logics, supposing that a deterministic but ill-understood event is random, or aggressively pattern-matching a cause onto a random outcome. Compared to collision or entity-state logics, chance logics are more robust in the face of misunderstandings; still, players who comprehend the distributions at work are at an extreme advantage compared to those who do not.

Formally, a chance logic can be represented in different ways depending on the constraints put on by the modeling tools suggested by other systems. It can be abstracted away to arbitrary indeterminism in the environment for the automata-based methods, in which case the interpretation is that the best or worst possible things may happen; or incorporated into formalisms like probabilistic logics and automata that augment existing frameworks with estimated probabilities. If modeled directly, chance can be handled by analytic or by sampling approaches. A game (or part of a game) can be considered as a Markov process and treated with tools suitable for those systems, or one can model individual events using Bayesian graphical models.

In general, it is interesting to ask both "what is the best/worst that can happen?" and "what should I expect to happen?" One can combine formalisms like model checking which assume arbitrarily good or bad luck with closed-form expected value calculations or Monte Carlo simulations to come up with answers to both sorts of question.

9.2.3 Collision Logics

Collision logics also have a simple communicative goal: that things can happen when two objects overlap. Their abstract process is concerned mainly with the categorization of ob-

jects into sets which can or cannot block each other's movement, and the determination of when these contacts or other overlaps occur. The question of spatial constraints has received a lot of attention in both logical and algebraic approaches, with both qualitative and quantitative approaches in the literature. Qualitative spatial constraints can be as vague as relations describing whether objects are touching, or they can frame more precise predicates: object A is-to-the-right-of object B, object A abuts-from-above object B, and so on.

Formulae of these spatial constraint languages can be made quantitative by framing them as the conjunction of terms from interval logics in the X and Y dimensions if objects are rectangular. If objects are circular or spherical, difference logics (measuring the distance between objects) suffice to capture a lot of the important aspects of collision checking. These quantitative and qualitative approaches fit well into SMT solvers and other regimes, and the only difficulty comes in how changes to position variables are modeled. For qualitative constraints, the movement of objects can be represented as finding a path between mutually single-step-reachable combinations of spatial constraints (for example, a character can't go from left-of to right-of without passing through neither-left-nor-right-of). This can scale poorly with the number of colliders under consideration, and has something in common with the verification of concurrent processes; but it can be done in a purely propositional regime which can be computationally efficient and robust to small changes. The quantitative case might yield either linear or, unfortunately, nonlinear constraints depending on whether the position variables change with fixed or variable velocity and acceleration (see section 9.2.9).

9.2.4 Control Logics

Control logics have two main roles: determining which meta-game agent (players, AI, etc) is controlling which in-game agent, and the mapping of control primitives (button presses, activation signals, etc) onto high-level in-game actions. This can be modeled at varying levels of resolution. At the highest level, modeling tasks can ignore the control logic entirely and just pick high-level in-game actions arbitrarily, perhaps also capturing which in-game agents may act under which circumstances. At a somewhat lower level, the meta-game agents can be modeled as collaborative or adversarial agents in a framework of multiple or alternating quantifiers—"is there a policy under which each of these agents, acting rationally, will achieve outcome A or B?"

Apart from quantified boolean formulae, tools from the game theory community can be brought to bear on problems of control. Analytic methods are somewhat limited in scale but numeric approaches based on Monte Carlo techniques are promising, provided that the rest of the game can be adequately abstracted. In some cases, phrasing the question of control—which meta-game agent will perform which action and when—can be handled in an SMT framework as an optimization problem (maximize one agent's score while minimizing another's), but this also puts constraints on what theories are used to model the rest of the game. Framing it as a question of satisfiability or model checking instead provides a richer palette of theorems at the cost of making it harder to ask questions about optimal play or even reasonable competitive play. Control problems can also be modeled as coordination problems between concurrent processes (finding an input sequence that achieves or prevents some outcome), and approaches based on

action logics or event calculi fall into this group as well. Generally, control has to be modeled in different ways depending on the goal of the analysis at hand, because control is implicit in the modeling formalisms that form the foundation of the modeling task.

9.2.5 Entity-state Logics

Entity-state logics suggest to players that individual in-game entities transition between distinct discrete states with different behavior in each state. Examples include game characters with complex locomotion like Mario and also game objects like question-mark blocks, moving platforms, and *Metroid's* regenerating breakable blocks. There is a broad variety of finite-state transition systems in the literature, so there are several possible approaches to modeling entity-state logics formally. Concurrent, hierarchical state machines is a good default choice which also admits easy translations to other formalisms.

One key constraint imposed by the operational logic is that it ought to be convenient to model the individual entities as separate transition systems or state machines, and treat the game as a composition of these component machines; some analysis tools and modeling formalisms make this easier than others. The other main constraint is to be sure that the formalism can express that the transition *guards*—the conditions under which state transitions can take place—may be terms from other operational logics and that the transitions may also have some degree of nondeterminism in them.

Networks of state machines are often used to model concurrent processes, and those tools can be used off-the-shelf for the purpose of analyzing possibly reachable combinations of game entity states (provided that the allowed transition guard formulae of those tools are

expressive enough for the game in question); symbolic finite automata yield another approach which can handle any Boolean algebra in their transition language, which should cover most operational integrations. Finite state transition systems are also modeled straightforwardly in an SMT framework as, for example, one-hot vector encodings (where the value at an index is 0 if the corresponding state is inactive and 1 if it is active) with some successor predicates.

9.2.6 Game Mode Logics

Another logic typified by finite state transition systems is the *game mode logic.* Usually there is only one game mode state machine at work (i.e., only one game mode can be active at a time), and it governs the behavior of large subsets of the game's operations. In this way it is distinct from entity-state logics which can often be analyzed in isolation from each other; a game mode logic is defined mainly by what it allows, forbids, and presents of other logics' behaviors. Storyboarding UI frameworks and menu structures in role-playing games present two common examples of game mode logics at work.

A single hierarchical state machine suffices to model the logic itself, but tools from module systems and type theory can help to ask and answer questions about what activities may or may not take place within a mode or in the transitions between modes. Game modes give a useful axis of modularity for the task of analyzing games defined in operational logic terms: a tool can ignore the attract mode while analyzing the main game mode, or it can ignore the combat mode while analyzing plot progression, or analyze just one sub-tree of the whole hierarchy. Key to this decomposition is a clean delineation of the roles, responsibilities, and *capabilities* of each mode.

9.2.7 Linking Logics

Linking logics are at work wherever players explicitly navigate or view graph structures. This structure could be the spatial links between *The Legend of Zelda* dungeons or *Zork* rooms or the conceptual links within a branching dialogue system or Twine game; it could be static or dynamic. The full graph might be visible or invisible, but in general an analysis must model the player's awareness of the links in the graph and whether particular edges are traversable.

Graphs have become quite interesting in the domain of social network analytics, so mature tools for querying graph data structures are well-known (e.g., GraphQL). In logical terms a tool could use binary edge relations to represent graphs and represent many interesting queries in first-order logic. Monadic second-order logic, where sets of nodes are objects over which algorithms can reason, allows for more expressive properties along with efficient algorithms and toolkits for answering queries (e.g., MONA); it has even been specialized for graphs in the form of *graph logic*. It is also simple to model graphs and *specific* queries in an SMT paradigm. Linking logics provide another helpful decomposition if one can analyze what might happen in each node independently, and turn the problem of navigating the whole game into one of successive navigation of smaller games.

9.2.8 Persistence Logics

Persistence logics capture and communicate how some parts of the game state reset from time to time while others do not. When you leave a room and come back, treasure chests and question mark blocks do not refill, but enemies respawn; some items may return if you

leave the dungeon, but others will be gone forever; your party's HP and MP values are reset to full after each battle. Persistence is a mechanism for *scoped resets* of game state to initial or default values, and scope entry or exit can be triggered by arbitrary in-game events.

Fluent calculi give a methodology to represent stored state in a logical formalism. The main role of a persistence logic is to define the frame axioms (what stays the same from moment to moment) of such calculi appropriately for the game in question. In this way, questions around state—when it is reset, when it will be maintained, what configurations of states are reachable—can be formulated by asking questions about when resets occur.

Like control logics, persistence logics are mainly defined in terms of their effects on other logics. The fluent calculus approach can be integrated into event calculus- or temporal logic-based encodings as well as those based on finite transition systems, where resets are a well-studied phenomenon.

9.2.9 Physics Logics

Physics logics characterize continuous motion in simulated continuous spacetime. A game may progress time discretely, but for the purposes of physics logics it is enough that the game seems to involve continuous movement. If collision logics describe *moving to a location,* physics logics describe *moving by an amount.* [2] Like collision, physics can be modeled qualitatively—in terms of fast and slow, towards and away—or quantitatively in terms of systems of differential equations.

An obvious question to ask about a game's physics logics is what parts of the game

[2] And where collision and physics concern spatial locations, linking logics can be used to describe *moving to an abstract location*—among other things.

world can be reached by a character following certain physics: what values can the x and y coordinates take, what is the maximum velocity in each dimension, and so on. As mentioned in section 9.2.3, the particular physics of a game can have significant consequences on the problem of analysis. The key issue in a constraint programming framework is whether there is a need to integrate by variable or fixed time; if a timestep is fixed then the model stays in the domain of linear numeric constraints, but scaling to longer playthroughs will require nonlinear constraints due to multiplying a delta-time variable by a velocity variable. This problem can be mitigated in some cases (for example, if velocity is constant or is one of a few constants), but it is thorny. Still, motion planning using SMT is an active research area (e.g., [225]).

One can also consider reactive approaches to answering this question if one are willing to accept suboptimal or incomplete decisions. Model-predictive control gives a way to combine low-level policies for physical movement into a high-level policy which will never fail to achieve some goal if the individual low-level policies are sound (for example, [14]).

9.2.10 Progression Logics

Progression logics capture the idea that as you play through a game, different instantial assets and configurations are selected based on progress along some measures. The prototypical example is plot progression in role-playing games, where the same non-player character might say different things depending on how much of the story the player has seen or which quests they have completed; in fact, towns or the entire world map might appear differently to players at different stages in the plot. Repeated dialogue, quest lines, or scripted in-game cutscenes can be seen as progression on a small scale (potentially with resets due to

a persistence logic). In RPG-oriented game-making tools like RPGMaker, nearly every interaction with the environment—non-player characters, treasure chests, everything—counts as a *script* whose progress is primarily monotonic. Progress can be triggered by collisions, passing resource thresholds, or other operators, and it can itself trigger changes in resource amounts, persistence resets, entity-state transitions, or other effects.

The natural mathematical structure for a progression logic is the *partial order,* a binary relation which is transitive, reflexive, and antisymmetric. In other words, if event *a* comes before event *b,* there is no way that event *b* can occur before event *a.* Drawn as a graph with a shared initial node or least element (a constraint which makes this partial order a bounded join-semilattice), it looks like a directed acyclic graph rooted at the beginning of the game. A game state from the perspective of a progression logic can be thought of as a frontier: a set of nodes of which none comes before any of the others. As the player plays and completes prerequisites, nodes on each branch of the progression structure activate in turn, fork off, and join up when all the prerequisites for a node are satisfied. Games like RPGs or idle games might have several progression structures which work in parallel (character levels, plot and sub-plot progression, quest lines, and so on), and these are composed using the comes-before relation.

The dependency graph itself can be analyzed using tools from graph theory and first-order logic; if progression edges and nodes are annotated with formulae like preconditions and invariants, this can be extended to find dependencies which are not explicit in the graph but implicit in the effects of other logics in the game. Picking any node as an anchor also supports finding abstract playthroughs leading up to that node, which a designer can inspect for reasonableness or which an automated player or model checker can use as a heuristic. Having

the graph available also allows for sorting and filtering witness playthroughs gathered from analytics, characterizing how far along in the game they are in a variety of senses, which could be useful for visualization. Augmenting the graph with node labels that admit progress if *any,* rather than *all,* of their predecessors are satisfied allows for determining whether the dependencies the designer intended to require are, in fact, required—if there might be some other way to achieve progress outside of what the designer had planned. Tools from concurrent systems analysis can also be used to measure concurrent effects and dependencies across branches of the progression structure, if the progression structure is considered as a fork-join parallel program.

9.2.11 Recombinatory Logics

Recombinatory logics are, like progression logics, fairly abstract and open-ended. The fundamental operation is to provide the player with different configurations of instantial assets under different circumstances, but there is no sense of monotonic progress or even necessarily any connection to the player's behavior in the game. They are the realm of procedural content generation in dungeon crawlers and the random choice of particular configurations of enemies in an RPG, or the insertion of item names and player character names into templated text. Like AI character behaviors, they can be backed by arbitrary processes, but unlike those behaviors they are not constrained to performing in-game actions, but merely to content selection (although this may be reactive to player behavior). The fact that these logics communicate to the player as a kind of *remix* of assets which they can individually identify is key to the distinction drawn between recombinatory logics and arbitrary AI behaviors.

The correct formalism to select when addressing recombinatory logics depends on the

nature of the recombination. The two main axes are whether the decision variables are discrete or continuous, and whether the artifacts are produced in a top-down way or if they are selected from a large generative space. Consider this informal and brief taxonomy which illustrates some of the space of possible implementations of a recombinatory logic:

- Discrete, top-down decision-making often can be modeled with production systems or automata, as in the case of Tracery [64] or Expressionist [210].
- Discrete, selection-based decision-making can often be modeled with constraint satisfaction or evolutionary algorithms.
- Continuous, selection-based recombination is essentially the use of maximum likelihood estimates in statistical models like neural networks.
- Finally, continuous, top-down recombination shares a lot of structure with Markov processes or the iterated or recurrent use of neural network predictions.

In my own work, I have found symbolic visibly pushdown automata [82] to be a very useful formalism for characterizing problems in text and dialogue generation [195]; it can formulate queries over what is generable and what is not, and the tools can do useful things like *count* how many generable artifacts there are which satisfy some predicate. Working in the regime of automata gives composability and the application of symbolic automata grants the use of arbitrary Boolean algebra in the choices of which piece of content or production rule to activate next. The whole production system can also be summarized in various ways—a tool can make guarantees about content which might be selectable and use that guarantee in place of the formal model of the whole system.

9.2.12 Resource Logics

Resource logics form a key component of many genres of games, from simulation and role-playing to sports, strategy, action-adventure, and arcade games. Resources tend to come in two flavors: commodities which can be represented as tagged integers and are easily created and destroyed, and unique items which maintain their identity across transformations like movements across resource pools. Depending on the analysis need, the line between these two types can sometimes be blurred as a useful abstraction.

If all resources are unique (or if it is useful to think of them that way), multiset rewriting is an extremely helpful formalism that integrates well into constraint satisfaction regimes. In multiset rewriting, resource transactions are modeled as operations that rewrite a bag of stuff on the left hand side into a different bag on the right hand side, with strong connections to linear logic, which is quite suitable for games focusing on the movement and transformation of named items between locations [162]. For analysis questions focused on rates of return and the flows of commodities, graph-theoretic tools from control theory like weighted graphs, Petri nets, and numeric transition systems can be useful.

Besides a game's explicit resources, an analyst can also *abstract* other logics at work in a game into abstract resources like influence, control, or advantage, or even treat time as a resource. This is a step beyond the main project of this dissertation, but it is an important step for human and machinic analysis of games: whether in calculating damage-per-second from an assumed player model and the game's rules or in determining a player's expected rate of return on an investment, abstract resources play a significant role in game design and high-level play,

as evidenced by Joris Dormans's work [91, 6].

9.2.13 Selection Logics

Selection logics govern how players understand when they are making a discrete choice among a set of possible choices, selecting one or more among many options. This is another mechanism which abstracts input actions: first from the control logic to the terms of the selection logic (*select current item, move cursor,* and so on) and from there to in-game actions (*cast the Fire spell*). They are mainly used in two ways: selecting an action from a set of actions (or, e.g., a character from a set of characters), and selecting a set of entities or items for some future use (e.g., to store as a saved group or to move from one's inventory to a shop).

Selection schemes can often be modeled as mini-games inside a larger game, and abstracted out for certain analysis tasks. If the selection itself is interesting for the purposes of analysis, a second-order logic where one can select over finite sets seems like a natural choice; another option is to use a set, array, or uninterpreted function encoding in SMT to capture the idea that some choice over options can take place. It is often useful to ask what the set of possibly-selectable choices might be under different circumstances, or to ensure that what a designer thinks are mutually exclusive selections are indeed exclusive.

9.2.14 Spatial Pattern-Matching

Spatial pattern-matching logics are distinct from collision logics in that they refer to *arrangements* of objects satisfying particular predicates, not to the overlap of objects. This is somewhat of a blurry line but it is useful to separate out the idea of matching three or more tiles

with the same color from the idea of crashing a spaceship into a black hole. Whereas unique resource logics use multiset rewrite rules to describe resource transactions, spatial pattern-matching logics can use spatial rewrite rules—rewrite rules where each term on the left hand side is associated with some distinct position, perhaps described in relative terms, and the positons of terms on the right hand side are constrained according to the corresponding positions on the rule's precondition. PuzzleScript is an excellent example of such a system [147], which has some connections to cellular automata (as in *T in Y World* [266]). Spatial rewriting is usefully constrained with respect to general term rewriting and, under some restrictions, supports computationally efficient analysis. Some of the machinery introduced for analyzing regular expressions and regular expression replacement can also be applied to this problem if generalized for the two-dimensional case.

9.2.15 Temporal Pattern Matching

Finally, temporal pattern-matching describes game systems where the player must complete certain actions according to a schedule or within some time tolerance. Combo systems in fighting games are one example, and timed action puzzles are another; but the prototypical example is the rhythm game's insistent demand of beat-matching.

There are plenty of temporal logics in the literature, from linear temporal logic (and its continuous-time cousin Metric LTL) to more qualitative approaches, with some ways of handling time tightly coupled into other formalisms like timed automata. Most of these are mutually translatable in various ways, and they can also model time as any other variable and address it using tools meant for analyzing concurrent programs (these tools appear so frequently in this

chapter's review because of their ability to express a variety of useful constraints across different data types and modes, while maintaining composability). Key analysis questions include whether a schedule is achievable given the game's other rules, how tolerances on that schedule should be adjusted for players with different skill levels and reaction times, and so on. If closed form summaries or approximations of the game's systems are available to the verification task, it is especially straightforward to phrase and answer these questions.

9.3 Next Steps

I have shown how each operational logic corresponds to useful formal logics, which is a key contribution of this dissertation. The following chapters will expand on this chapter's achievement with concrete examples of how this work has already yielded significant contributions to game design support tools, game modeling languages, and the new field of automated game design learning.

Chapter 10

Game Design Support Tools

Designing games is difficult for many of the same reasons that computer programming is difficult: game designers, like programmers, define some initial state and operations on that state given user input, and these operations can interact in unexpected ways or the initial state may be misconfigured. The role of nonfunctional requirements like enjoyment or teaching a progression of skills is much greater in games than in many conventional computer programs, but existing tools for analyzing and testing computer programs mainly stop at functional requirements (with the notable exception of performance profiling). [1]

All but the simplest games are complex emergent systems where it is hard to predict the broad outcomes of even small rules changes. There are many "filters" for this in the process of game design: The designers' intuitions are a first filter, designer playtesting a second, "fresh player" playtesting a third, and play community testing a fourth filter. Each filter tests for different things [181], but some of these cases can be handled by *playerless playtesting* [180]

[1] *Portions of this chapter originally appeared in "Playspecs: Regular Expressions for Game Play Traces" [193], "Automated Game Design Learning" [196], and "Evaluating a Solver-Aided Puzzle Design Tool" [191].*

and this approach can be generalized in principle [227].

Accordingly, several researchers have proposed tools and techniques to assist game designers in their work.

One way to organize these tools is to distinguish artifact-oriented approaches like the mixed-initiative level editors Tanagra [235] and Sentient Sketchbook [155] from dynamics-oriented tools that work on the set of possible game playthroughs, e.g., the use of survivability analysis to investigate *Flappy Bird* variants [123] or the (concrete or symbolic) exploration of solution space to ensure puzzle design goals in *Refraction* [233], *Cut the Rope* [218], or *Treefrog Treasure* [32]. This dynamics-oriented design support community is my academic home; my earliest publication at AIIDE applied computational critics to game balancing [199].

Direct search on the low level transition systems of games is in general intractable or worse. Designers and researchers therefore often analyze abstractions of games that are produced manually. For example, it is conventional to call the special case of navigating a character through a virtual space "pathfinding." Pathfinding is generally implemented by abstracting the game's movement rules onto a gridded or other schematic space, doing search on that higher level representation, and then grounding out the results into low-level action plans; sometimes these plans are not realizable, and sometimes there are legitimate paths which are not found in these approximations, but this cost is generally accepted in order to obtain real-time behavior. This encodes designer knowledge into explicit abstractions, in a sense ignoring the behavior of the actual game design as realized in the program. The automatic calculation of reachable regions of a game space from designers' knowledge of physics rules is at the core of navmesh-based approaches to pathfinding [17]. At the same time, convex decompositions often abstract

the physics rules too much, so some have additionally used under-approximations like random search to find what parts of the possibly-reachable space the game's true dynamics can actually reach [175, 262].

Many successful projects in game analysis have used these manual abstractions. *Treefrog Treasure's* design tools transitioned from directly driving the game code towards path planning via parabola-line segment intersections (in fact, this abstraction later made it back into the game code) [32]. In this case, treating the game character as a point was an over-approximation that simplified search without introducing too many false paths. Shaker *et al.'s* editor for *Cut the Rope* levels used polyhedral over-approximations to show which parts of the stage were influenced by particular puzzle elements (essentially finding closed-form solutions to aspects of the puzzle) [217].

Refraction is an educational game for teaching fractional arithmetic, so it is vital that each game level tests the player's understanding of specific concepts. If a puzzle is supposed to teach, e.g., lowest common denominators, solutions that do not involve altering denominators should not be possible. Design support tools for the educational game *Refraction* use a tight over-abstraction of the game's core rules where the order in which puzzle pieces are placed does not matter, whereas the spatial relations between those remains important; this is key to keeping the encodings of puzzles and solutions small [232]. Smith's tool evaluated (and synthesized) puzzles by enumerating (abstractions of) solution states and ensuring that all states for each level satisfied designer-provided properties [233], and Butler leveraged that tool along with a partial order over educational domain concepts to synthesize a progression of levels that taught concepts in a valid sequence [50].

Creating these abstractions is important, but it is ad hoc, game-specific, and labor-intensive. Essentially, they are redundantly describing the design in several places: once in the code, and once for each abstraction over which the tools operate. My hope is that the machinery introduced in the preceding chapters could give a principled way to transform a formal specification of a game between different abstractions which are useful for different analytical purposes; in Chapter 13, I show how to extend this project to include the automatic recovery of such specifications from observations of game behavior.

10.1 Software Verification

Taking advantage of the availability of correct specifications (either provided by analysts or inferred from observations of the program), software verification ensures that a program meets a specification or that the specification satisfies particular properties. This is a lively area in software engineering, with notable success stories in the verification of hardware drivers [26] and distributed protocols [183]. There is also a substantial investment in static analysis tools which verify pre-specified properties of arbitrary programs in a given programming language (usually C, for example via [59]). Verification tools use a variety of techniques to ensure that a program meets a specification, including search through possible program states (often abstracted into some more computable domain) or closed-form analysis of specifications in simpler models of computation. This latter category is particularly interesting—so much so that programs are often abstracted into finite automata [118, 92] or specified in terms of hybrid systems or constraint programming to take advantage of efficient solution methods for those

problems.

Two families of techniques deserve special attention for their success at scaling up verification to real-world software, as opposed to toy examples. First, automated *abstraction* of programs or specifications in the style of *CEGAR* (Counter-Example Guided Abstraction Refinement) [61] tries to prove the property of interest on an over-approximation of the original system obtained through automatic methods such as allowing state transitions which are absent in the original or relaxing the problem definition. If the abstraction does not have the property of interest, then neither does the original system. Otherwise, the proof of the property in the abstraction might apply to the original as well; if not, a more precise abstraction can be found using the proof in the less precise abstraction as a lemma (under-approximations can also be used, inverting the constraints above respectively). Separately, *compositional* verification techniques which take advantage of the modular nature of software permit the verification of independent parts of the program separately, with narrow interfaces defined as assurances that subsystems provide to each other; this can be seen as a special case of abstraction but it's helpful to consider it on its own.

In games, verification is not widely used (in fact, even automated testing is uncommon enough that conference presentations advocate for it [52, 105, 148, 207]). Where verification is used, search-based methods which perform forward simulation of the complete game rules dominate, and as far as I know only *manual* abstraction techniques such as quantization of game space or time [277], collapsing time completely [233], or reduction to constraint programming [256] have been applied. Compositional verification is likewise missing in action. Since the interactions between game rules in general interact strongly, simulation-based approaches with-

out composition or abstraction to reduce the exponential complexity are intractable for larger games. Can the same benefits of modularity that operational logics provide in specification be repurposed for verification? Can models of computation which admit efficient analysis, automated abstraction, or relaxation be identified with particular families of operational logics?

This problem also arises in the verification of software, whether written in general purpose programming languages or specialized modeling languages, and in constraint solving as well. But the more one knows about a domain of analysis, the more tools one has for dealing with scale. A brief and high level overview follows, with some pointers as to how one might apply these techniques to games.

Hoare logic is a fundamental tool applied to static (i.e., without running the code) verification of sequential programs. Its key idea is that one can work backwards through program statements from a property to be proved (the postcondition) to find a claim that must be satisfied at the beginning of the program (the precondition, optionally provided by the developer); if the inputs and initial state satisfy the precondition, then the postcondition will hold afterwards. Simple rules triggered only by syntax suffice to find the necessary precondition. One important exception applies: since loops may execute an arbitrary number of times, the verifier must provide a logical claim called a loop invariant which effectively reduces the loop to a single iteration; the body of the loop must be proved to maintain the invariant, which is all that can be guaranteed about the loop. Picking an invariant is a difficult task which I've explored in my own work as part of larger projects [98]. Bounded model checkers like CBMC [60] and LLBMC [172] avoid the need for explicit invariants by *unrolling* loops up to a bounded depth, enabling them to effectively analyze code that does not rely too heavily on deep or infinite loops; this

makes them appropriate for *parts of* games but, unfortunately, only up to a bounded number of turns.

Instead of working backwards, a verifier might instead go forwards through the program accumulating *constraints* (e.g., *the final value of x is double its initial value*), seeded by a precondition or an example set of inputs, and forking this set of constraints for each encountered conditional. If each such set entails the postcondition, then the procedure is correct. This *symbolic execution* has many similarities to logic programming or type checking and is often used for *value analysis,* determining the range of possible values a variable might take.

Apart from these approaches that build logical formulae directly from program code, many verification tools work on abstract models defined by hand or inferred from code, runtime behavior, or both. Promela [118], Alloy [124], and TLA+ [144] are good examples of such modeling languages. These finite-state models can be checked either by exhaustive search similar to the model checkers above (e.g., TLC) or by automata-theoretic means (e.g., SPIN). The latter category is of special interest: while less expressive than specification languages like TLA+, there are efficient algorithms for proving a property holds or does not hold of a finite automaton without searching over every possible state.

Instead of verifying that a particular game program or game design is correct, one might imagine approaching the problem from the opposite direction: what if the designer could *only* construct valid game designs that did not violate some constraint? This is the idea behind *proof assistants,* which are user interfaces for building and verifying sentences in particular logics (e.g., the calculus of constructions [38] or higher-order logics [185]). In a proof assistant, users perform two main types of actions: defining new terms and discharging proof obligations.

After giving a definition of, say, a function to sort a list, the user might next define a proposition which asserts that the output of that function is always a sorted permutation of its input list. This gives the proof assistant a goal which must be achieved, and it is up to the user to input directives or give high-level proof advice (e.g., to unfold a definition, to apply a transitivity property, or to request that the proof assistant perform *search* to find a sequence of operations that solves or simplifies the goal). After such a directive is given, the goal is transformed and either the proof can be concluded (*Qed.*) or another directive must be provided. While interesting in their own right as a platform for defining and proving conjectures about mathematics or other domains, most interactive theorem provers support the automatic generation of programs (in, say, Caml or Haskell) which are *guaranteed* to have the properties proved of the specification. In this way, researchers have obtained verified, correct-by-construction implementations of C compilers [132] and other significant systems. It is interesting to ask what a proof assistant for game design might look like, but it is unfortunately outside the scope of this dissertation.

Incomplete approaches to verification can also be attractive for their relative conceptual simplicity. Rather than preventing errors, runtime fault repair has seen some initial success in games [152, 151]. Property-based testing in the style of QuickCheck [58] generates input data for functions according to templates, checking the results against developer-provided rules. Complex randomly generated counter-examples are then simplified and shrunk using various heuristics. This can be seen as a kind of black-box test generation, a naive cousin of symbolic white-box approaches to test coverage like Pex [254]. Much like the distinction between plausible reasoning and formal deduction, property-based testing (and related approaches like mutation testing) give confidence that the code is more or less correct, while formal approaches

ensure path (and even state) coverage. Visualizations of program behavior and metrics, even if not automatically checked or validated against some standard, can also give insights to developers who might have certain prejudices about their code's behavior violated by summaries of the actual behavior.

A final area of interest to games is *taint analysis,* a special case of data-flow analysis which characterizes how so-called tainted data (perhaps untrusted input from a user or third party) flows through the program, and whether it is acted upon directly or indirectly without first being cleansed; or, by the same techniques, whether high-security-clearance data is leaked to low-clearance procedures. Using unsafe data to construct safe data makes the latter unsafe, explaining the metaphor of cleanliness. The applications to games which I have in mind include characterizing the role of user input in influencing the evolution of the system, blaming certain visible phenomena on first-, second-, or nth-order effects from game rules or previous frames' user input, and to gauge how tightly different operational logics are integrated in a game design.

10.2 Playspecs

I have pursued a variety of projects in game specification; my work on Playspecs offers a modular and customizable approach which supports varying levels of formalism. It has not yet been adapted to support analysis in terms of operational logics, but I think this is a natural next step; I present it here to explore part of the possibility space around game modeling and specification languages.

Informal or missing specifications require frequent playtesting and human interpreta-

tion to determine desirable or undesirable behaviors. Formal game modeling approaches like BIPED [229] automate some of this design verification, but often require a comprehensive logical model of the game design to operate. This can be challenging for game designers and programmers unfamiliar with formal logic. Even familiar tools like unit or functional tests have limited utility for checking the correctness of game programs: gameplay situations are complex to configure and correctness criteria often involve the evolution of game state over time. Looking only at the behavior of games (rather than their code), most game play trace analysis uses aggregated metrics like event counts to collapse sequences of game states into sets of numbers [93]. Some work has also been done in searching for traces which contain certain events, in some cases only when prior to other events. Trace visualization and gestalt approaches [158, 190] are more prevalent than targeted queries; this may be because writing targeted queries is difficult, often combining multiple database lookups with code in a general-purpose programming language.

Playspecs [193] is a uniform but customizable play trace specification language (tailored for efficiently searching through play traces) that can be reused across different games. This language also scales up from trace filtering through to rigorous formal verification, permitting the use of the software engineering community's verification tools.

The input to these Playspecs can be witnessed play traces gathered from telemetry (or random play, or solution search) or a logical model of a game. Playspecs can be adopted at the level of the game engine or the individual game, and with a little developer effort they can be relatively natural for designers to author directly. Formally, Playspecs are ω-regular expressions over program states (instead of characters in a string); ω-regexes are a well-understood language

for specifying the behavior of computational systems [9]. Playspecs as a whole provide an application of proven techniques from formal verification to game design; a specification tool with a scalable degree of formality; and an emphasis on the use of game- and game-engine-specific concepts and syntax when specifying a game design. It is likely that these syntaces could be driven by the operational logics-focused modeling formalism of the preceding chapter.

On the shallow end, Playspecs can be used like an enhanced version of conventional regular expressions, only on sequences of game states rather than sequences of characters. If this seems to identify useful properties, the same specifications could drive automated search with minor changes to the game program; if the developer wants to scale up to longer traces or more complex properties, they can model the game in a specification language (e.g., Promela, or the specification formalisms I propose in this work) or recover a specification automatically and perform formal verification on that (possibly including compositional and abstraction-based techniques), still using the same property language.

In puzzle games, it is important to gradually present the game's concepts to the player, building up to more complex problems. This was the challenge faced by *Refraction's* designers, as I mentioned earlier—unfortunately, applying their techniques to a new game requires working largely from scratch. How can the effort required for this kind of verification be reduced?

I took the approach of extending *PuzzleScript* with an automated solver based on heuristic search and with support for Playspecs: any *PuzzleScript* game can define Playspecs that must hold for found solutions of each level. Solutions which violate those specifications are presented to the designer. This builds confidence that levels have no shortcuts and require the player to have learned certain concepts. I extended Playspec's specification language syntax to

support a subset of *PuzzleScript*'s native syntax: a specification can demand that a 1D pattern or 2D level fragment matches the current game state or that a win-condition-like predicate holds. These forms exactly mirror the *PuzzleScript* syntax.

Some solution checks expressible in Playspecs include "the red, green, and blue switches must be flipped in that order"; "the player must remove a box which starts on a goal position, replacing it later"; and "this level requires at least 20 moves, moving two particular boxes at least once each." Because this was written for *Puzzlescript* as a system and not for a particular game, this work could transfer towards any number of *PuzzleScript* games.

Customizing Playspecs for a specific game is also straightforward. *Prom Week* is a social simulation puzzle game (driven by the AI system *Comme il Faut* [170]) populated by characters who have ever-shifting *relationships* and *attitudes* towards each other. Players change the *social state* by having characters engage in *social games* with each other, such as *Ask Out* or *Backstab.* Each level of *Prom Week* asks the player to solve different social puzzles such as wooing a date or giving a bully their just desserts within a limited number of turns. The social games available on a given turn are determined by over 5,000 *influence rules* which reason over all aspects of the social state, including relationships (friends, dating, enemies), network values (buddy, romance, cool), and other factors. This rich dynamism comes at the cost of predictability: though adding additional rules is important for making characters believable, it is difficult for the designers to anticipate how any given rule affects the rest of the system.

There are three main applications of Playspecs to *Prom Week:* finding traces with interesting or surprising behavior or strategies; verifying that level-specific goals can still be met when influence rules are changed or added; and identifying traces as *unbelievable* if characters

perform unrealistic actions like repeatedly breaking up and starting to date again on successive turns. Playspecs can reason over the same social facts as the influence rules do with a custom syntax resembling labeled edges. For example, `Doug-friends-Jordan` checks whether Doug and Jordan have the `friends` relationship, and `Doug-romance<0.3-Chloe` succeeds if the value of Doug's romance network for Chloe is less than 0.3.

All of these applications help ensure that the actual game design is in line with an author's intent, and can reveal otherwise hard to discover insights about player behavior. But Playspecs could in theory be employed to help drive game engine decisions as well. In *Prom Week,* there aren't any systems in place to recognize the *type* of story that is being told; characters determine their actions based on the social state—so characters are always behaving believably (more or less)—but there's nothing guaranteeing that the story being produced is dramatically interesting. Playspecs could be applied as a recognizer to see if the actions and states generated thus far in a play through of the game adhere to any number of story archetypes and, if so, could influence the game engine to try to make the rest of the archetype more likely to pan out.

An example of a sad *Prom Week* story, easily implementable in Playspecs, might be that two characters that start off with opposite feelings of romance towards each other (i.e., a situation of unrequited love) gradually exchange their opinions of each other (i.e., the scorned turns into the recipient of the other's affections, though they are now no longer interested). This might be captured as a Playspec as:

```
Doug-romance<0.3-Chloe & Chloe-romance>0.7-Doug,

...,
```

```
Doug-romance>0.7-Chloe & Chloe-romance<0.3-Doug
```

That is, Doug begins by not being interested in Chloe, who is infatuated with him, but by the end their feelings have reversed. If connected to the engine, when the system catches wind of Doug and Chloe having this starting relationship, it could affect the existing influence rules–or create new ones–that would encourage Doug and Chloe to take actions which would lead to them both having lukewarm feelings for each other, until eventually Doug is interested in Chloe who no longer cares about him.

In short, Playspecs could be a means of augmenting *Prom Week's* dynamic character-centric social simulation with the satisfaction that comes from well structured stories. Note that this is not currently implemented, but is an exciting potential application of Playspecs for future work.

10.3 Playspec Syntax

Now that I have sufficiently motivated Playspecs, I can describe their details in a little more depth. First, I use the term *play trace* broadly to mean a sequence of data generated by activity in a game. The simplest possible play trace might be a log of the inputs provided by the user. Richer traces carry more data: more abstract inputs and events, more elements of the game state, and so on. These traces might be physical files stored on disk or may be generated on the fly by a verification tool. In the case of *PuzzleScript,* a trace is a sequence of player movements and level states found during a search for puzzle solutions. In *Prom Week,* traces contain individual social games and their outcomes; they come from either observed game play

Playspec	ω-Regex	Explanation
`p`	`p`	The fact `p` holds in the current state
`P&Q`		`P` and `Q` both hold in current state
`P|Q`	`[pq]`	Either `P` or `Q` holds in current state
`not P`	`[^p]`	`P` does not hold in the current state
`F,G`	`FG`	Sequence `F` and then `G`
`F;G`	`F|G`	Either sequence `F` or `G` holds
`F^G`		Sequences `F` and `G` both hold
`...`	`.*`	Matches any number of states
`F ...`	`F*`	`F` matches zero or more states
`F 1...`	`F+`	`F` matches one or more states
`F M...N`	`F{M,N}`	`F` matches between `M` and `N` states
`..`	`.*?`	Reluctantly matches any number of states; same variations as `...`
`F***`	F^{ω}	`F` repeats forever

Figure 10.1: Playspec and analogous ω-regex syntax.

(gathered by telemetry) or from exhaustive enumeration of games up to a fixed number of turns. Playspecs assume that game traces are *sequences of sets of facts.*

A Playspec *matches* a trace in the same way that a regular expression matches a string. One regex might match several distinct positions in the string or use start and end anchors to only match complete strings; analogously, a Playspec may match a portion of a play trace or an entire trace. Just as a regex describes a set of possible strings (a *language*), each Playspec describes a set of possible traces.

A regex checks whether each character in the string is a member of a particular set of characters, but Playspecs support a variety of complex (often game- or game-engine-specific) queries on individual states. Boolean combinations of these game-specific *basic facts* make up the *state* language fragment of Playspecs, and the game-independent syntax for describing the evolution of states over time is the *trace* (or *regular*) language fragment (Fig. 10.1 describes both fragments in a side by side comparison of Playspec and ω-regex syntax).

There are four basic facts which are the same across all games: `true`, `false`, `start`, and `end`. The first two are self-explanatory; the third and fourth indicate the beginning and end of the play trace respectively (roughly analogous to regex `^` and `$` anchors). Note that while the propositional formulae of the state language can be negated, the temporal sequences of the trace language may not be; the lookaround extension proposed later in this chapter could be put to this purpose.

All other basic facts are game- or genre-specific: *Prom Week's* basic facts query the social state, while *Puzzlescript's* basic facts concern the current configuration of the puzzle. Some facts could be provided by general-purpose game engines; there is also a connection

to the *authorial affordances* of operational logics and domain models [166, 192], which are portable across game genres.

When considering a sequence of states in Playspecs, the fundamental requirement is a way to advance (or consume) the current state. Regular expressions implicitly advance the stream of characters: `abc` means an `a` followed by a `b` followed by a `c`. This can also be read as the regex `a` followed by the regex `bc`. This is called *concatenation.* Playspecs have more complex syntax for each individual state so they use a comma (`,`) to indicate concatenation; it could be read as *and then* or *followed by.* A simple example using *Prom Week*: the Playspec `Doug-mean-Chloe, Doug-guilty, Doug-nice-Chloe` would only match traces in which the player first had Doug do something mean to Chloe, on the next turn Doug immediately felt guilty, and then on the following turn the player had Doug do something nice to Chloe. Each of these three basic facts is checked in turn against successive positions of the trace.

A designer doesn't always know in advance how long a property should hold. For instance, one might want to find traces where two characters begin dating but eventually break up. This is analogous in regex to asking for a lowercase letter eventually followed by a number using the Kleene star (`[a-z].*[0-9]`); there can be any number of characters between the two points of interest (the letter and the number), just as any sequence of game states or player actions may transpire between the characters falling in and out of love. In Playspecs, this *repetition* might be written as `Doug-dating-Chloe 1..., not Doug-dating-Chloe`. The use of `...` is read as *dating until no longer dating* (the preceding numeral `1` requires that the characters are dating for at least one state). The ellipses describe potentially multiple states in which a property holds (`Doug-dating-Chloe` in this example), with `true` as a default. `...` is also *greedy*:

it will prefer to consume as many states as possible when matching. The `..` variants consume as few states as possible. When there are multiple matches, these operators will yield all the same matches in opposite order. Minimum and maximum bounds can also be provided (as in `1...`), and these default to 0 and infinity.

The regex notion of *alternation* can be used to discover if at least one of several Playspecs holds. A Playspec can check for either an `a` followed by a `b` or else `xyz` using the regex `ab|xyz`. To avoid ambiguity with the state language's propositional disjunction `|`, and for symmetry with the `,` of concatenation, Playspecs use `;` for alternation. Continuing with the dating example, one might want traces where Doug ends up single, i.e., he breaks up with whoever he dates or else never dates in the first place. The Playspec `not Doug-dating-Chloe ...; Doug-dating-Chloe 1..., not Doug- dating-Chloe` matches when Doug never dates Chloe or they date before eventually breaking up, but doesn't account for Doug and Chloe resuming their relationship and staying together afterwards. By applying the repetition operator to that whole specification, the specification matches only traces in which Doug never lives happily ever after: `start, (not Doug-dating-Chloe...; Doug-dating-Chloe 1..., not Doug-dating-Chloe)..., end`.

Note that the number of states consumed by the alternation depends on the branch that matched. A Playspec like `Doug-lonely, (Doug-dating-Chloe; Doug-friends-Oswald, Doug-friends-Jordan), not Doug-lonely` consumes either three or four states. Alternation can be understood as cloning the expression once for each operand (i.e., either side of the `;`), replacing the expression with each of the operands, and succeeding if any of these clones match.

As a notational convenience, I introduce `^`, read as *and* or *intersection,* which is dual

to `;`. Like `;`, it effectively clones the Playspec for each operand. Unlike `;`, all clones must match the *same* portion of the trace for the match as a whole to succeed. For simplicity I require that all operands consume the same fixed number of states or that they can be stretched via `...` to fit the same segment of trace. Formally, this is the intersection of languages: a grammar which only matches strings that appear in both languages. This syntax can be used to find play traces where multiple paths leading to a single state must contain certain events or state sequences. Suppose I want to know when Doug becomes enemies with his former friend *and* lover, regardless of whether their initial relationship was Platonic or romantic: `(..., Doug-friends-Chloe, ... ^ ..., Doug-dating-Chloe, ...), Doug-enemies-Chloe`. While regex libraries in most programming languages do not offer intersection, I include it, in part to help define universal quantification (an extension I leave for future work). Further examples of the Playspec syntax described thus far can be found in Fig. 10.2.

So far, this syntax only recognizes play traces of finite length. While not applicable for *matching* existing traces, Playspecs which specify traces of infinite length can be useful for *verifying* game designs. Checking for infinite loops can detect cases where players get stuck, which would only show up as quitting in a real trace. To forbid such infinite traces, one must be able to describe them. I therefore assume by default that Playspecs recognize finite portions of traces, but introduce syntax for recognizing an infinite *suffix* which satisfies a Playspec repeatedly. The *forever* operator (called ω in ω-regex) is written with `***`: it is a semantically and visually lifted version of `...` indicating that the Playspec it modifies repeats forever. It may only appear at the end of a Playspec, and it will not match any finite play trace. Unlike `...`, it accepts no time-bounding arguments.

There are two main decisions when integrating Playspecs with an existing game or game engine: the degree of formality and the content of traces. For *Prom Week,* I have used Playspecs to filter and match existing play traces; this requires the least effort but is also the least flexible. With *PuzzleScript,* I have generated solutions using heuristic search and then tested those solutions against Playspecs; this requires only a few modifications to the underlying game engine, but exhaustively checking solutions has a high computational cost. Formally modeling the game under consideration would require extra authoring effort but could answer targeted queries very quickly.

The first step towards integrating Playspecs is transforming the game's play traces into sequences of sets of facts. This could involve modifying how traces are recorded, converting traces in an external tool, or performing on-demand translation into this format. In *PuzzleScript* I start with a sequence of input directions which are replayed through the game engine to recover the configuration of the level at each step. I write Playspecs over these augmented traces. A game with more complex state might define each set of facts implicitly via a function that determines the truth of a proposition.

As for what comprises a trace, there are three important factors. First is the trace's level of abstraction: the game activity represented in the trace. A trace might contain frame-by-frame or turn-by-turn game states or abstract level-by-level progression. Frame-by-frame traces will be too fine-grained for most use cases besides the unit-testing application.

Second is whether the trace contains instantaneous events, durative states, or both. Using only game events will yield more compact traces, but some specifications may be harder to write; on the other hand, traces with only states may make other Playspecs awkward. Includ-

ing both (or recovering states by replaying inputs) can be a good option.

Finally, the implementer must decide on a syntax for game-specific basic facts. These often use tokens and syntactic structures outside of the existing Playspec grammar, for example *Prom Week's* relationship tests or *PuzzleScript's* 1D patterns. Parsing and checking these predicates is necessarily implementation-dependent, but a portable grammar formalism could cover most use cases. The JavaScript reference implementation instead provides a customizable parser.

10.4 Checking Playspecs

The simplest case for determining whether a Playspec holds is matching a Playspec against a specific witnessed trace. As with regular expressions, Playspecs are efficiently matched by mechanical transformation into finite automata or an equivalent representation [252, 78]. Playspecs using the `***` operator may be rejected in this application—any real play trace is necessarily finite. The main difference from regex matching algorithms is that instead of checking a stream of bytes against values or ranges of values, an implementation checks a sequence of states against logical formulae over each state's set of facts. If a specific game already supports more formal usage of Playspecs, that work can be reused to decide whether an observed trace meets a Playspec: for example, a trivial formal model could be synthesized which only advances through the states in a given trace.

The second level of formality pairs Playspecs with search. For *PuzzleScript,* I developed a heuristic search which solves levels and then ensures those solutions match given

Playspecs. Matches are only made against complete solutions, but it would be straightforward to match each candidate action incrementally to guide search towards solutions that violate the specifications. Integrating Playspecs with the *PuzzleScript* engine required minimal additions beyond the reloading and replaying necessary for the automated solver's operation; parsing and evaluating the basic facts largely reused built-in features of the engine.

Finally, I consider the verification of Playspecs given a formal model of the game design. Program verification tools commonly work by transforming both the input program (often in a special-purpose modeling language) and the negation of a logical specification into Büchi automata (the infinite analogue to finite automata), intersecting them, and finding whether there is any sequence of inputs accepted by the newly constructed automaton [76]. This deserves unpacking. *Büchi automata* are used instead of the regex-equivalent finite state automata because programs running in finite memory have finitely many states, but may loop infinitely. Then, a model checker can construct the *negation of the specification* to find whether a trace can be produced which *violates* the original specification, in other words one which satisfies its negation. *Intersecting two automata* means producing a third which only accepts those traces which both the originals would accept—those traces that the program could conceivably produce which also satisfy the negation of the specification. Finally, *detecting non-emptiness* of the language represented by the resulting automaton shows whether there exist problematic traces. This last check is computationally easy once all the other work has been done.

All this is to illustrate that Playspecs can be checked by standard algorithms and, indeed, by many standard tools, since they are readily translated to conventional ω-regular expressions and thence to Büchi automata [118, 57, 92]. The Playspec and the formal model

should be at the same level of abstraction here—for instance, if the Playspec concerns game turns rather than frames, the model should as well. In practice, formal models could certainly be authored by hand using formalisms like transition systems, event calculus, Machinations diagrams [90], or a partial order of requirements [264]. Existing non-game modeling languages like Promela [117] are also excellent candidates, and the translation from Playspecs to a format which those tools support is straightforward and could be automated. If the implementer wants to work directly against their game's program code, semi-automated tools like SPOT can transform a conventional program into an automaton with some API support [92]. There are also program verification tools which can recover models from programs without such API help at the cost of larger models, e.g., DiVinE [29].

Finally, it is worth noting that there are many extensions to both linear temporal logic (a conventional verification language which is translated into the Büchi automata described above) and regexes, ripe for inclusion into Playspecs. Some examples include Metric-LTL [139], which is appropriate for realtime systems like action games; regex lookaround, which could make phrasing certain properties much more concise; first-order quantification, which permits a high level of abstraction over traces—in short, Playspecs that don't check the state of specific characters (like Doug or Oswald) but rather the states of *roles* that could be filled in by any character ("Doug's friend", "Oswald's lover") [141]; Probabilistic-LTL which describes likely versus unlikely possibilities [24]; and regex-style capture groups to identify particular subsequences of interest within the matched segment of a play trace (the reference implementation supports this extension already).

The game design support tools described so far have mainly been based on model

checking or randomized play. One can think of such tools as automated playtesters; besides such player-centric testing, Playspecs can support the unit-testing activities of game programmers. In this sense, they could be viewed as a generalization of *Inform 7's* Skein [178]: instead of expecting a concrete output text for a given input sequence, a tester could provide inputs and require that, on replay, the resulting trace satisfies one or several Playspecs. Playspecs' ability to elide details about the sequence of game states should make writing unit tests for games easier and more modular. If the input sequence were also generated via Playspecs, property-based testing tools [19, 121] could find violations; this is an incomplete solution compared to formal verification, but might be easier to integrate with existing software.

Finally, I note that some effort has been expended on converting puzzle solutions into strings where each character encodes information about a solution step [15]. I have learned via personal communication that researchers have analyzed such strings using textual regular expressions; this is similar to Playspecs in that it uses regular expressions, but it is less general.

I hope that game developers—especially game engine developers—adopt Playspecs at least at the level of matching and selecting existing or randomly generated traces, and ideally that they offer ways to drive their game engines via Playspecs. I further hope that game researchers will consider Playspecs as their language of choice for specifying properties of interest for formal models of games. I believe that as a language for defining the evolution of game states over time, Playspecs could undergird a rigorous study of game *dynamics* in the sense used by the MDA framework [122], which are so far under-theorized compared to game mechanics and aesthetics. Playspecs could be applied to AI play by running them *backwards,* generating rather than recognizing traces (as in the use of linear temporal logic in the planning

community [25]). I also suspect that Playspecs (perhaps with fuzzy or probabilistic extensions) have applications for general game playing, particularly in opponent modeling or characterizing sets of play traces produced by random or self-play. They could also be used in generative systems to filter out generated content that violates designer-specified invariants, or to support player-adaptive games by matching against the currently-unfolding play trace (perhaps in the service of drama management).

With extensions for first-order quantification, Playspecs could describe game rules directly; the authors have done so for noughts-and-crosses and other simple games. The insight is that a game implicitly defines a set of valid input traces, and Playspecs also define sets of traces. Game rules whose effects are distributed over time or with complex conditions are especially good candidates for definition in this style.

Playspecs' utility is determined in large part by the naturalness of their syntax. Making this syntax convenient enough to define on a game-by-game basis is key to their success and represents important future work.

In the next two chapters, I show game design support tools based on operational logics. While Playspecs have not been integrated into these tools, this is a natural next step; grounding Playspecs in operational logics should give a universal game play trace description language usable not only for model checking and design support, but also for analytics and other purposes.

Puzzlescript Examples

`[ROff] & [GOff] & [BOff] 1..., [ROn], [GOff] & [BOff] 1..., [GOn], [BOff] 1..., [BOn], ...`

The red, green, and blue switches must be flipped in that order. Switch objects are named by a color (`R`/`G`/`B`) and status (`On`/`Off`) pair.

```
..., '=@', ..., '=*.=', ..., '=*.=', ..., '=P.=', ..., win & end
                =O..          =OP*          =@.*
                =..*
```

The player must move a box which starts on a goal position, replacing it after moving a second box. In these 2D patterns, `@` means both a box (`*`) and a target (`O`) are present; `=` indicates a wall and `P` the player. In this level, these patterns identify unambiguous locations.

`20...  ^ ..., 5@5 [no Box], ...  ^ ..., 7@3 [no Box], ...`

This level requires at least 20 moves, moving two particular boxes at least once each. `X@Y` forces patterns to match at a specific location.

Prom Week Examples

`Doug-romance<0.3-Chloe ...  ^ not Doug-dating-Chloe 1..., Doug-dating-Chloe`

With a romance network value of less than 0.3 the entire time, Doug must go from not dating to dating Chloe.

`Doug-dating-Chloe, ..., not Doug-dating-Chloe, ..., Doug-dating-Jordan & Doug-friends-Chloe`

Doug begins by dating Chloe, but they eventually break up. Doug begins dating Jordan and befriends his old flame.

`Doug-friends-Chloe, ..., Doug-dating-Chloe ; Doug-dating-Chloe, ..., Doug-friends-Chloe`

Doug befriends Chloe and then starts dating her, or else Doug starts dating Chloe before they become friends.

Figure 10.2: Example *PuzzleScript* and *Prom Week* Playspecs

Chapter 11

Verifying Grammar-Based Recombinatory Logics

This chapter applies the operational logics-based knowledge representations from 9 towards verification problems for games authored in Ryan's *Expressionist* formalism [210]. Expressionist extends context-free grammars with semantic tags that can be interpeted in various ways by different generators. Considered as a recombinatory logic, it seems that the verification task of determining whether an Expressionist grammar could produce a particular output (or output with particular tags) is connected to questions a designer might ask about the expressive space of any grammar-based recombinatory system. Expressionist is a good stand-in for other implementations of recombinatory logics since it has a nicely constrained yet still expressive input language and it has structural similarities to many other grammar-based systems, which are a frequent choice for problems of controlled procedural generation. The use of tags can also naturally support the integration of other logics into choices on grammar expansion, as

it does in James's other projects. [1]

Grammar-based text generation is enjoying a small renaissance. The mostly context-free formalism Tracery has spurred the creation of new communities of grammar authors [64], and Expressionist and its tagged-grammars approach is being used in a variety of videogame projects [210, 150] and in the middleware technology of Spirit AI [236]. Beyond applications of Tracery and Expressionist, others are employing grammar-based approaches in games and interactive storytelling, too [119, 168, 86, 255, 153]. As an alternative to conventional natural language generation pipelines—the dominant approach in mainstream text generation—grammars are appealing because they are easier to author (especially for writers who do not code) while still being very generative (due to an inherent combinatorial explosion) [2]. The core appeal of Expressionist in particular is that its tagging mechanism enables *targeted generation* (at runtime, content can be requested by specifying the meanings it should express) and *content understanding* (the meanings expressed by generated content are known, which means a larger system can understand and act on generated content). These features are combined in an easy-to-use authoring environment.

While combining these features with ease of authoring provides significant expressive power, an unfortunate hazard has emerged from the high generativity of the context-free underpinnings. Due to the rapid combinatorial growth of generable spaces yielded by grammar authoring, where a day of authoring can yield billions or more outputs [210], targeted generation becomes a search challenge: even when the meanings expressed by each and every generable

[1] *Portions of this chapter originally appeared as "Analyzing Expressionist Grammars by Reduction to Symbolic Visibly Pushdown Automata" [195].*

[2] Elsewhere, Ryan has more extensively situated the grammar-based approach against conventional techniques [209].

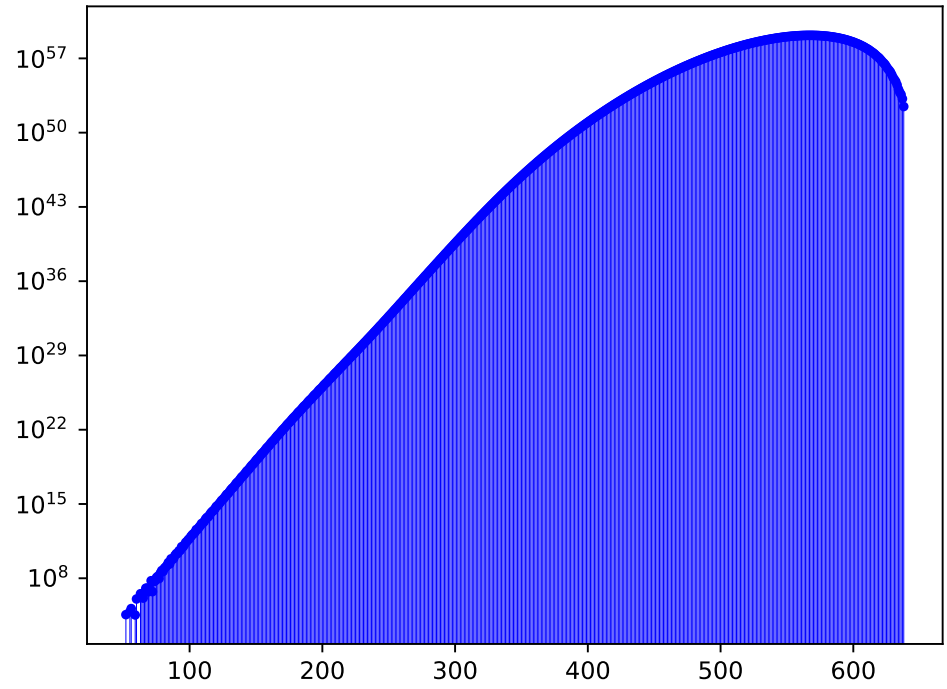

Figure 11.1: A histogram of grammar output count (log-scale) by output length. This grammar has 10^{64} possible outputs.

output are known, staggeringly large search spaces must be searched to retrieve a meaning that is requested at runtime. This problem is even worse at authoring time, when authors want to analyze an authored grammar for higher-order considerations beyond the meanings of individual outputs. This requires more complex queries over huge content spaces, which means this kind of design support is an even more significant search challenge.

Thus, the idea that a system can analyze a tagged grammar and make claims about the sorts of outputs it is likely to generate—or whether it can produce outputs that have certain meanings—is important both for expressively constrainable content generation at runtime

and to support grammar authors at design time. As grammars become more complex and as designers hope to use them to address more dynamic situations, it may not be obvious whether a grammar could produce, for example, a line of dialogue where a character says their name twice in the course of one introduction. An author may also want to know how probable a given output is in the context of the grammar's total generable space, or to visualize a histogram of the possible lengths of all outputs, as in Fig. 11.1. At runtime, such a system must furnish generated content (not necessarily strings) with the right meanings at the right time. These design questions and runtime considerations may not come up for small grammars, where exhaustive search is feasible, but (in this renaissance of grammar-based text generation) typical grammars authored for use in real expressive applications will rarely or never be small [210].

The problem, again, is a search problem that emerges from the naturally massive generable spaces yielded by the grammar-based approach to text authoring. To solve this problem, these huge spaces must be reduced to smaller ones that can more efficiently be operated over.

Fortunately, the text generation community is not alone in using formalisms like grammars to describe objects of interest (as shown in the previous chapter). The area of *software verification* uses equivalent flavors of automata to great effect in formulating and answering questions including whether a regular expression might fail to catch a dangerous class of user input. Many of their techniques and algorithms are applicable here by considering grammars as something like a program with function calls and returns, which admits directly analyzing their *structures* rather than their huge *generable spaces.*

In this chapter, I investigate and compare a family of techniques for controlling and analyzing Expressionist grammars, using actual authored examples as test cases. First, I *re-*

duce these grammars to *symbolic visibly pushdown automata* [82], and then use off-the-shelf algorithms to analyze those automata (this follows the approach of Chapter 9). I evaluate this approach on small, medium, and large grammars, with test cases of varying difficulty. While this chapter represents significant steps toward unrestricted targeted generation from tagged grammars, providing a set of explicit performance baselines, performance issues illuminate opportunities for future improvements.

11.1 Related Work

Beyond applications of Tracery and Expressionist, several recent projects in expressive text generation have used generative grammars [119, 168, 86]. By employing such grammars, these systems harness the power of *templated dialogue*, a pattern used in *Prom Week*, Versu, the LabLabLab trilogy, *Event[0],* and various works of interactive fiction [170, 95, 149, 174, 220]. One account of these approaches is that they forego the power of conventional natural language generation pipelines to emphasize authorability and expressivity above other concerns. In turn, the current project aims to further improve authorability by providing novel design support at authoring time (authors form offline queries about what meanings generable content can express) and to further improve expressivity by providing novel generation support at runtime (a system requests generated content in terms of the meanings it ought to express).

Sadly, there has not been much work on design support for procedural text authors, but Garbe *et al.'s* work on visualizing the combinatorial content space of the interactive narrative *Ice-Bound* is a touchstone in this area [102]. Emily Short has also written about approaches

to and prospects for visualizing the combinatorial text spaces yielded by generative grammars [222, 223]. Reductionist's approach, which supports queries about the array of meanings that generable content can express, can be thought of as a method for authors to investigate the *expressive range* of their text generators. The notion of expressive range is due to Smith and Whitehead [234], and recent work by Cook *et al.* has explored design support via visualizing expressive range [74].

With regard to runtime concerns for procedural text, Short and Dias have individually articulated the need for generative text that expresses salient meanings, rather than content yielded by random sampling from a generable space [221, 87]. In procedural generation more broadly, this relates to the *10,000 Bowls of Oatmeal* problem articulated by Compton [63] and expanded on by Cook [67] and Cardona-Rivera [51], in which a generative system can fall prey to producing an astronomical number of content variations that do not provide salient or meaningful differences to a human observer. The design of Expressionist inherently confronts this issue by allowing authors to efficiently index their generable spaces according to prominent authorial concerns, which means distinctions in the meaning of generable content will be reified as distinctions in the tags attached to such content.

11.2 Targeted Generation

Expressionist is a tool for text generation that associates units of generable content with their meanings, by a scheme in which tags may be attached to the nonterminal symbols in a context-free grammar [210]. When content is generated, it comes bundled with all the tags

attached to all the nonterminals that were expanded to produce it. *Meanings* is Expressionist's term to indicate any semantic, pragmatic, or other information (e.g., a character personality trait) that content may express in the application context. As mentioned earlier, this scheme critically enables two things:

- **Content understanding.** Because generated content comes bundled with its meanings as structured metadata, the larger system may understand it and execute any associated effects. For example, a generated insult might carry tags specifying that the recipient of the line should lower their affinity for the speaker by some amount.

- **Targeted generation.** A system requests content by requesting the meanings that it should express. For instance, the system might request an insult, or something more specific, as in the example given below.

Targeted generation can naturally be thought of as a search task: find in the space of generable outputs one that has the requested meanings (i.e., requested tags). Due to the combinatorial explosion of generative grammars, this search task becomes a major challenge. In response to this, previous work introduced a search method called *middle-out expansion*, where a nonterminal with a desired tag is targeted and the system traverses the grammar in both directions [209] —avoiding undesirable tags—until a completed path has been formed. This method only works for requests to target a *single* desired tag, however: unless there is an individual nonterminal with all the desired tags, targeting multiple tags requires finding a path that connects multiple nonterminals that together have all the desired tags, which could equate to search across a space of trillions or more paths. To approximate solutions to cases where

multiple tags are desired, a follow-up method, *heuristic expansion,* carries out *greedy* middle-out expansion by preferring production rules whose expansions have nonterminals with desired tags [208]. This method has its own drawbacks, though: it does not guarantee that all desired tags will be collected, and it is susceptible to local maxima.

The overarching goal of this line of work, which the above methods have not satisfied, is to support content requests that contain the following: any number of tags the content must have, any number of tags that it must *not* have, and optionally a scoring metric specifying the desirability of all other tags. Moreover, these kinds of content requests should be fulfillable at both runtime and authoring time, as a form of design support taking the form of queries about the range of content requests that *can* be satisfied at runtime. While these two kinds of support have different practical considerations (relating to the purposes they serve, what is then done with the generated content, etc.), they have the same technical requirement: a mechanism for satisfying content requests of the structure shown above. Note that *content request* (runtime) and *query* (authoring time) are essentially interchangeable terms: each is a solicitation of content satisfying a provided set of criteria.

11.2.1 Example

Here, I will illustrate this idea using actual examples from the system that generates text in *Juke Joint* [208]. In this game, procedurally generated characters have visited a small-town bar to mull over personal dilemmas, and the player is a ghost who haunts a jukebox in the bar, selecting which of its songs will play next. As the lyrics of a song emanate across the bar, each stanza elicits *thoughts* in the minds of the characters—expressed in generated

natural language—that work to gently guide their streams of consciousness toward prospective resolutions to their dilemmas. When a lyric plays, its *themes* become activated in the minds of the characters, and this may cause other concepts to also become activated. For instance, if the current stanza has the theme *commitment*, the concepts *my job* and *my family* may also become activated for a character who is committed to their job and their family. Meanwhile, the authored Expressionist grammar includes tags that correspond to all of the concepts that can become activated. After a stanza plays, all of the concepts that have become activated in a character's mind are collected to form a content request for an elicited thought to be generated for the character. In the above example, the content request would look something like this:

```
present:  'commitment'
absent:  N/A
metric:  ('my job', 3), ('my family', 1)
```

This request specifies that the generated content must express *commitment,* the lyrical theme, which the generator knows can be expressed by producing text that is associated with a tag called `commitment` (which authors have included in their Expressionist grammar). While the request does not include any tags that must *not* be collected, it specifies the relative desirability of `my job` and `my family`, with the weights in the tuples defining the former to be three times as desirable. To satisfy this request, the generator collects all the generable thoughts that have the `commitment` tag, and then uses the scoring metric to determine which ones best express `my job` and/or `my family`. While this is an example of targeted generation at runtime, one might also want to carry this out at authoring time. For example, an author may want to check that at least one thought *can* be generated from the grammar that expresses `commitment`, `my job`, and `my family`, or they may want to generate dozens of examples to check for content quality.

11.3 Symbolic Visibly-Pushdown Automata

Finite automata are a fundamental tool of thought in computer science; here I briefly explore a variety of increasingly sophisticated automata to obtain important support tools for expressive text generation, which I will discuss below. The role of automata in grammar-based text synthesis is undisputed, given the importance of the Chomsky hierarchy and its equivalences between (some) important classes of grammars and increasingly more powerful automata: regular grammars and the finite automata, context-free grammars and pushdown automata, and context-sensitive grammars and linearly-bounded Turing machines. Because Expressionist deploys a context-free grammar, I limit my attention here to the first two classes.

Finite automata can be seen as simple machines (defined as labeled directed graphs) that recognize whether a given finite input sequence belongs or does not belong to a language. Such an automaton has a set of *states*, of which I denote (without loss of generality) an *initial* state and some *terminal* states, along with a set of *edges* between pairs of states (self-edges are allowed) where each edge has an associated *symbol*. An automaton *accepts* a string if and only if it induces a sequence of *moves* along the edges starting from the initial state and ending exactly on a terminal state, without skipping any characters in the string or making any additional moves. The string is processed one character at a time, and a move is only allowed if the current character is exactly the same as the symbol of the prospective edge. While this is phrased as a decision problem, it is clear that one could also use a finite automaton to construct strings by traversing it as a directed graph, accumulating symbols as the traversal proceeds, and terminating when a terminal state is reached; any complete search algorithm would suffice.

Finite automata have some wonderfully useful properties: they admit efficient emptiness checking (it is easy to see if an automaton accepts no strings), and they are *closed* under intersection, union, concatenation, and complementation. Closure here means, for example, that the intersection of two finite automata (an automaton which only accepts the strings which *both* of the component automata would accept) is also a finite automaton. This means one can easily ask for all strings which are valid phone numbers but also have no repeating digits (intersection), or one can ask for words that do not end in "-ing" (complementation) with the exception of "fling" (union). In effect, this allows for manipulating potentially infinite-magnitude *languages*—the sets of recognized strings—in terms of operations on the finite automata. Unfortunately, the performance of many of these algorithms scales poorly with increasing numbers of symbols; ASCII is a challenge and full Unicode is unusably large.

The issue here is that an automaton needs one unique edge between states per distinct symbol. But all edges that go from one state to the same target state are in some sense equivalent—the predicate is not really a single character equality, but a set membership problem. *Symbolic* finite automata (SFAs) address and generalize this concern [39]. In a symbolic automaton, the machine is still processing a finite string of inputs, but each input comes from a potentially infinite set; to account for this, edges have associated *guards* which are predicates from a given Boolean algebra (a structure admitting disjunction, conjunction, and negation with designated true and false elements). The classic algorithms on finite automata generalize and maintain their closure properties, and because the number of edges remain small they can be extremely efficient. Symbolic finite automata have seen great success in the safety analysis of regular expressions, string sanitizers, and string-manipulating programs [273, 83, 23].

Pushdown automata (PDAs) generalize finite automata in a different direction: by adding some memory in the form of a *stack* and allowing edges to *push* or *pop* symbols or *peek* at the stack's topmost symbol. A pushdown automaton splits the alphabet into input symbols (as for finite automata) and *stack symbols,* and gives edges an associated action (which might be the null action); moreover edges can push a stack symbol or pop a specific symbol off the top of the stack. This allows them to recognize context-free languages involving for example matched parentheses or other counted pairs. Sadly, pushdown automata are not closed under intersection or complementation, although they are closed under intersection with regular languages. For the purposes of validating properties about grammars, intersection is extremely important (the above applications of symbolic automata relied on them extensively).

It is important to note that this lack of closure applies to context-free languages *in general*, but *some* pairs of languages can be intersected *without* leaving the realm of PDAs. Regular languages are one such subset; as it turns out, there is another set in between the regular languages and fully context-free languages called *visibly context-free,* recognized by *visibly* pushdown automata (VPAs), which is extremely useful for the purposes of this chapter [11]. *Visibly* here means that all stack operations (pushes and pops) happen because of specific, recognizable symbols in the input string (these are respectively *call* and *return* symbols by analogy to function calls). To be available for a transition, a return symbol must match the stack symbol pushed by the corresponding call. Accordingly, these are also called *input-driven* PDAs. Formally, a VPA has a set of states, initial states, terminal states, edges, input symbols, and stack symbols (like a PDA), but the input symbols are further partitioned into calls, returns, and *internal* symbols.

Not every context-free grammar designates its nonterminals with such opening and closing "parentheses," but any context-free grammar (CFG) can be *augmented* to output these as well (becoming a *visibly CFG*); it is straightforward to transform a visibly CFG into a VPA. The key transformation is that every nonterminal should begin with a call and end with a return, while every generated terminal should be an internal symbol. VPAs are closed under intersection and complementation. I end this tour of automata with *symbolic* VPAs, which, like SFAs, generalize edge conditions to arbitrary Boolean algebras [82]. These enjoy all the closure properties of VPAs but remain compact after intersections and other operations, and have recently been used to determine what control flows through programs might have elicited a particular error log [186].

11.3.1 Reducing Expressionist Grammars to Automata

Because of the expressive power of symbolic automata, and their successful application in many domains that seem similar to what was necessary for Expressionist, I wanted to apply an automata-based approach to Expressionist grammars as a solution to the problem formulated above. I have used the off-the-shelf library SVPAlib [81] because it comes with an implementation of SVPAs and several useful Boolean algebras. In this section, I explore the reduction of Expressionist grammars to SVPAs and how queries about generable content with targeted meanings, *i.e.,* desired tags, can be answered through an automata-theoretic lens. I aim for a *reduction* in the computer-science sense: properties of interest should hold on the SVPA if and only if they also hold on the original grammar. As an aside, most of the algorithms described here work for grammars and SVPAs which can contain loops, which are actually more

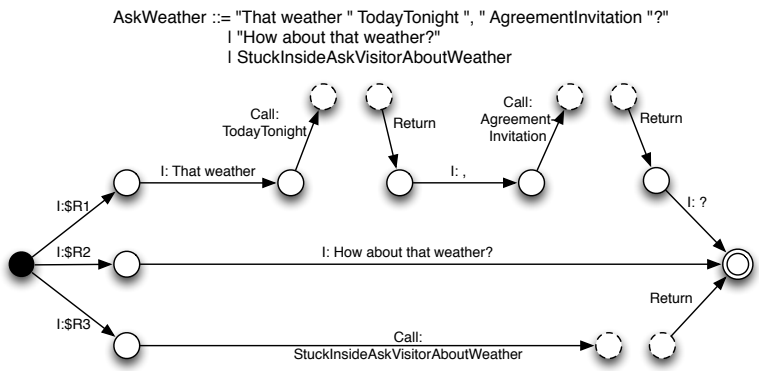

Figure 11.2: An Expressionist rule and corresponding SVPA fragment. Edges are labeled (I)nternal, Call, or Return; dotted circles show placeholder start and end nodes in the SVPA fragments built from other rules.

expressive than what Expressionist supports.

The goal is to create, for a given Expressionist grammar, a *deterministic* automaton. Many algorithms of interest require deterministic automata, and while every nondeterministic SVPA is determinizable, this operation can make an automaton much larger, slowing down future processing. The Boolean algebra used here is essentially a finite set algebra with a universe comprising every terminal, an identifier for every nonterminal, and an identifier for every Expressionist tag—in other words edges might have atomic predicates regarding set membership, set union, set complement, set intersection, and so on, with the overall edge predicates being Boolean combinations of those basic queries. A transition which generates a specific terminal, for example, will have as its predicate the singleton set containing that terminal. A transition which denotes a call into a nonterminal will have as its predicate a set containing the union of the nonterminal's identifier and the set of tags attached to that nonterminal. To create a fully deterministic automaton and to simplify the construction, I further augment the set of terminals by adding as many *rule selector symbols* as there are rule choices in the "widest" nonterminal

(e.g., if a grammar has a nonterminal with five production rules and no nonterminal with six, then there will be five rule selector symbols). I also add special *root selector symbols* for the possible choices among initial nonterminals (those designated as *top-level* in the grammar). The set of internal symbols is the set of terminals, while the set of opening and closing symbols are derived from the nonterminal identifiers.

For each nonterminal, I build an automaton *fragment* in the following way (illustrated in Fig. 11.2). First, I allocate an initial node and a final node (every node has a unique integer index; the stack symbols are just integers). Starting from the initial node, every *production rule* induces an internal transition to a fresh state with the corresponding rule selector symbol. Each production step of each rule is either a terminal or a nonterminal. If it is a terminal, I extend a new internal edge with that terminal from the current state to a placeholder state. If it is a nonterminal, I create temporary call and return edges with that nonterminal's identifier and add them to a list of references to be resolved later (the current state's index is used as the stack symbol). The call edge is extended from the current state and the return edge is pointed to a placeholder state. If there are more steps in the rule, the placeholder state is replaced with a freshly allocated state and I continue with the next step; otherwise, the placeholder state is replaced with the nonterminal's end state. I also make a pseudo-nonterminal for the root of the grammar with one production rule for each top-level nonterminal and construct its fragment in the same way.

To construct the full automaton, I start by connecting all the nonterminal fragments together by resolving the targets and sources of the temporary edges. The call edges are attached to the target nonterminal's initial node and the return edges are attached from the nonterminal's

final node. The automaton's initial and final nodes are the initial and final nodes of the root fragment.

11.3.2 Answering Queries

Enumerating all outputs of an SVPA is similar to enumerating outputs from a finite automaton. It must be extended in two ways: first, to allow for stack operations by including the stack in the search state and updating it during calls and returns; and second, to go from an abstract path where each element is a set predicate to potentially many concrete paths where each element is a terminal or nonterminal production. Some grammars are essentially not enumerable because they have so many possible outputs, but even these can be addressed well using something like iterative deepening search. Finding a single output is a special case; often, one wants to find the shortest output so as to apply a dynamic programming approach to find the fixpoint of the state reachability relation—this gives all-pairs shortest paths between states, and the system can use the first found path between any start and end state as its output.

I address the *targeted generation* problem by combining output-sampling with SVPA intersection using the standard product construction (as implemented in SVPAlib). The approach here is to create an SVPA for the *property*, for example an SVPA that recognizes sequences of calls with the desired tags, and then intersect that SVPA with the one derived from the grammar. The resulting automaton will *only* generate those outputs of the original grammar which have the desired property. I can enumerate outputs using the algorithm outlined above; moreover I can efficiently ask whether the automaton's language is *empty,* i.e., whether it is impossible to generate an output with the given property.

To handle cases where queries also contain forbidden tags, I consider two low-level properties: a tag is present somewhere in the output or it is absent. If Reductionist can describe these low-level properties using automata, it can proceed to create any combination of present tags and absent tags by intersecting these together. A tag-present automaton comprises one initial state and one final state; there is an edge from the initial state to the final state when a call matching the desired tag is made. All other possible transitions are embedded as self-loops from initial to initial and self-loops from final to final. In other words, the automaton advances to its terminal state when the tag is found. The tags-absent automaton is identical, still built of two states, but initial state and final state are the same state, while the other state which is the target of the tag-matching call is not final and has no path back to a final state.

I can also build more sophisticated property automata describing any visibly pushdown property: some tags in order, no repetition of a particular production rule or terminal, some set of tags if and only if another tag is found, matched pairs of tags (e.g., every problem is eventually followed by a matching resolution), and so on. Testing any property is roughly as easy as testing any other property. This is a key benefit of the automata-based approach: one algorithm solves arbitrary expressible property queries.

Output counting has a more efficient algorithm than just exhaustive enumeration. I implemented the standard dynamic programming formulation of finite automaton path counting, extending it for the symbolic setting by letting a single edge induce multiple paths (as described above for output enumeration) and for the pushdown setting by tracking stack state as well as automaton state and matching calls and returns. Briefly, I want to know how many strings are in the automaton's language; in other words, how many paths there are from initial to final states.

Because in general automata may have loops, I rephrase the question to ask how many strings of length exactly k are in the language, and sum from $k=0$ to $k=k_{\max}$. Expressionist grammars may not have loops, so the algorithm takes the longest path through the induced SVPA as a bound on k.

Begin by allocating a matrix M of dimensions $k \times |S|$, where S is the set of states in the SVPA. Each entry in the matrix is a set of (Q,N) pairs with Q a stack state (i.e., a list of integers) and N an integer. $M_{0,s}$ is initialized to $\{((), 1)\}$ if state s is an initial state and $\{\}$ otherwise. For each value of k up to the limit, fill in the next layer of the matrix based on the previous layer, with the values of the states whose edges enter each state s at $k-1$ contributing to the values of $M_{k,s}$. Each state's (Q,N) values for a given k represent how many ways a state can be reached from the initial configuration within k steps. If at any time new ways to reach a final state are added, accumulate those as part of the final return value. Additionally, accumulate how many strings there are of each intermediate length. In this way, one can determine the size of grammars with over 10^{64} possible outputs in seconds, which Table 11.1 shows (as explained below). I refer the interested reader to the source code for more detail [189].

11.4 Evaluation and Discussion

To evaluate this approach as a practical solution to the problem formulated above—satisfying content requests specifying the set of tags that generated content should have (required tags) and the set of tags it should not have (forbidden tags)—I applied it to four actual Expressionist grammars that have been authored for videogame applications (and range in size).

Grammar	Creation	Counting
Small	1.9	3.4
Medium	6.2	5.1
Large	2.3	2.9
Extra-Large	24	17

Table 11.1: Times (in seconds) to create an automaton from representative grammars and to count its possible outputs.

Difficulty	Intersection	Check
Easy	0.2	0.4
Moderate	0.01	0.02
Hard	11.3	19.0

Table 11.2: For each test case, times (in seconds) to intersect each constructed automaton with properties of interest (corresponding to realistic content requests) and to check for (and produce) a satisfying output.

The first grammar was authored for character dialogue generation in the ongoing project *Talk of the Town* [209]; it has 2.8M generable outputs, which in the context of Expressionist makes it a *small* grammar, and its expressive range spans 333 *expressible meanings*, i.e., unique sets of tags that may be attached to generated outputs. The *medium* grammar was authored for the hacker level of the released game *Project Perfect Citizen,* honorable mention for the Independent Games Festival's Nuovo Award; it has 65 quadrillion generable outputs and 37 expressible meanings. The *large* grammar was authored for character thoughts in the in-development game *Juke Joint* [208] and features 6.3 quintillion generable outputs and 1,918 expressible meanings. Finally, the *extra-large* grammar was authored for an experimental natural language generation system for *Talk of the Town* [246] has a staggering 10^{64} outputs and an unknown number of expressible meanings (the algorithm previously used by Expressionist to count these exhausts available RAM and fails).

To carry out the evaluation procedure, Reductionist first reduced each grammar to a

SVPA using the method described above and then counted the number of generable outputs for each (which could be a useful authoring metric). Table 11.1 shows how long this procedure took for each grammar on a workstation laptop from late 2012 (2.6 GHz Intel Core i7 CPU, 16 GB RAM). While the durations may appear large, note that this procedure only has to be carried out once prior to any session of querying the resulting SVPA. For considerations of targeted generation at runtime, this could still happen offline, e.g., during compilation of the game code. When it comes to authoring support, however, waits of several seconds could be too long if each edit is followed by a fresh reduction of the grammar. Interestingly, the medium grammar took longer to reduce than the large grammar—this must be due to structural characteristics of each, but further investigation remains for future work.

Next, for each grammar, I formulated an example content request that could realistically be encountered at runtime and demanded content that satisfied this request (these examples are available alongside the source code as part of its unit tests). These content requests represented three increasing levels of technical difficulty: as an *easy* test case, a content request for the small grammar with two required tags and two forbidden tags; as a *moderate* test case, a content request for the medium grammar with two required tags and two forbidden tags; and as a *hard* test case, a content request for the large grammar with three required tags and three forbidden tags. To execute a test case (and thereby satisfy the example content request), Reductionist must first intersect the reduced grammar SVPA with a property SVPA (capturing the required and forbidden tags at hand).

Table 11.2 shows how long this method took to perform the intersection and checking steps for satisficing outputs for each test case (note that checking also yields a witness output

satisfying the property). Again, there is unintuitive nonlinear behavior, with the easy test case taking an order of magnitude longer than the moderate case; this also requires future investigation. More troublingly, the results demonstrate a sharp threshold between content requests that can be satisfied extremely quickly and ones that take far too long. Note that intersection of more than one property on the extra-large grammar was not possible due to extreme memory usage. Finally, I attempted to check harder test cases on the smaller grammars (with content requests containing 15-30 total stipulated tags), but these also timed out. All of these results suggest that the structure of the grammar is very important to the success of the method, and in particular the performance of automata intersection algorithms is key. Initial profiling suggests that the timeouts and memory exhaustion occur *after* the intersection algorithm is complete, when checking the intersected automata for unreachable states to minimize its size; perhaps there are some shortcuts that can be taken here to improve performance.

11.5 Conclusion and Future Work

The method I have introduced here, which works by reducing Expressionist grammars to symbolic visibly pushdown automata, represents a first demonstration of targeted generation in pursuit of multiple requested meanings. As mentioned above, earlier work could only target a single required tag, and for additional desired tags the heretofore leading approach resorted to a greedy search that could not guarantee that generated content would have all the requested meanings. While the results here represent a clear leap forward in grammar-based text authoring, the evaluation revealed some troubling nonlinear qualities in the computational task, which

manifest in a mysterious threshold between trivial and intractable test cases. This may be due to the specific library being used, or there may be inherent considerations of computational complexity; in future work, I will deeply investigate these matters, considering alternative related approaches. Keep in mind, however, the magnitude of technical challenge that this task represents. The moderate test case, for example, requires implicitly searching through a space of 65 quadrillion possible outputs to find one that has two specific properties and does not have two others. This is a monumental search task, and Reductionist carries it to completion in 0.02 seconds. Still, while it can check moderately complex properties on complex grammars, it cannot yet check complex properties even on relatively simple (order 10^7) grammars.

Another avenue for future work pertains to a feature of Expressionist that I have not yet mentioned: authors can attach to production rules *probabilities of application*, which makes the underlying grammar formalism a *probabilistic* CFG. As such, I would like to extend the SVPA approach to handle that setting, both for design support (concerning the likelihood of generable content with properties of interest) and runtime support (probabilistic generation). Moreover, some Expressionist authors have exploited the free-text nature of tags in the tool to make use of special *condition logic* tags, which gate the expansion of nonterminals according to the system state [210]. Handling the semantics of preconditions is another possible extension, especially since SVPAs offer symbolic guards which can be from any Boolean algebra.

Text generation in games and interactive storytelling is typically carried out *iteratively:* outputs may occur in series, or may be constrained according to the context or earlier outputs, or may affect the generation contexts of future outputs. As such, I am beginning to think about how an SVPA could capture these dynamical aspects of text generation to provide

even more advanced design support at authoring time—for instance, about the space of possible *sequences* of generated outputs.

While I have focused on Expressionist so far, due to its reification of properties of interest as tags, I anticipate this approach being applicable to other tools and methods, too, such as Tracery and templated dialogue. Here, the work will be in determining a set of properties of interest and a method for discerning them. For example, in lieu of Expressionist-like tags, a system could recognize surface characteristics of terminal symbols in Tracery (such as their length) and then build property automata whose edges are associated with these properties. This would make it possible to generate outputs with desired surface characteristics.

I am particularly interested in the artistic affordances of *grammar sculpting*. By this, I mean the direct construction of new grammars using the operations of intersection, union, concatenation, and subtraction, since these can be applied between pairs of grammars just as I have done with property-grammar pairs. An author could form a permissive grammar and subtract out a smaller grammar of repetitive phrases or unpleasant juxtapositions; or combine two grammars on related themes to obtain a new grammar synthesizing those themes, perhaps along their shared tags or nonterminal IDs; or instantiate the same base grammar differently for different regional dialects in a fictional world (or indeed, for different characters in the world). While this would be very experimental, especially since the resulting grammars may only be machine understandable, I already imagine some interesting use cases—for example, authoring a permissive grammar and then automatically subtracting out classes of undesirable outputs.

This chapter shows how to leverage a specific connection between an operational logic and a corresponding formal mathematical system (in this case, visibly pushdown au-

tomata); while constant factors and engineering concerns got in the way of a total success, Reductionist does offer some useful affordances not present in the *ad hoc* tools constructed for Expressionist and remains a valuable complementary tool.

Games rarely consist of just a single logic, so the next chapter showcases HyPED, which is founded on a more complex composition of operational logics commonly referred to as *graphical logics.* The schema of picking a combination of logics, selecting appropriate formal logics using Table 9.1 as a guide, and then building tools around that formal representation is *repeatable* and productive.

Chapter 12

Solving Action Games with HyPED

Recent work in game design support has successfully argued that games' emergent qualities—the chaos that results when players interact with games' complex rules—leave a substantial role for automation in the game design process. The educational puzzle game *Refraction* used model checking to ensure that all solutions to a puzzle required the use of necessary mathematical concepts [233]. Some continuous-time games incorporate such solution-finding techniques into their game design itself: *CloudberryKingdom* generates new game levels on the fly but ensures that they can be won by a player with bounded reaction time [99]. [1]

Besides solution search, one general approach has become increasingly popular in recent years: visualizing (approximations of) reachable regions [32, 218, 263, 123]. In this chapter, I show how to improve the availability of this technique without committing to specific game engines or game-making tools. The key trick here, as in the rest of this part of the dissertation, is to use operational logics as a starting point.

[1] *Portions of this chapter previously appeared as "HyPED: Modeling and Analyzing Action Games as Hybrid Systems" [194].*

I follow after the game description language *cum* game engine *cum* model checker BIPED [230], which leverages declarative game specifications to enable both human-playable prototypes and machine-analyzable models. BIPED is specialized for games with discrete state spaces and fairly coarse discrete time. What would a BIPED-like system for continuous-time games with hundreds of real-valued variables look like? I answer this question by combining folk approaches of action game makers—state machines, object pooling, collision detection, game physics simulation, and so on—with formal approaches from the hybrid systems literature and recent work in action game design support.

HyPED (Hybrid BIPED) is a modular formalism for defining action game characters, grounded in the theories of operational logics [166] and hybrid automata [12] (*hybrid* here refers to the fact that these systems hybridize continuous and discrete behavior). HyPED models game entities from *graphical logic* games, privileging collision logics, physics logics, and entity-state logics. These games center simulations of continuous space and time, collisions between objects, and objects with small sets of discrete variables whose behaviors change under different circumstances (generally indicated by changes in visual appearance). HyPED entities and environments can thence be readily converted to Unity or other game engines which center graphical logics.

Action games (whose mechanics focus on movement and collision detection) constitute arguably the most prevalent genre of digital games. Accordingly, many popular game-making tools focus on them: *GameMaker, Unity 3D, Unreal,* and other systems offer user interfaces and programming APIs supporting collision checking, simple physics, and in some cases state machines. At the same time, none of these offers significant support for model checking,

visualization of possible system states, parameter synthesis, or other features that the available domain knowledge would seem to enable. These tools could be of great use to game makers, reducing the need for expensive manual testing and design space exploration.

In this chapter, I define the action game modeling language HyPED and show examples of the diverse entities it can define. I also show how my undergraduate mentee Brian Lambrigger applied incremental search (specifically, Rapidly-exploring Random Trees) to the reachable-regions problem for HyPED games (a more expressive class of games than those to which these techniques have been previously applied). I further illustrate how deep knowledge of system dynamics can improve the performance of RRT for games, mainly through dimensionality reduction.

12.1 Related Work

Some game-making tools do support partial declarative definitions of game entity behaviors. GameMaker provides for entities with behaviors driven by events (such as collisions or timer elapse) that trigger handlers supporting various conditional responses (e.g., changing velocity or incrementing a variable). These entities also have varying animations at different times and collision areas which correspond to the animations. Unfortunately, game entities with atomic behaviors outside of that predefined set are inexpressible in the declarative style; the GML scripting language is provided as an imperative escape hatch for such cases.

Game designers often narrate entities' behaviors in terms of state machines, and this trend is captured in both the academic [226] (HyPED may be seen as a rigorous grounding

of the same fundamental idea) and game design literature [247]. The 3D game-making tools Unreal and Unity both provide for explicit state machines specialized for character animation, but in both engines most atomic behaviors are implemented in imperative code, and none of the above tools has a formal semantics.

This state machine-like description addresses entities' physical dynamics, audiovisual representation, and discrete variables like health or ammunition. These are not formal transition systems: they may have undecidable transition relations (due to various combinations of dynamics and transition guards) or their discrete state spaces may be infeasibly large. Some of these variations can be captured by other types of automata, but games generally employ state machine-flavored discrete systems and not formal automata.

12.1.1 Hybrid Automata

Hybrid automata combine a discrete transition system (a finite state machine) with a set of continuous variables and a switched set of differential equations over these variables called *flows* [12]. In each state, a different subset of the flows is applied to the continuous variables until the state is exited along a transition, which may be guarded on continuous or discrete variables; transitions may also instantaneously modify variables' values. The usual semantics for hybrid automata is that they alternate between periods of continuous flow (called *delay* or *continuous transitions*) and instantaneous *discrete transitions*.

Hybrid automata have seen extensive use in modeling cyber-physical systems where linear (or simpler) dynamics adequately describe the partially known or complex true dynamics of a system whose behavior is different at different times. They have many varieties and syn-

tactic extensions, and often their dynamics are restricted in one way or another [114]. These restrictions are helpful because even under very simple physical laws, the question of state reachability becomes intractable or even undecidable (though semi-decision algorithms exist).

Key analysis questions here include:

- *Safety* or state reachability (safety is often phrased as never reaching an unsafe, e.g., illegal or stuck state);
- Calculating a *reachable region* (what possible values can the continuous variables take?);
- *Controller synthesis* (is there a control policy or AI player which will be safe or meet some optimality criterion?);
- and *parameter synthesis* (given an automaton with some unknown parameters—jumping speed, gravity, platform height—can a tool find values for those parameters satisfying some constraint?).

Hybrid automata are readily identified with action game characters thanks to Table 9.1: action game characters focus on collision, physics, entity-state, and control logics, pointing towards, respectively, a constraint language, differential equations, finite state machine, and an action language. Hybrid automata are essentially the composition of finite state machines with differential equations, where transitions are taken or not based on a Boolean algebra (in this case, the composition of spatial collision constraints and instantaneous control actions). HyPED comes directly out of this observation.

12.1.2 Hierarchical Hybrid Automata

Game entities often have highly structured behaviors; Super Mario's grounded movement comprises walking, running, and standing still, while his aerial movement has distinct rising and falling behaviors composed in parallel with moving left and right in midair. Modeling these distinct flows requires dozens of states, most of which are slight variations on each other.

Game designers generally handle these overlapping concerns by adding hierarchy and concurrency to their state machines. This may be implemented as sets of Boolean variables with allowed and disallowed combinations (exactly one of `on_ground`, `jumping_up`, and `falling` is true at a given time, independently of `crouching`). These approaches to modularity have also been explored in hybrid systems; the CHARON language [13] is a very generic approach with well-defined semantics.

Besides the complexity of individual entities, hundreds of game entities may be created and destroyed during play; this alone puts a significant stress on existing hybrid automata tools which generally scale poorly with the extremely high-dimensional state space. Hybrid systems researchers call adding or removing automata from the system at runtime "reconfigurability," and CHARON's derivative R-CHARON addresses this along with ways for automata to reference each other explicitly [140]. HyPED can be seen as a specialization of R-CHARON for games.

While hybrid automata seem to be a natural fit for modeling action games, previous attempts to apply them have failed to scale either in terms of expressiveness or performance past

toy examples due to limitations of the modeling languages and tools used [5]. I believe HyPED resolves many of these issues.

12.1.3 Rapidly-exploring Random Trees

Rapidly-exploring Random Trees (RRT) is a commonly-used incremental search algorithm for planning in high dimensional spaces [146]. The basic RRT algorithm is to repeatedly sample points from the *configuration space* $\mathcal{C}_S$ of possible combinations of values of all the variables in the world, finding each time the closest node in the tree to the sampled point and growing it towards the sampled point; if this can be done without violating any constraints, the resulting node is added to the tree and the process continues. In the case where the goal is to search for all possible states rather than a specific path, RRTs are useful because they quickly expand the tree to cover the state space. This is due to their implicit *Voronoi bias*: newly sampled states are most likely to be in the largest Voronoi region induced by the tree built so far. Additionally, for problems where a solution is required and traditional searches are not feasible (e.g., in high-dimensional state spaces), RRTs are an attractive choice because they require little knowledge of the actual dynamics of a system.

RRTs have previously been applied to the reachability question for hybrid automata [45]; most previous work in this area uses a lexical distance metric and assumes a single automaton. In the case of HyPED, there is a high penalty for mismatched discrete modes, but lexical distance is not used because not all mode transitions of the same transition system distance are equally easy to make (consider taking a door from the `locked` to `unlocked` state versus taking a character from `standing` to `crouching`: both are one-step transitions but the latter is much

easier to realize). Note also that for HyPED, constraint violations are conditions such as player death or going out of bounds, rather than assuming any collision is a constraint violation—it does not model robots moving around a factory floor, but game characters through a level.

HyPED also forgoes an accurate cost-to-go metric (used to locate the nearest node in the tree), finding other ways to maintain the Voronoi bias that makes RRT effective. This is achieved in part through extensions to the RRT algorithm which are built for that purpose, and in part through analysis of the hybrid system to find a tight approximation of the reachable configuration space, preventing infeasible states from being sampled.

12.1.4 Game Design Support

RRTs are commonly used for pathfinding in games [10]. These approaches generally work on an abstraction of the concrete dynamics, e.g., by assuming a top-down 2D world where entities can move freely in straight lines, ignoring dynamic objects in the environment. In general, however, dynamic objects may not simply be obstacles to avoid, but also bridges or platforms that *must* be collided with in order to reach a goal location. This is a marked departure from traditional application areas for RRT, and one which merits further study in the realm of simulation and state space exploration.

RRTs have also been used to aid in level design and solution verification as a substitute for time-consuming testing by designers or playtesters. They are useful in these cases because their bias towards exploring the state space completely accounts for the wide range of possible states a player may find themselves in.

Bauer *et al.* used RRTs to trace jump trajectory and landing position of game agents

in *Treefrog Treasure* [32], creating a visualization of reachable states and paths available to the player throughout the level. This implementation, however, required advance knowledge of the parabolic motion of the character and the static world geometry. In contrast, the combination of HyPED's modeling formalism and RRT presented here allows for a wide range of both level and entity dynamics.

Tremblay *et al.* used RRT in more dynamic environments [263]. This work experimented with various search algorithms, most notably the combination of RRT and local goal-directed planners like A* and Monte Carlo Tree Search. Some of the test environments had dynamic level geometry, such as moving platforms, to add a degree of difficulty above that of Bauer's work. Like the *Treefrog Treasure* work, planning was performed on an abstraction of the game character's true physics.

As Tremblay *et al.* noted, their RRT algorithm only samples in $\mathbb{R}^2$ (a designated player character's x and y position) and compares 2-D Euclidean distance, losing completeness (or at least causing extremely slow convergence) for levels with reactive elements. Many games have design elements whose state depends on player input, like collapsing bridges or moving platforms that stay inert until the player touches them. These reactive elements are troublesome for search algorithms as they exponentially increase the state space, preventing successful search if not carefully considered in state sampling.

Consider, for example, a sampled state on the other side of a gate which depends upon a switch to change the `locked` state of the gate. RRTs may try to expand nodes which, while close by euclidean distance, cannot progress without first backtracking to the switch; if the discrete state of the gate is not taken into account in sampling or the distance metric an RRT

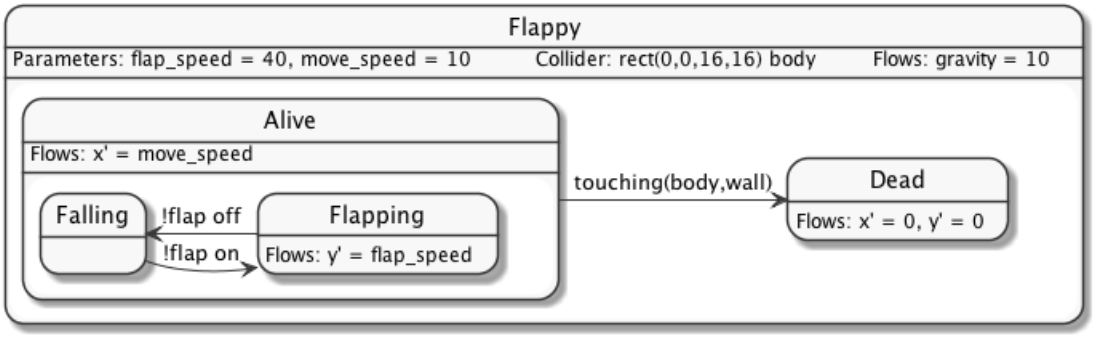

Figure 12.1: Flappy Bird.

planner could easily become trapped. This expansion is addressed here by sampling in the full state space while simultaneously reducing the dimensionality of this space as much as possible via static analysis.

12.2 HyPED

The main design goal of HyPED is to translate concepts from hybrid control theory to the theory of action games so that tools and techniques from the former can be applied in the latter. There are substantial differences between classical hybrid automata and game character state machines, some of which have been detailed above. Here, I present a high-level account of HyPED's syntax and semantics. This explanation is self-contained, but more complete documentation can be found in the HyPED source code repository [194].

My immediate goal in adapting hybrid automata to action game characters was to reduce repetition using hierarchical and parallel composition of behavioral modes. Fig. 12.1 illustrates hierarchical (but not parallel) modeling with a simple *Flappy Bird*-like entity. Flappy

has two mutually exclusive top-level states: `alive` and `dead`. In the `alive` state, the entity moves to the right according to the value of its `move_speed` parameter (this type of continuous variable evolution is called a *flow*); this state is itself split into `flapping` and `falling` states which alternate when the `flap` button is pressed or released respectively. Implicitly, Flappy's acceleration in y is `gravity`, but this is overridden by the `flapping` state which fixes y velocity.

Flappy has one square-shaped `collider` positioned at the entity's origin (entities may in general have several colliders, which can be conditionally active or inactive) tagged with the `body` type; when it touches something tagged `wall` the entity transitions into the `dead` top-level state no matter whether it's `falling` or `flapping`. Metadata in the HyPED game definition describe which tags are checked for collision with which other tags, and whether collisions should also reset the implicated velocity components to zero.

Flows defined in a state also apply in its descendants unless overridden (as in the `dead` state's $y' = 0$, which overrides the effects of gravity). Compared to classical hybrid automata, HyPED automata are extended mainly by admitting external theories like collision and button input.

For a more complex example, consider Mario (Fig. 12.2). Mario has several concurrent modes, including his `size`, whether he has temporary invulnerability after being `hurt`, and his `movement` (separated by dashed lines). The flows of a entity with multiple active parallel modes are the union of those flows; conflicting assignments to a single variable are illegal (in the case of Mario, only the movement mode applies any flows). To keep this rule straightforward, HyPED only allows concurrency at the top level of an entity.

The `Size` behavior group introduces two new concepts. First, whenever the entity

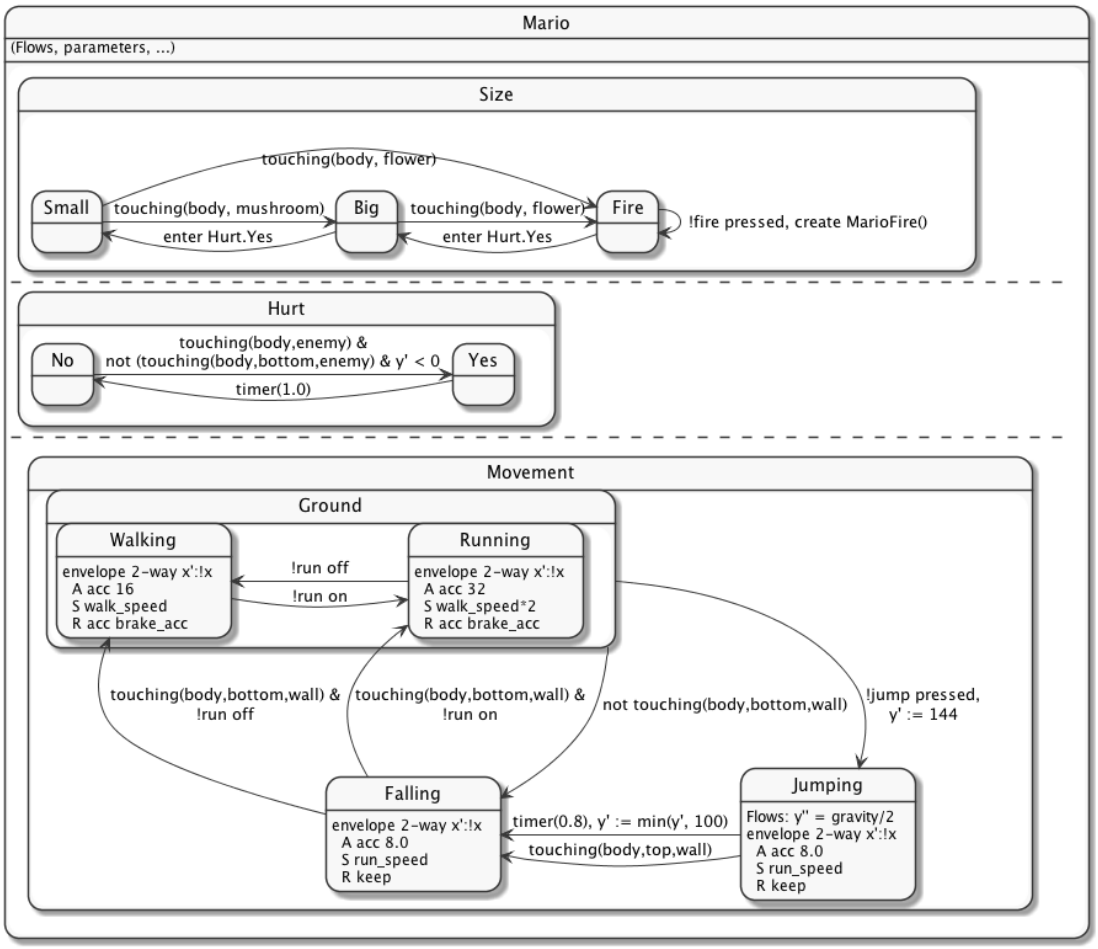

Figure 12.2: Super Mario (abbreviated).

`enters` the `Hurt.Yes` state, it becomes smaller; dying is the responsibility of a different behavior group (not shown). Second, the self-transition in the `fire` state produces a new `MarioFire` entity at Mario's current position. The `Hurt` group also introduces a new type of edge condition: the `timer`, which becomes true if the edge's origin state (in this case, `Hurt.Yes`) has been active for more than the given amount of time.

Finally, the `movement` group in Mario is much more complicated than it was for Flappy. The main behaviors here are `ground` movement and the two aerial modes which differ in gravity and air control. Note that these states do not for the most part define flows directly, but instead define *envelopes* in the sense used by Swink [247]. These attack/decay/sustain/release envelopes, inspired by audio processing and control theory, are like miniature automata on their own. They characterize (in this case) the left/right mirrored control over x velocity typical of platform game characters with acceleration (attack phase) up to a maximum speed (sustain phase) which quickly decreases to 0 when the buttons are let go (release phase). A character like *The Legend of Zelda's* Link would be moved by a four-way envelope activated by the `x` and `y` control axes and controlling Link's x and y velocity. Note that while Mario's ground movement has a deceleration when buttons are released, aerial horizontal movement `keeps` the current velocity instead.

The last new feature in this diagram is the edge update (e.g., $y' := \min(y', 100)$) which instantaneously changes a variable's value during a transition. This is used to set Mario's initial jumping speed (on the transition into `jumping`) and also to clip that speed when the jump button is released to get Mario's characteristic fine jumping control.

A HyPED game (or *world*) consists of a number of entity definitions as described

above along with an *initial configuration* defining static level geometry (e.g., tilemaps, possibly imported from the Video Game Level Corpus [240]) and the initial *instances*, if any, of each entity definition. HyPED further divides up the world into distinct *spaces* which are linked together in the style of *Zelda* rooms, each with its own population of entities.

HyPED is very expressive; it can describe not only entities like those above but also completely different ones including Link, moving platforms, doors, *Treefrog Treasure's* jumping frog, a top-down racecar, and even (in theory) three dimensional entities and spaces, though this latter case is not yet supported by the interactive player program. To support these and other entities, HyPED also provides for references between entities (as in R-CHARON) and for discrete variables which may only be updated during transitions. Discrete variables provide for quantities like ammunition or keys, which is important if we want to incorporate resource logics into the model.

12.2.1 Static Analysis

Because HyPED has deep knowledge of game entities' discrete and continuous dynamics, it can derive useful information to simplify game entities or to improve performance. For example, it is straightforward to combine the dimensions of static level geometry with information about entities' colliders to determine what positions they might be able to inhabit. It can also leverage the structure of entities to generate fast cache-aware data representations or low-level code.

Consider a horizontally moving platform which smoothly animates left and right between two fixed positions using a `left` state and a `right` state. Some useful properties are

immediately obtained: it is clear from the flows of each state that this platform's y position will be constant and that its x velocity will always be one of two fixed values. HyPED also can learn (by looking at the transition guards) that the platform's x position will always be within a fixed interval. This information helps reduce the dimensionality of configuration space, which leads RRT to find better solutions faster without losing completeness. Keeping the sampled configuration space closer to the true reachable space of the system is a good way to maintain Voronoi bias.

In the future I hope to extend this to approximate entities' behaviors, e.g., by finding closed-form equations for cycles through an automaton; this could be used to turn Mario's jump into a (piecewise) parabola suitable for model-predictive control, reducing the number of discrete dimensions of the system. I am especially excited at the prospect of generalizing these techniques to help generate hierarchical plans, treating a game like *Zelda* as the combination of a low-level moving-and-attacking game with a high-level graph navigation game. Smoothly climbing up and down between different abstractions of a game entity is key to this approach, as is finding concrete propositions which, if they were to become true or false, would help achieve an intermediate goal. HyPED models contain the information necessary to perform these sorts of transformations. This can help achieve the goals of modular and compositional analysis set out in Chapters 9 and 10.

12.2.2 Translation to Unity

Although it sports an interactive player application, HyPED is not a complete game engine—that is explicitly *not* a goal of the system. HyPED is for prototyping and analyzing

game worlds and entities. I expect that HyPED entities and levels will be exported or translated to other engines once their design is sufficiently well-understood; here I outline that approach for the Unity engine.

The static geometry currently used in HyPED is a simple tilemap for which a translation to Unity is obvious. This section will focus on translating HyPED entities to Unity prefabs. An entity combines several concurrent top-level behaviors with some colliders from a fixed set, of which only some might be active at a given time. These colliders correspond to e.g., Unity `BoxColliders` and have similar parameters. I imagine that one `MonoBehavior` per HyPED entity type could manage collider positioning relative to the entity and activate or deactivate these colliders as necessary.

Posit one additional `MonoBehavior` subclass per concurrent top-level behavior group; it should carry a bitfield or a set of Boolean values describing which child states are currently active. In `FixedUpdate`, each active mode's continuous flows and envelopes are applied and transition guards are checked; if any guard is satisfied, that transition is taken (updating the active states) and instantaneous updates are applied, skipping the rest of the guards. The behavior class should also track which modes were entered and exited during this update cycle. The low-level implementation of these classes should closely mirror the Python version.

The Unity object obtained by assembling these behaviors and colliders together should exactly realize the HyPED definition. Game-specific aspects including graphical properties can be added on top of this base object.

12.3 Dynamic Analysis via RRT

A key argument for formal modeling languages is the ready availability of algorithms and software tools for querying or verifying properties of the model. Incremental search has proved successful in analyzing dynamical systems, so I wondered how well these techniques would work in this somewhat more complex hybrid domain.

I explored three improvements to RRT for high-dimensional or highly-constrained domains. Resolution-Complete RRT (RC-RRT) back-propagates failure signals similarly to Monte Carlo Tree Search in order to prune unsuccessful branches from consideration [53]. In the scenarios presented here, dynamic level geometry creates many situations which can trap progress. With RC-RRT, branches leading to bad states are less likely to be expanded, improving coverage.

Reachability-Guided RRT creates an approximate hull of reachable points around the leaves to help determine if, for any sampled state and nearest node, there is an input which can control the node to the state [219]. This is useful for complex constraints as it prevents the selection of nodes that, while close to a sampled state, may be blocked or trapped.

Environment-Guided RRT (EG-RRT) uses the back-propagating failure of RC-RRT in tandem with the reachability approximation of RG-RRT [126]. This improves search performance in situations where both alternatives fail. The test cases contain both complex hybrid dynamics and non-static geometry, so one would expect EG-RRT to perform the best here as well.

12.3.1 Refined Sampling

Because HyPED knows the system dynamics, it can constrain the configuration space without losing completeness. Naively, one might imagine that each entity with 2D position, velocity, and acceleration along with a discrete mode would require six continuous dimensions and one discrete dimension in configuration space. If the system knew in advance that some entity was (e.g.) a platform that only moved with constant positive or negative velocity between two x coordinates with a fixed y position, it could easily reduce this down to one continuous dimension with narrow bounds (x) and one Boolean discrete dimension. HyPED calculates such reduced spaces automatically by simple inspection of flows after constant propagation in each discrete space. The reduced space $\mathcal{C}_{SR}$ is still an over-approximation (no actually-valid states will be outside $\mathcal{C}_{SR}$), so this simplification does not lose completeness.

12.3.2 Evaluation

For testing, I crafted several high-dimensional planning problems with non-trivial solutions, both to test the expressiveness of HyPED's models, and to evaluate solution planning in the complex spaces afforded by hybrid automata representation. Each RRT-based planner (RRT, RC-RRT, RG-RRT, EG-RRT) was given a budget of 20 seconds to find solutions in the test levels, with the average time to find a solution calculated and recorded in the following tables. I also tried variants of these algorithms which only sampled in $\mathbb{R}^2$, but these were strictly dominated by sampling in the full configuration space.

Compared to Tremblay and Bauer's results, computation times are higher. This is in part because HyPED works on real (and arbitrary) system dynamics of more complex characters

and environments, and in part because the initial implementation had some constant-factor inefficiencies that have since been resolved. For similar reasons, Brian and I could not successfully apply A* as a local planner.

12.3.2.1 Test 1

Test 1 (see Fig. 12.3) introduces reactive level elements with a simple timing puzzle. Starting on the far left, the agent must navigate across the platforms to the colored goal square on the right. The agent itself may only move right, left, or stay still. Platform 2 (in blue) on the middle-right continually moves between the middle of the gap and the other island, while Platform 1 (in brown) remains stationary until the agent touches it, after which time Platform 1 acts similarly to Platform 2. To navigate to the goal, the agent must time their boarding so that Platform 1 meets Platform 2 in the middle of the gap, then board Platform 2 and ride it to the goal. Humans can do this after a few tries, but heuristic search performs very poorly since it must do nothing for a while before making progress (it is a surprisingly challenging problem despite its visual simplicity).

In these tests, EG-RRT succeeds most consistently (see Tab. 12.1); the failure propagation metric prunes any branches which fall off the level or activate platform 1 at poorly-chosen times. EG-RRT's success is further increased by the refined sampling described above.

12.3.2.2 Test 2

In Test 2 (see Fig. 12.3), the difficulty is increased further, as the timing puzzle introduces both an obstacle and a means of progress. In this test, the agent is a Mario-like agent

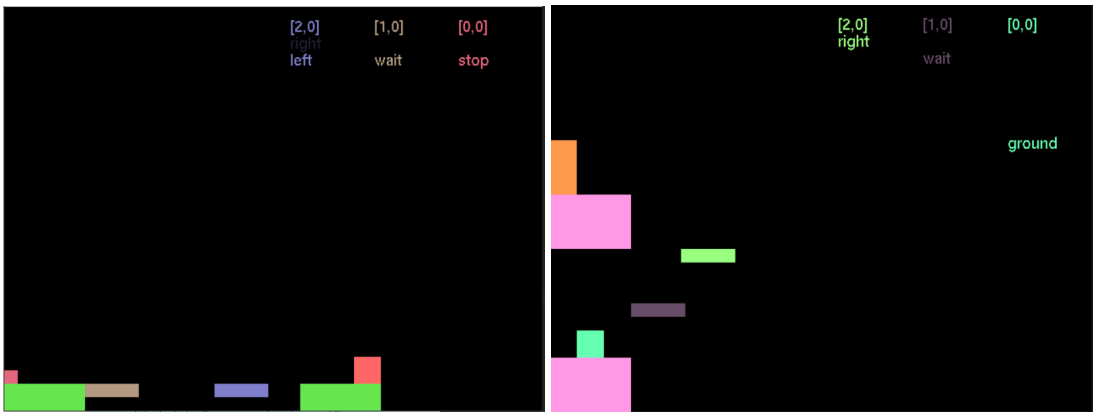

Figure 12.3: Tests 1 and 2.

which can jump with complex dynamics. Once again, Platform 2 (in green) moves horizontally between the upper geometry and a position about 200 pixels to the right. Platform 1 behaves as before, remaining at rest until touching the player.

In this test (see Tab. 12.2), EG-RRT and RG-RRT perform poorly without reduced bounds due to the fact that if the sampled state is not closer to the reachable hull than to some node in the tree, they will choose a new sample state and begin the process again, evidenced by the much smaller number of nodes explored compared to RRT and RC-RRT. Only when the state space is reduced does EG-RRT once again outperform the rest; it improves even more than the other algorithms do. It is interesting to note that (at least for this problem) the less-informed search algorithms can obtain results much faster when they are able to obtain results at all, but their success rate is much lower.

Naive Bounds				
	#Nodes	Goal Reached		#Success
		Time	#Nodes	
RRT	1087	4.87s	334	27
RC-RRT	1083	4.94s	308	22
RG-RRT	580	3.34s	109	20
EG-RRT	771	7.91s	315	32
Reduced Bounds				
RRT	1048	3.31s	219	24
RC-RRT	1018	3.11s	219	24
RG-RRT	505	2.44s	69	13
EG-RRT	814	9.8s	365	46

Table 12.1: Results for Test 1; timeout at 20 seconds (50 runs)

12.4 Future Work

Bringing down constant factors and getting more RRT exploration steps for a given time budget is an obvious area of improvement. The results above used a naive implementation of RRT nearest-neighbor finding to make it easy to try out variations on the algorithm; improving this is also obvious and should give a substantial performance boost [22]. When I switched from a graph-of-objects representation of the state space to a set of flat arrays, I saw roughly 70x wallclock-time improvements which translates into much more successful rates of search for a given time budget. I also introduced a scalable variation on nearest neighbors which sampled N points randomly and returned the nearest of those, where N scales with the logarithm of the population size. Using a different distance function also seemed interesting: the work above used squared distance, but this is not a good choice for very high-dimensional data.

Later, I would like to further refine the configuration space by deeper static analysis and also to explore non-incremental-search approaches to the reachable-regions problem (for

Naive Bounds				
	#Nodes	Goal Reached		#Success
		Time	#Nodes	
RRT	224	1.61s	23	21
RC-RRT	221	2.53s	32	24
RG-RRT	99	0.62s	4	8
EG-RRT	113	1.19s	7	13
Reduced Bounds				
RRT	204	2.75s	45	40
RC-RRT	199	3.55s	45	32
RG-RRT	92	4.62s	25	28
EG-RRT	107	6.33s	38	43

Table 12.2: Results for Test 2; timeout at 20 seconds (50 runs)

example, bounded model-checking). I also hope to incorporate more RRT variants including RRT+ [133] and A*-guided RRT, and to characterize automatically how well the RRT search trees cover the configuration space. Wrapping these search algorithms in useful interfaces for design support is also important future work. Finally, I believe I could improve the coverage of this technique by using random forests instead of individual random trees to address differences in starting position or game state.

Outside of direct search, my undergraduate Nikolai Chen has prototyped a *compositional* task planner for HyPED games which are broken up into discrete non-interacting spaces. Taking advantage of three special cases—games where spaces besides the player's current location are static *and* where goals can be achieved mainly through collisions *and* where movement is roughly on a 4-connected graph—Nikolai's work finds abstract partial solutions for each room under varying assumptions and then uses these as macro actions for a high-level planner (and, in the future, to guide a low-level motion planner) to solve the whole game. In domains

like Zelda dungeons, these tools can summarize individual room-to-room transitions as sets of linear implications, where the dungeon as a whole is solvable if there is a formula from an initial environment to a solved state. This is only possible because HyPED provides a uniform format for describing game characters and their discrete states and connects this usefully to their real-time behavior. This utility comes directly from HyPED's grounding in operational logics.

Having seen Reductionist and now HyPED, the reader can surely imagine other game modeling and support tools starting from other combinations of operational logics. In the following chapters, I will shift my focus from game modeling to the automatic extraction of game design knowledge from observation of the game's behavior in play. Whether in solving for properties of a game specified by hand or uncovering facts about a game's design, operational logics are an extremely useful foundation for knowledge representation. These applications are dual to each other but, importantly, they can share many of the same underlying knowledge representations—as I will illustrate in Chapter 15, where the same formalism used by HyPED is the basis of a project in summarizing game character behaviors as hybrid automata.

Chapter 13

Automated Game Design Learning

While general game playing is an active field of research, the learning of game design has tended to be either a secondary goal of such research or it has been solely the domain of humans. I recently proposed a new area of research, Automated Game Design Learning (AGDL), with the direct purpose of learning game designs directly through interaction with games in the mode that most people experience games: via play. In this chapter I detail some existing work that touches the edges of this field, describe current successful projects in AGDL and the theoretical foundations that enable them, point to promising applications enabled by AGDL, and discuss next steps for this exciting area of study. Working from operational logics has been key to unlocking AGDL as a productive research area, and the following two chapters illustrate how specific combinations of operational logics can yield suitable knowledge representations for AGDL systems. [1]

The key moves of AGDL are to use game programs as the ultimate source of truth

[1] *Portions of this chapter originally appeared as "Automated Game Design Learning" [196].*

about their own design, and to make these design properties available to other systems and avenues of inquiry. The division here between the design and the program is not meant to imply that the *true* game is fully-formed in a designer's head before *translation* to code; rather, that game makers, being human and not machines, necessarily have a different and more abstracted understanding of their code (or else software bugs and surprising emergent behavior would not exist). Fortunately, many design properties of interest can be tested by looking only at the abstract design, regardless of the low-level implementation.

This contrast between game designs and game programs is similar to the difference between the specification of a program and its concrete implementation. In both cases, the design is often provided at varying levels of abstraction and completeness, admitting a broad class of valid implementations. Over time, the implementation becomes the authority on how things *do* work (difficult though it might be to discern), and how things *ought* to work is informal and often conflicting knowledge held by designers, programmers, managers, and users. The management of design artifacts is its own field outside the scope of the proposed dissertation (see [271] for a survey); this attests to the difficulty of producing and maintaining designs that are correct and programs that faithfully implement them.

While a program is in some sense an extremely concrete design, the degree of specificity and the amount of incidental complexity can conceal the latent specification. In games, there have been many attempts to extract game designs from game programs. Raph Koster even posits that the *fun* of games lies in the pleasure of (manually) reverse-engineering their rules and the consequences of those rules [137]. Recognizing a game design in a game program is necessary to promulgate the craft knowledge of game-making through careful study. By the same

token, computer programs that aim to learn, analyze, imitate, alter, or improve game programs must implement techniques for retrieving elements of a game's design from its implementation at a source code or binary level.

Manual interpretation of game designs' craft qualities is a topic of interest among game designers. The most common suggestions have been for shared vocabularies [75, 56]: descriptions of games' rules, dynamics, and other formal characteristics (including normative statements about the same) in a textual or sometimes graphical notation. The 400 Project [96], game design patterns catalogues [41], and the Game Ontology Project [274] all inherit this approach. These are good at identifying static structures in games, but generally bracket off the experiential arcs of individual players. The open-ended enumerative style of these projects—*bags* of *game stuff*—contrasts with the compositional quality of operational logics.

Game grammars, on the other hand, tend to emphasize this modularity along with changes in the player over time (rather than the evolution of the game state). Koster's grammar foregrounds the composition of games out of smaller subgames, with the player's skill and interest providing the engine for breaking down and building up complex units of play [136]; D. Cook's *skill atoms* [65] and *loops and arcs* [66] serve a similar analytic purpose at differing levels of abstraction. Bura's Emotion Engineering is unique in explicitly predicting changes in players' affective state based on changes in game state [49]. Compared to operational logics, these approaches are somewhat orthogonal and higher-order: while OLs explain how a player infers causal relationships and gives semantics and valence to observed phenomena, these grammars describe learning and experiencing play in the large.

Operational logics have, however, been applied to extremely high-level interpretations

of games: Treanor et al build arguments for sophisticated interpretations of games by tying operational logics along with cultural knowledge to semiotics [259]. The technique has connections to deep reading and focuses on the construction and comparison of arguments rather than the identification of intrinsic truths about games (this often leads to entertaining discussions without correct answers, which is why I think it is a good choice for AI to target). Game grammars are sandwiched in between these layers of "how does the game work?" and "what does the game mean?"

Several attempts have been made at the automatic interpretation of game code and the visible phenomena of games. Some systems only interpret and recover fixed design elements such as level layouts [110] or rules (unpublished work by Matthew Guzdial), while others attempt to transfer design knowledge across games as a general artificial intelligence project. M. Cook's Mechanic Miner works at the level of source code, looking for authorial affordances with which to generate new mechanics; this naive model of game mechanics is surprisingly effective in identifying interesting sites for intervention [69]. General game playing agents such as FluxPlayer [215], ClunePlayer [62], and Kuhlmann *et al's* player [142] perform static analysis of GDL game rules to devise search heuristics. The Arcade Learning Environment [36] poses a challenge and some benchmarks for learning how to play Atari 2600 games (a well-known platform thoroughly explained with respect to its hard-wired operational logics); its bibliography also points out useful prior work in inferring game rules from game phenomena. Machine learning techniques have previously been applied to discover transferable skills which generalize across a whole game genre, and I believe there are opportunities for crossover with that work [115, 135].

13.1 General Game Playing

Automated game design learning is explicitly *not* General Game Playing (GGP). AGDL is not about achieving optimal play, and only care about rewards to the extent that they reveal truths about the design. Of course, AGDL borrows from GGP and General Video Game-AI (GVG-AI), and insofar as learning the design of the game can help produce a more intelligent general player the contributions of AGDL can feed into further GGP and GVG-AI research.

The two main areas of inquiry which GGP has begun to explore and which AGDL generalizes and could support more broadly are heuristic learning and transfer learning. Generally speaking, heuristic learning is a way to learn, on a game-by-game basis, about intermediate goals of the game or rough strategies for guiding search. The GGP agent FluxPlayer made its name by statically analyzing game rules to automatically determine a position evaluation function to be used as a heuristic [215]; i.e., from just the rules it could make an estimate as to how close or far from the goal a particular game state was. FluxPlayer also established which aspects of the rules defined high-level structures like ordered relations, quantities of game resources, board positions, and so on, feeding those into the distance function used for heuristic calculation.

In general videogame playing too, heuristic methods have become quite successful [206, 205]. Some agents match heuristics from a fixed portfolio against the game they are playing, and several agents try to determine whether each non-player character in the game is dangerous or desirable. These can be seen as essentially both GVG-AI and AGDL agents, in that they are trying to learn about the game's design (AGDL) in order to play it effectively

(GVG-AI).

Apart from learning heuristics for a single game, transfer learning is a key area where the portable design representations learned by AGDL could be of use to GGP agents. Transfer is a key aspect of human learning, and indeed human game playing relies heavily on literacies obtained by playing other games in the past. Banerjee and Stone extracted high-level features from the value functions learned on one GGP game to accelerate learning in another [27] . Outside of the GGP context, Könik *et al.* learn "tasks" suitable for transfer within the current environment (if the same problem comes up again later on) or to other environments with similar tasks by biasing value functions when similarities are found [135]. I believe AGDL could support this latter type of learning, giving a way to bridge high-level design concepts across games and to learn tactics and high-level actions within a given game. One promising approach due to Braylan and Miikkulainen [46] is to leverage design concepts like *game characters* to modularize reinforcement learning models, in the process admitting transfer learning by analogizing characters across games.

Another interesting thread is the general play of essentially opaque commercial games, as in for example the Arcade Learning Environment [35]. While learning directly from visual features is important research, here I focus on a particular pair of projects: Learnfun and Playfun [176]. Learnfun observes human play of a given NES game to learn both macro-actions representative of realistic inputs for that game *and* a lexical order over RAM locations, with the intuition that success in many games can be characterized by a number or a set of numbers increasing. For example, progress in *Super Mario Bros.* is measured (roughly) by observing which world and level the player is in, and then by horizontal position within the level. Its

counterpart *Playfun* uses those learned macro-actions in a heuristic search process attempting to optimize that lexical order. Of course there are games for which this process works very poorly, and it will tend to react to an impending loss by pausing the game and halting, but it is surprisingly effective at surfacing useful features.

13.2 Manually Reverse-Engineering Games

Reverse-engineering design properties from games is challenging even for humans, but people are highly motivated to do this. Normally, a user has nothing more than a video of play—or, if they're lucky, a ROM—and wants to get at games' audiovisual assets or other features. This can be motivated by a desire to remix the game's designed elements in various ways; to learn subtle aspects of the game's rules; to cheat at the game; or to modify the game itself, e.g., to translate it into another language or change its levels, characters, or even rules.

The most popular activities around the extraction of game design elements from game programs are likely the manual processes of *ripping* sprites, tilesets, 3D models, and other audiovisual assets from games' code and included resources. Using screenshots and image editing programs, enthusiasts laboriously copy and paste game characters' animation frames into composite images [1] or stitch together screenshots into full level maps [4]. This latter activity has been attacked with greater rigor in the form of the Video Game Level Corpus [240], which establishes standardized level formats.

Game players, especially competitive players, are also interested in learning about a game's rules. Textual walkthroughs can be seen as a high-level summary of a game design,

and the resources accumulated by fighting game enthusiasts [100, 249] are essentially reverse-engineered specifications of the game design suitable for study and practice.

The community of game *speed-runners* have a special interest in the connection between the high-level game design and the low-level rules that enact it. Here, the question of interest is essentially a classical combinatorial optimization problem: is there a sequence of button inputs of length L such that the objective z (for example, the length of L or the number of distinct button presses) is minimized? A sub-community of *tool-assisted* speed runners explicitly write such sequences of inputs one by one in specialized text editors, and both need and develop extremely deep knowledge about games' true, source code-level design features. Recently, a YouTube video describing how details of modulo arithmetic enable the completion of a *Super Mario 64* level with "0.5 A-button presses" obtained millions of views [201]. The accompanying image gallery shows the process of discovering this sequence of moves and the tools and techniques built to support it [202].

The FCEUX emulator for Famicom and NES has a variety of tools to support this community, including memory comparison interfaces, debuggers, and Lua integration to drive the emulator's main loop for experimentation [2]. Enthusiasts of individual games also produce purpose-built tools for viewing and modifying those games' data structures in design-relevant ways; for example, to change their levels, their characters' statistics and appearances, or even plot events and script sequencing [3]. This all depends on knowing how these design elements are realized in the underlying game code.

13.3 Automated Game Reverse Engineering

Because the manual processes outlined above are so game-specific, on top of being tedious and labor-intensive, there have been some efforts to directly analyze games automatically to extract design-relevant information. Chapters 5 and 8 illustrated the approaches taken in the tradition of proceduralist readings: given a specification of a game, applying a series of logic rules suffices to deduce the readings that might be present in a game. In the context of AGDL, one might view these techniques as building up low-level observations about a game (the player controls sprite *A*) into mid-level inferences (the player will attempt to make sprite *A* collide with sprite *B* because that will increase the resource that prevents them from losing) and high-level interpretations (the game is futile since the difficulty increases monotonically). This can be seen as an alternative application of approaches to automatically finding heuristic functions.

There are also techniques that work directly on black-box game programs which inspired the AGDL project. Murphy's *glEnd() of Zelda* project aimed to automatically present a first-person 3D view of 2D NES games, without modifying the games [177]. This requires knowledge of a game's design: whether the camera is top-down or side-view, whether gravity exists, which sprite on the screen is the player, and so on.

glEnd() explored two especially interesting ideas: First, to look at the NES's graphics processor's memory rather than the output pixels to get certain high level features for free; and second, to perform guided experiments that prove properties of interest and find parameters of the game's design. For example, to determine which sprite is controlled by the player an

algorithm might compare how much the position of each sprite on screen changes after holding left, holding right, or standing still from the same start state. To tie locations in system RAM to sprites' on-screen positions, it could find all locations which have the same integer value as that position and see if modifying each such address once and then waiting a frame causes the sprite to draw in a different place (here, the intuition is that one RAM location determines the others). It could determine whether gravity applies to a character by placing it in the sky (by modifying RAM) and then resuming play without providing any other inputs; if the character falls, then gravity applies to it. It could figure out whether a tile blocks movement by putting the player next to it and attempting to move the character. Apart from the existence proof that recovering game design properties from game binaries is possible, *glEnd() of Zelda* provides an important insight: even if some game situations are impossible (e.g., placing the character in an unreachable part of the screen), they might still be *informative.* Murphy's work is effective for the automatic 3D-ification of NES games, but how could it be generalized to other kinds of properties and made more robust?

My collaborators' earlier work attempted to learn "latent causal affordances" of game entities via observation of collisions and the potential effects of those collisions [241]. They applied a variety of machine learning techniques to cluster entities based on their observed mechanical properties, learning things like "these objects all occlude Mario's motion" which one could summarize as being "solid."

Guzdial and Riedl learned implicit rules of *Super Mario Bros.* level design by observing videos of gameplay [110]. Machine learning techniques were used to build probabilistic models which captured level design knowledge like *treetop tiles should be above a rectangle*

of tree trunk tiles. With the same video source as input, Summerville *et al.* [239] leveraged player path information to bias the generation of a level towards the specific play style of a given player. Using recurrent neural networks they learned level design and player paths simultaneously, allowing the generator to implicitly bias the generation towards actions it saw (e.g., if a player interacted with question mark blocks it generated more of those blocks to make the generated path more likely to interact with them).

In the general game playing domain, techniques for learning game rules from observations of game behavior have been explored in recent years. Björnsson's Simple Game Rule Learner [42] derives the rules for a restricted class of Game Description Language (GDL) games by learning finite-automaton models for each of the game's pieces. Gregory *et al.* extend this work by incorporating techniques from the planning domain model acquisition literature [107], learning interactions between pieces and admitting the dynamic addition and removal of pieces.

13.4 Software Specification Mining

The problem of recovering a game design from a game program is a special case of recovering a specification from a piece of software, and the problem of automatically extracting a specification from a software system is of general interest. Variously called specification mining [80], design recovery [40], or just reverse engineering [54], the general schema is the same: looking at a program's source code or runtime behavior, synthesize an abstracted model of the software suitable for analysis.

Specifications are useful in that they provide a reference against which to compare im-

plementation behavior; on the other hand, specifications must be manually inspected to verify that the behavior they describe is desired by the software's creators. Recovering a specification from working software is a good compromise to admit the use of verification techniques that only work at the level of specifications, to help prevent future regressions, and to see the software's latent specification in simpler terms than source code.

Like games, many user-facing applications are developed in an iterative process. The specifications and even the requirements behind them change over time as designers learn more about the problem domain. In this sense, iterative design leads to just-in-time specifications, which are often only encoded in the program itself. User-testing (or, in games, playtesting) is deployed to ensure both that the design worksand that the software conforms to the design, but the latter condition could be verified automatically if the specification were prepared in advance.

The particular area of specification mining overlapping with current explorations in game understanding seems to be *concept assignment,* where *domain-level* concepts like (in the case of games) avatars, enemies, and points are mapped onto source-code modules or statements. This is an active area of research [104] with interesting overlaps in program slicing [113, 103] and feature location [88]. Two techniques of special interest for games are the clustering and segmentation of execution traces [20, 171] and user interface reverse-/re-engineering [28], especially black-box approaches [18] which have similarities to game rule learning.

13.5 Automated Game Design Learning

Specification recovery is essentially unsolvable due to the halting problem and to the unbounded space of possible specifications that a given program refines. Why do I suppose that progress can be made in this area in the domain of games? Games differ from arbitrary software in two key ways. First, games must be legible to players (and indeed their own designers): it must be possible for players to form reasonably accurate models of a game's behavior, or the game is unplayable. Second, there are diverse and mature theories of game design to inspire and organize knowledge representation.

Researchers have long known that picking the correct problem specification and knowledge representation is vital for a system's success [7]. A successful game design learning system (as distinct from a generic specification recovery tool) must be parameterized by a way of organizing game design elements specifically. Just as players and designers form models of the games they play, so must AI systems.

The first half of this dissertation explained and argued for the use of operational logics in accounting for the behavior of games and players' perception of that behavior. Applying operational logics to the problem of specification recovery in games is a key step forward for game design learning, as prior work in this area mainly considered games as bags of parameterized mechanics [278, 107]. Such approaches can be effective, but they are not connected to player experience and, worse, they assume mechanics are the fundamental building blocks whereas most mechanics are built out of several smaller pieces (e.g., collision detection, resource transactions, and other constraints and effects).

Over the last year or two, my collaborator Adam Summerville and I have made significant advances in the automatic extraction of domain-level (i.e., game-design-level) specifications from observations of the behavior of game programs.

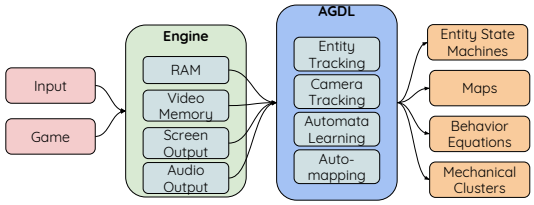

Figure 13.1: Pipeline for AGDL. Player input can come from sources of opportunity (e.g., speed runs), AI players, or human playtesting. Different game engines will expose different sets of observations which feed into the learning process (some of which are listed). Various learning modules (Blue sub-components) can be designed to produce relevant corresponding outputs (Orange). This list is not exhaustive, but shows some of the features, techniques, and outputs I have successfully used in recent projects.

Games are full of complex and emergent behaviors, and their runtime behavior is hard to predict from their source code or binaries. The game code *is* the design in a final sense, but neither the designer nor the player have direct access to this design. In general, source code is specified at too fine a grain (frame-by-frame transition rules) and in a way which is hard for humans to work with. Instead, game creators (programmers and designers) and players each have a model of the game with which the code may or may not agree. Both groups learn more about the game as they play it. Even an initial brush against a game reveals an abstract specification concerning high-level design elements: characters' appearances, animations, and general behaviors; level layout and connections; a game's goals; and so on.

I see several key application areas for AGDL:

Improving Human Play — E.g., seeing a fighting game character's state machine, collision

frames, and so on is useful for players.

Improving General Game Play – E.g., learning to construct useful abstractions to guide planning.

Quantitative Game Studies — E.g., comparing mechanics across games or pulling out game level data for architectural study.

Data-Driven Procedural Content Generation — E.g., learning the mechanical properties of the tiles and entities in a game for corpus generation purposes.

Code Studies — E.g., tracing high level design rules down through runtime behavior to understand source (or machine) code.

Game Design Verification — E.g., extracting formal models of game rules and ensuring that certain design properties hold on the model.

How can game design be extracted from a game program? Often, human designers do it by reading code and thinking, but reading code in any way besides just executing it is in general hard for computers (and many very different programs could be behaviorally identical). I have therefore focused on the way that most people learn a game's design: *play.* Players and designers (playtesting being a crucial activity) both interact with and learn a game's rules through playing, experimenting, and thinking—this is the mechanism I propose automating.

The goal is to extract high level design properties like level structure, character behaviors, progression, resource exchanges, and so on from observations of human or AI play in as automatic a manner as possible. This is an open-ended problem and, at the fully general level,

it is surely undecidable; nonetheless, I discuss ways I have found to make the problem tractable in several interesting cases.

My starting point, as in the rest of this dissertation, is a knowledge representation founded in Operational Logics (OLs). Again, I am not treating games as bags of mechanics—e.g., I do not operate at the level of saying "this game is a platformer" or even "this type of collision (always) triggers that type of reaction." Instead, my systems learn in terms of individual OLs' abstract operations—the pieces from which mechanics are made. The communicative strategies provided by operational logics give a clear way to connect what a game *looks like* like with how a game *works.*

For the research presented here, I targeted games on the Nintendo Entertainment System (NES) for several reasons:

- It hosts a diverse set of games
- The architecture offers a clean way to identify high level image features easily
- It has an active community providing a library of playthroughs of these games
- Efficient software emulators offer fine-grained control and capture of audiovisual and other state

13.5.1 Observations

The second question to be asked of any computational-intelligence system after "How does it understand the world?" must be "How does it observe the world?" In the most constrained case, a system can only observe video of games being played; some of the works cited

above have obtained good results from only this data source [110, 239]. If a system also has access to player inputs (e.g., a timed sequence of button presses), it can attempt causal reasoning, blaming changes in character behavior on player actions [244]. Sometimes the system can observe the game's internal runtime behavior including its memory address space or, for games run in emulation, the states of memory and registers of the emulated hardware [243].

A computational-intelligence system might also control the platform on which the game runs, which allows it to manipulate memory values or drive new input sequences on-demand, saving and loading memory states to jump around and perform search. There are also binary analysis tools like S^2 [55] and *angr* [224], which are designed to exercise programs' hard-to-reach behaviors.

The above is about as much as one can hope for given a black-box game program. On the other hand, sometimes source code is available as well; in such cases one can deploy programming-language-specific model checkers and static analysis tools, or augment the approaches above with source-code level knowledge (e.g., blaming observed in-game interactions on specific lines of source code). I am also interested in exploring the extent to which in-game text, game walkthroughs, or instruction manuals can bootstrap learning (as in [44]), as can guessing the valence or semantics of visible game entities based on their appearance. These are important information channels for human players and this kind of cultural or meta-game knowledge can be readily incorporated in the framework of OLs.

13.5.2 Human Guidance

Videogames by nature have extremely broad and deep search trees. Fortunately, in many cases human players record their explorations through games as speedruns (often down to the level of timed input sequences). Moreover, there exist archives of game save files and saved system states, admitting easy bookmarking of interesting regions of a game's possibility space even if the concrete path to get there is not recorded.

A key insight is that a system can use these *savestates* to achieve much better exploration of the possibility space than naive search alone could produce. If it has access to a playthrough (either from a motivated player or from a designer), it could uniformly sample along prefixes of that playthrough (or randomly mutate it) to explore different parts of the game easily; this gives a narrow global path from which it can branch out locally to investigate the nearby area. In the design support context, such a system could use the game creators' previous playthroughs and manual testing as starting points for automated exploration, bootstrapping verification or rule-learning/updating tasks.

13.6 Dynamic Analysis of NES Games

The following chapters explore work to date on dynamically analyzing NES games for design-relevant content. Each of my projects this area has begun by linking an NES emulator runtime, loading up a game program (colloquially, a *Read-Only Memory (ROM)*), executing a series of inputs, and examining the emulated system's state by interrogating the emulator's runtime data structures. Many emulators support checkpoint save and restore operations, taking

a snapshot of the whole system state that can later be restored at any time. This admits the use of search techniques as well as speculative execution along one or many possible futures.

While the NES has a general-purpose CPU and RAM, a key aspect that I have taken advantage of is its dedicated processor for image generation. The Picture Processing Unit (PPU) has its own associated memory holding the data structures necessary for tiled rendering and sprite drawing. This lets AI systems extract visual information without doing extensive image processing or background subtraction. Algorithms can also take advantage of information like color palettes, shared tile indices, and horizontal or vertical flipping to bias rule learning. Furthermore, the separate treatment of tiled graphics and sprites helps easily distinguish between game characters and backgrounds (this is complicated when the perceived characters are built mainly from background tiles, as in certain boss fights in *Mega Man 2*).

There are some significant issues with working from PPU data; the key problem is that what is in the PPU's state at the end of the frame is not necessarily what was in that state when a given pixel was drawn. The bank of sprite data and the hardware scrolling registers can change in the middle of a frame; I have developed an emulator with deeper instrumentation to record these data for each drawn pixel. More importantly, while the NES's software library is large this sort of technique does limit the work to platforms with NES-like hardware affordances (and for which someone has written the appropriate instrumentation code); working directly from the output framebuffer would be ideal.

13.6.1 About the NES

The NES has a general-purpose MOS 6502-derived CPU with its own RAM and dedicated co-processors for audio synthesis and image generation. The latter Picture Processing Unit (PPU) has its own associated memory, with the data structures necessary for tiled rendering and sprite drawing. The technical details of the NES are important for my purposes because they offer richer information than just the visual output of the console. My AGDL systems can precisely observe the sprites on the screen or the composition of a game level without doing extensive image processing or background subtraction. They can also take advantage of information like color palettes, shared tile indices, and horizontal or vertical flipping to bias rule learning.

Sprites on the NES are 8 pixels wide and either 8 or 16 pixels tall. Most game characters are larger than this: Super Mario, for example, is 16 pixels wide and 24 pixels tall. Such characters are made out of several hardware sprites which are moved together to give the illusion of a solid object. While the system has direct access to high-level features like sprite positions and appearance, it must perform sprite fusion to treat these larger characters as atomic. Sprites can also change appearance or dimensions over time: *Metroid's* Samus Aran can sometimes be very tall and sometimes morph into a small ball, and to a player these must be considered the same character. The general technique is to observe the bounding boxes of these hardware sprites over time, and if there are boxes which consistently overlap just along a particular shared edge I presume they comprise the same object (this notion of object coherence comes from *entity-state logics* in operational logic parlance).

Importantly, not all NES game characters are made of sprites, or at least they are not *only* made of sprites. Large enemies that do not animate much are often built from background tiles (as in *Mega Man* and *Dragon Quest*). Objects like movable blocks in *Zelda* or breakable bricks in *Super Mario Bros.* are usually tiles, but temporarily turn into sprites when interacted with so that they can animate smoothly (i.e., not constrained to multiples of 8 pixels). My systems thus far do not yet account for the mostly-static-tile characters, though intuitively they should be treated similarly to regular characters; I believe this is related to concerns like doors, resource pickups, or other dynamic objects that interact with the player but do not move. I also have only recently explored the problem of migrating the identities of game entities between tiles and sprites. Still, the sprite fusion technique outlined above handles moving characters well, and moving characters have been the main object of my rule-learning inquiry.

If a system can detect scrolling of the game screen and knows the contents of the background tilemap, it is clear that it could automatically map game levels, for example to populate a corpus such as the VGLC [240]. I have done recent work in mapping larger networks of rooms, which also incorporates spatial *linking logics*: briefly, the problem is to determine whether the player is leaving or entering a room and in solving it I have applied some heuristics like the temporary loss of real-time control to hint this determination. I explore this problem further in Chapter 14.

Level layouts and character identities are structural properties which humans can often resolve just by looking at still images. This work becomes more interesting when considering learning dynamic design properties like the interactions between characters and the environment or character's dynamic behaviors. *glEnd() of Zelda* shows that simple heuristics

suffice to understand many interactions with background tiles [177]; the rigor provided by the operational logics framework yields more tools to distinguish types of tiles without enumerating in advance all the possible tile types. I explore the problem of learning character behaviors in Chapter 15.

13.7 Automated Game Design Learning in the Future

This vision suggests several new research areas that would have been tedious or infeasible without automation. I also propose a few natural extensions for future work.

The field of game studies relies on extensive play and examination [131, 257]. This demands a large amount of a researcher's time and limits the scope of possible analyses, particularly constraining the number of games that can be investigated at once. Fasterholdt *et al.* [97] studied a number of platformer games to form a general model of jumping, but their (labor-intensive) analysis was limited to four games. My work building on this [243] automatically performed in-game experiments, allowing for the capture and juxtaposition of the jumping dynamics of 48 games, a 12-fold increase over previous work. Large data-based game studies [127, 116] generally rely on textual data such as Wikipedia or GameFAQs. I believe that similar quantitative work extended into the actual dynamics of the games (not just text about the games) is an exciting new direction.

Data-driven Procedural Content Generation (PCG) has become a popular research field in recent years, but a key drawback is the lack of usable, clean data sources. Projects such as the Video Game Level Corpus (VGLC) have begun to remedy this, but even this corpus only

addresses 13 games from 7 different series. While there are large databases of game levels as images [4] or videos (on YouTube), they lack much of the information required to accurately learn game mechanics, most notably control information. Furthermore, while some automated processes of parsing these exist [108], they rely heavily on human input and annotation to be able to understand any of the mechanical properties of the level information. Automatic gathering of level and behavior data from the game, combined with property-directed testing of assets for mechanical properties could allow for a much larger and richer set of data for Data-driven PCG.

Beyond levels, there is a broad variety of content that has yet to be generated in a data-driven manner including entities, mechanics, level progressions, and so on. Due to the aforementioned reliance on computer vision and the lack of models of interaction, these have been impossible—until now.

AGDL's current incarnation is tied to the NES and to found play traces. Generalizing this to other platforms, including the Super NES, PuzzleScript, or even engines like Unity would be straightforward in some ways (the abstraction-learning code of CHARDA and MARIO would be largely unchanged), but it would also yield new challenges and opportunities. Imbuing the design-learning agents with the ability to explore the game's possibility space on their own is also a natural next step: whether this means branching out from different instants of a given play trace or exploring from the beginning of the game, relational-learning agents, heuristic selection of experiments to try (like MARIO or *glEnd() of Zelda*) and curiosity-driven search [106, 203] could help learn a broader variety of game rules more precisely. Finally, addressing more types of operational logics is a clear step forward: resource logics, game-mode

logics, and progression logics are both widespread and valuable to learn, as are the non-spatial linking logics used in game dialog trees.

In this chapter, I have described AGDL, a nascent field of research aiming to automatically learn useful, portable representations of game rules and instantial assets via observation and interaction with the game itself. This is a productive research agenda that has been approached obliquely but rarely engaged directly. AGDL has deep roots criss-crossing game design support, reverse engineering, statistical learning, inductive logic, specification recovery, and general game playing. The next two chapters will each explore a specific AGDL project in detail.

Chapter 14

Mapping NES Games with Mappy

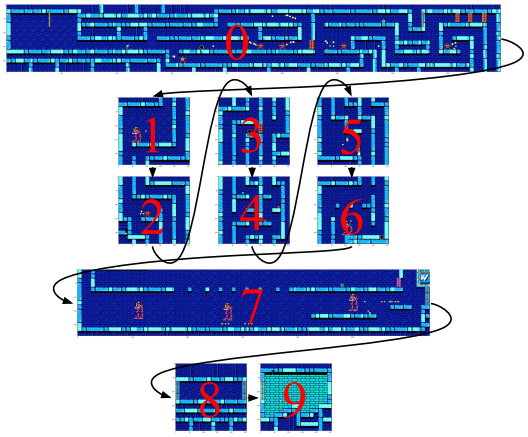

Figure 14.1: Flash Man's stage from *Mega Man 2*. *Mappy* handles rooms and arrangements of arbitrary size.

The production of game maps—both of local, scrolling rooms and the global structure that links those rooms together—is of interest to the general game-playing audience and has applications to general videogame AI and to data-driven procedural content generation. Human players need, create, and make use of maps to improve their play. Players track locations in their heads, draw informal maps, and in some cases capture screenshots and meticulously compose

them into game world atlases [1].

Many games, including action-adventure games like *Metroid* or *The Legend of Zelda,* essentially compose together two distinct games: an action game of dodging and attacking played at sixty frames per second, and an adventure puzzle game in which the game map forms a graph search problem. In this latter game, players engage in activities such as retrieving items from one room to open another room's exit, returning to suspicious locations to see what new opportunities have opened up, and collecting supplies before proceeding to a boss monster. AI game-playing agents that do not remember both high-level map structures and sufficient low-level details of each room cannot hope to formulate plans like these on their own. Clean, complete maps of game worlds (including the ways in which they might change due to player actions) are also integral to research in automatically learning game rules [241] and to data-driven procedural content generation [245].

This chapter describes *Mappy,* a partial solution for automatically mapping certain classes of Nintendo Entertainment System (NES) games (see Fig. 14.1 for an example of a map made by *Mappy*). To apply this approach to other game platforms, software renderers would need to expose information about the tilemap being drawn for the lower-level feature extraction and the data representation to be applicable. I also consider only games without modal menus or other non-spatial uses of tiles. Specifically, I assume that a (given) sub-rectangle of the screen is *scrolling* and all other regions of the screen can be safely ignored.

[1] *Portions of this chapter originally appeared as "Automatic Mapping of NES Games with Mappy" [197].*

14.1 Related Work

While there are several online communities at work producing full maps of games, this process is laborious and largely manual. Ian Albert says of his process for extracting *Super Mario Bros.* maps:

> ...these maps were created using FCEU, a free Nintendo emulator for Windows. I used the ROM image for the *Mario/Duck Hunt* version of the game. Screen captures were tediously pieced together in Photoshop. Some text was reproduced using the *Press Start* font by codeman38, which emulates the same font used in *SMB*. Some Game Genie codes were used to make mapping easier... the maps should not be any different due to the use of these codes [8].

This workflow requires that a human play through the game, fully test the level (discovering any hidden objects), and periodically take screenshots. This is only the beginning of the tedious effort involved: at this point, the screenshots must be checked for accuracy (if the level can be changed by player activity they might fall out of sync with each other), and failures here could force the mapper to re-do sections of play. Next, the screenshots must be stitched together in an external piece of software and any dynamic objects like game characters must be removed. Finally, all of this must be annotated with semantic information including how the rooms are connected via discrete links!

Every step of this process is time-consuming and error-prone. Furthermore, due to the "cheats" being used, there is no guarantee that the map is truly faithful to the original. They seem safe for this example, but that does not mean that a different game with different (but similar) cheats might not subtly alter the map (leading to an inaccurate map). While the map produced by this work supports human interpretation, it requires further human intervention before a machine can effectively process it. This is due to the image-based format, which

requires manual annotation of mechanical properties using an *ad hoc* visual language requiring an understanding of game-specific rules.

For example, Albert uses a visual shorthand of putting a mushroom icon above a block tile to denote that the block in question contains a mushroom. The actual visual depiction is something that would never be seen in the game (if the mushroom had appeared, the block would not still be a question mark or brick tile), but is read easily by humans. Similarly, links between maps (between, say, Level 1-1 and the hidden room found by entering the fourth pipe) are depicted with writing on the image. In both cases, these annotations would have to be written in a machine-readable form to be processed by a computer, or the computer would have to be "taught" how to read the image so as to disambiguate mechanical properties from the spatial map.

Although many games have been mapped, quality and standards vary significantly from mapper to mapper. Some mappers produce just the tile backgrounds, while others include the position of all game characters, while others still show hidden mechanical properties. For some games, there are multiple maps where each describes one such component, but no single map holds all of the relevant information (e.g. the *Link's Awakening* maps available at the website Zelda Elements [94] do not show the contents of treasure chests but do show all characters, while those at VGAtlas [21] show the full treasure chests but no characters).

14.1.1 Map Extraction

Manually-produced maps are unsuitable for some applications—for example, unpopular or hard-to-find games are not likely to have high-quality manually-created maps, and these

maps' purely visual representation can make them difficult for automated systems to process. Extracting maps automatically could circumvent such issues; this is an area of interest for game fan communities as well.

Enthusiasts who want to *modify* a game's maps and other internal data must first be able to identify and extract the game's built-in maps. This requires detailed, game-specific information about how the map data are stored and encoded. For games that are routinely modified in such a way, the map formats become a kind of common knowledge; these communities even produce polished software tools to automate the process of viewing and editing maps. The main games addressed in the present work (*Super Mario Bros., The Legend of Zelda,* and *Metroid*) all have well-understood map formats reverse-engineered from examining source code and memory locations at runtime.

Some games—for example, *Doom* and its successors—define their levels in standalone data files (sometimes wrapped up in larger archives, as in *Doom's* WAD format). *Doom's* active fan community developed tools to extract these maps and, later, re-pack them to replace the original game's levels. On the other hand, *Metroid* and *Super Metroid's* levels are only partially defined by bytes of data; the remainder are produced at runtime by an algorithm which is somewhere between decompression and procedural content generation [112], and these levels can only really be seen accurately by dynamic analysis: watching the game being played over time (or, equivalently, simulating its code to produce the output levels).

This obviously complicates the automatic extraction of maps, but it can still be done on a game-by-game basis with extensive effort. Static analysis can produce superior maps for those sets of map features where the program's use of the data is well-understood, and once

it works for a given game it can work for all games that use the same internal data formats (including modifications of the game). It can also support games that generate their levels procedurally, since the generation algorithm can be reverse-engineered and fed with different seeds to enumerate possible maps. Unfortunately, this requires a lot of deep knowledge both of the game's platform and each individual game's machine code.

Some of these limitations can be overcome if the game's runtime memory format is well-understood. Extant map extraction schemes for *Dwarf Fortress* based on the *dfhack* tool analyze the game's memory structures to pull out complete maps; this hybridizes static and dynamic analysis. Conversely, dynamic map analysis can be done just from video as in work by Guzdial [109], but it is hard to learn linking structure without control information (see Sec. 14.3). Guzdial's work associates video frames together into "chunks", but it is difficult to know exactly the relationship between these chunks and the levels that they are derived from, as their work finds approximately 47 chunks per level. Their work also relies upon human annotated spritesheets with game-specific prior knowledge to correctly identify tiles and sprites.

Many approaches to automatic game playing result in the construction of internal maps based on the agent's sensory data. Rog-o-matic [167] constructed three separate maps (an object map, a terrain map, and a room-cycle loop map) which it reasoned over while playing *Rogue*. Golovin [138] is an interactive fiction playing agent that constructs a map of the world as it travels. While their game world is depicted as text instead of a graphical representation, it shares some of the same challenges encountered here—namely that a location's depiction might change over time and that multiple locations might share the same depiction. Note that for these and other game playing agents, map-building is merely an intermediate by-product and not the

system's intended output. Furthermore, these (often special-purpose) approaches typically only require a map to be good enough to guide play, not to be a definitive artifact usable for other purposes.

To summarize the above concerns: automatic map extraction from game data files requires laborious case-by-case reverse engineering. Doing the same for games that use procedural content generation additionally requires understanding either or both of the game's code and its runtime memory storage formats. *Mappy* does not require any of this reverse engineering effort, at the same time avoiding the problems of purely video-based techniques by having some knowledge about game *platforms* (in this case the NES), as opposed to specific games.

14.1.2 VGLC

Because gathering game level data has so many complexities, Summerville *et al.* assembled the *Video Game Level Corpus* (VGLC) [240]. The VGLC, as of publication time, archives and adapts levels from 12 games into three different map formats.

Some of the highest-quality maps in the corpus were assembled from static analysis (the WAD files for *Doom* and *Doom 2*). Unfortunately, this approach cannot extend the VGLC very quickly because static level extraction tools are game-specific and difficult to produce.

Half of the games in the overall corpus were added completely by hand-annotation. The remaining four games' maps were obtained by a mixture of human and computer annotation. Specifically, Summerville *et al.* used template matching to combine a picture of a game map (assembled manually as above) with a human-annotated set of tile types to derive a complete semantic tilemap.

The VGLC proposed three file formats for standardization, of which *Mappy* could be used to generatexs the tile- and graph-based formats. The tile-based format represents levels as a $W \times H$ matrix, where W is the width and H is the height of the level. Each element of the matrix is represented as an UTF-8 character. Along with each level file, there is a `legend` JSON file that denotes the semantic meaning of each character (e.g. `-` is empty and `X` is solid). The graph based format adapts the DOT graph description language to represent rooms (nodes) and doors (edges) between them. *Mappy* targets the tile-based format for individual rooms and tracks the linkages between rooms using the graph-based format.

14.2 Mappy

Mappy is designed to work on games where an *avatar* moves around a large *world* broken up into smaller *rooms*. This covers significant aspects of a broad class of games including platformers, action-adventure games, and role-playing games. I based this view of the world on these games' usual composition of four operational logics: collision logics, which describe spaces made up of distinct objects which can touch each other and possibly block each other's movement; linking logics, which define larger conceptual spaces including connected rooms and the transit between them; camera logics, which account for the fact that the visible part of the world is a window onto a larger contiguous world; and control logics, which map e.g. button inputs to in-game actions. As in the preceding chapters, selecting a specific composition of logics immediately yields a knowledge representation.

In its current form, *Mappy* takes as input a playthrough of a game and the game

program, then runs an NES emulator on that program and observes the system's state over time. *Mappy* watches a portion of the screen for changes; this screen rectangle is currently given in advance, but it could be determined automatically in the future. At each timestep, *Mappy* determines what tiles are visible on the screen, whether the screen is scrolling and if so by how much, and whether the player currently has control over the game (through speculative execution of inputs). *Mappy* accumulates a map of the current room as the game is played: when *Mappy* sees a new part of a room, it adds those tiles to that room's map. If a tile in a room changes, *Mappy* also notes that the tile has changed, storing a history of each coordinate's contents over time. This is important for capturing e.g., breakable blocks in *Super Mario Bros.* or collapsing bridges in *Zelda 2.* *Mappy* also watches for cases when the player might be moving between rooms and starts on a new map when the move is complete. Finally, *Mappy* analyzes the rooms it has seen and suggests cases where two witnessed rooms might actually be the same room so that a human may choose whether to merge them together.

Mappy works on NES games because that platform's hardware explicitly defines and supports the rendering of grid-aligned tiled maps (drawn at an offset by hardware scrolling features) and pixel-positioned sprites. The NES implements this with a separate graphics processor (the *Picture Processing Unit* or PPU) that has its own dedicated memory defining tilemaps, sprite positions (and other data), color palettes, and the 8×8 patterns which are eventually rasterized on-screen. During emulation, *Mappy* can directly read the PPU memory to access all these different types of data.

Static geometry, including background and foreground tiles, is not built of sprites but is instead defined in the *nametables,* four rectangular 32×30 grids of tile indices; these four

nametables are themselves conceptually laid out in a square. Since the PPU only has enough RAM for two nametables, individual games define ways to mirror the two real nametables onto the four virtual nametables (some even provide extra RAM to populate all four nametables with distinct tiles). On each frame, one nametable is selected as a reference point; when a tile to be drawn is outside of this nametable (due to scrolling) the addressing wraps around to the appropriate adjacent nametable. Note that many game levels are much wider than 64 tiles—the game map as a whole never exists in its player-visible form in memory, but is decompressed on the fly and loaded in slices into the off-screen parts of the nametables as the player moves around the stage.

Mappy remembers all the tiles that are drawn on the visible part of the screen, filling out a larger map with the observed tiles and updating that map as the tiles change. A *Mappy* map at this stage is a dictionary whose keys are a tuple of spatial coordinates (with the origin initially placed at the top-left of the first screen of the level) and the time points at which those coordinates were observed, and whose values are *tile keys*. A tile key combines the internal index used by the game to reference the tile with the specific palette and other data necessary to render it properly (from the attribute table and other regions of NES memory). After *Mappy* has determined that the player has left the room (see Sec. 14.3), the map is offset so that the top-left corner of its bounding rectangle is the origin and all coordinates within the map are positive; this is rasterized and output as an image. *Mappy* thereby constructs the level as it is seen from the perspective of (tile-based) collision logics: the (mostly) static geometry and its (semantically significant) visual appearance over time.

Mappy learns the full history of every tile, rather than committing to its initial or

final appearance, for four main reasons. First, during scrolling, stale tiles are regularly replaced with fresh ones, and in some games this can even happen at the edges of the screen causing visible glitching. Second, the screen often fades into or out of rooms (or performs some other animation), and just taking the first- or last- seen tile could lead to unusable maps. Third, many tiles animate during play (for example, ocean background tiles or glittering treasures). Finally, the player can interact with many tiles: switches can be flipped, blocks can be broken, walls can be bombed, and so on. So *Mappy* must store all the versions of a tile to admit applications like learning tile animations or interactions. For rasterization and visualization, *Mappy* generally picks the tile's appearance 25% of the way into its observed lifespan, but this is an arbitrary choice and the generated maps are mainly for human viewing. A more principled choice might be to take the most common form the tile took during its lifespan.

While in general the nametables are used for terrain and the hardware sprites are used for game characters, there are some exceptions like large enemies or movable blocks which I outlined in the previous chapter. *Mappy* does not account for these special cases yet.

Because some important level objects are sprites and not tiles, *Mappy* also learns the initial placements of dynamic game objects in the larger map. *Mappy* identifies abstract game objects by observing hardware sprites over time using the sprite tracker described in [242]. This system uses information-theoretic measures to merge adjacent hardware sprites into larger game objects and maintains object identity across time using maximum-weight matchings of bipartite graphs (object identity and positioning in 2D space are natural conclusions to draw from collision logics). *Mappy* takes the first-witnessed position of each object, registers those coordinates relative to level scrolling (explained in the next section), and renders its constituent

sprites into the level maps to capture, for example, that a mushroom pops out of a particular question-mark block.

14.2.1 Scrolling via Camera Logics

Although the PPU features hardware scrolling, and (some) of this information persists in the PPU's hardware registers, capturing screen scrolling information is surprisingly subtle. Games can alter the hardware scrolling registers essentially at any time during rendering, to achieve, for example, split screens or static menus over scrolling levels (the NES does not support layered rendering, unlike the Super NES). *Super Mario Bros.* and its sequels draw the top part of the screen containing status and score information without scrolling, and then turn scrolling on for subsequent scanlines. *Super Mario Bros. 3* puts status information on the *bottom* of the screen as well, so only a small window of the larger screen scrolls. These are concrete examples of camera logics, where a portion of the screen is dedicated to a viewport backed by the illusion of a moving camera. As mentioned above, the system registers the visible part of the level in a larger tilemap, under the assumption that a rectangular viewport will view a rectangular region of a potentially larger space.

While *Mappy* could have obtained pixel-precise scrolling information by instrumenting the emulator to trace when hardware scrolling state changes, my collaborator and I wanted to see how far the system could go without such interventions to remain as general as possible. *Mappy* deploys two techniques, each with their own strengths and weaknesses: a perceptual algorithm based on registering each frame's visual output with the previous frame's and a hybrid approach which registers only the current frame's visual output (converted to grayscale) with

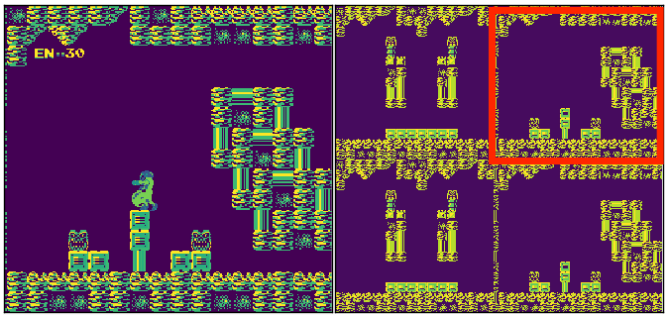

Figure 14.2: Visible screen registered with PPU nametables. Note vertical mirroring and horizontal wrapping.

the PPU's four nametables to determine which rectangular sub-region of the larger tilemap is being shown (see Fig. 14.2). The former technique can break down with animated backgrounds (for example, waterfalls), while the latter will fail if the perceived scrolling is done mainly by sprites rather than background tiles, as in certain boss fights in *Mega Man 2*—this would also be an issue if the system tracked hardware scrolling with the instrumentation described above. In either case, once *Mappy* has precise scrolling information it can convert coordinate spaces from the subset of tiles drawn on the screen into the frame of reference of the larger map it is assembling.

14.3 Linked Rooms via Linking Logics

The goal of this work is to learn not only one large tilemap, but the graph structure by which smaller rooms are linked together (game worlds are not in general planar or even Euclidean). To do this, the system must determine when the player leaves one contiguous space for another. *Mappy* considers two main ways in which linking logics communicate room changes to players:

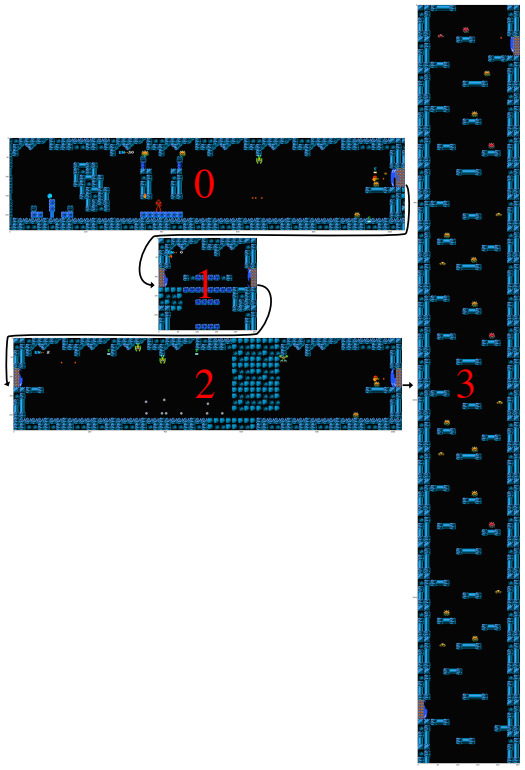

Figure 14.3: The first four rooms from *Metroid*. Note that the playthrough only observed a small portion of room 1, which is actually another tall vertical corridor.

- Smoothly scrolling between connected rooms
- Teleporting between rooms

The first type of transition is the most common type in *The Legend of Zelda* and *Metroid* (see Fig. 14.3). In these games, when traversing between most rooms the player loses control for a period of time while the screen rapidly but smoothly scrolls completely into the new room. After the player regains control, they are in a new room. To test for this type of transition *Mappy* must know for each frame whether the screen is scrolling and whether the player has control; it already knows about scrolling, so it uses the savestate features of the emulator to determine whether the player has control.

The central question of player control is: "Would the world have been different if the player had done something else?" Because the system knows the full input sequence it can look a few moves ahead to see how the world will evolve according to the playthrough; it automatically takes a screenshot of that state for reference. Next, it simulates seven possible futures (one for each button besides "start") three frames ahead and compares a screenshot taken in each of those eventualities against the reference state. If these actions produce different outcomes than the reference, then the player must have had control at the initial frame.

In many games, some animations enacted by the player implicitly remove player control for some period of time (e.g., the fixed length jumps in *Castlevania*), so there is a configurable parameter for how long control must be taken away before counting as a complete loss of control. Since most room transitions take at least one or two seconds, and most in-game animations remove control for less time than that, this allows for a clean separation of the two

causes for losing control. Of course, it is conceivable that the player does not have control but is not entering a new room, so *Mappy* assumes that the screen must also be scrolling while control is lost (and that it must have scrolled by at least half the scroll window width/height). This accounts for freeze frame animations such as when Mario acquires a mushroom and grows or the fanfare that plays when Samus acquires a new item, which show a loss of control but the screen stays stationary.

The second category of spatial transition above places the player in a new room that has little or no visual relation to the previous room, perhaps from descending a staircase or going down a pipe. *Mappy* treats these by looking at the overall appearance of the game screen, and if it changes too drastically within a short timeframe it assumes that the player has probably teleported to a new room. This is complicated by game levels that incorporate drastic sudden changes to the visible portion of the tilemap (such as the "dark storm" level in *Ninja Gaiden* or Bright Man's stage in *MegaMan 4*), which yield false positives where *Mappy* thinks that it has gone to a different room. Given the optional room merging discussed below (and the possibility of stronger heuristics which I leave for future work), this is not a fatal flaw.

14.3.1 Cleaning Up

At this point, *Mappy* has detected individual rooms and linkages between them, but it assumes that every link leads to a brand new room. In most games, at least some links are two-way or converge on the same destinations—most game worlds form a graph and not just a tree. It could simply merge rooms that look identical to each other, but there are numerous cases where this might fail. For instance, there are rooms in *The Legend of Zelda* that

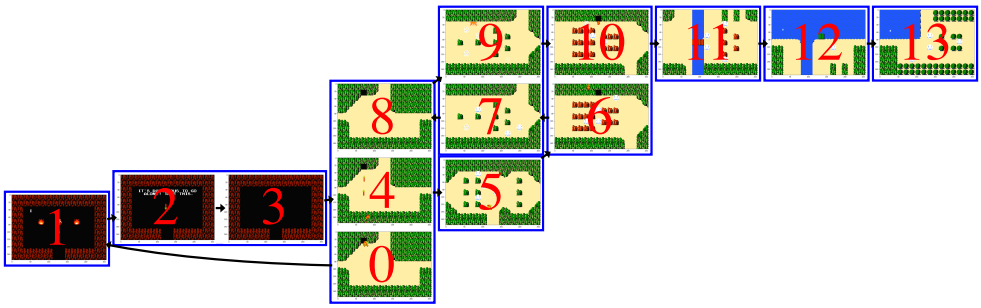

Figure 14.4: Example room linkage detection and room merging step. Red numbers represent the order of traversal. Rooms that are believed to be identical are grouped together in blue boxes. It is up to the user to choose which rooms should or should not actually be merged.

have identical tilemaps but are in fact different rooms. There are also instances with more complex mechanics at play: in *Zelda's* "Lost Woods," the player moves through a sequence of identical-looking rooms and must use the correct door in each of those rooms or return to the first room in the sequence. It is unreasonable to expect that a system could automatically cover all such cases since in the end room connections are defined in opaque game programs and *Mappy* cannot hope to address every possibility. The system therefore leaves it up to a human analyst to select which rooms should or should not be merged. *Mappy* provides suggestions based on overall similarity; in Fig. 14.4, *Mappy* is largely correct (though it misses the fact that 1, 2, and 3 are the same room) and the final map should consist of the merged rooms $(0,4,8),(1,2,3),(5),(7,9),(6,10),(11),(12)$, and (13).

Note that there are important candidate merges *Mappy* does not detect. For instance, there is no method to detect returning to a different part of the same room. Fig. 14.5 shows an example where the player takes a warp pipe (from room 1) to a bonus room (room 2) and then emerges later in the level (room 3). The correct map would show that the first room and the third are actually two parts of one larger room; but even a human player must explore multiple paths

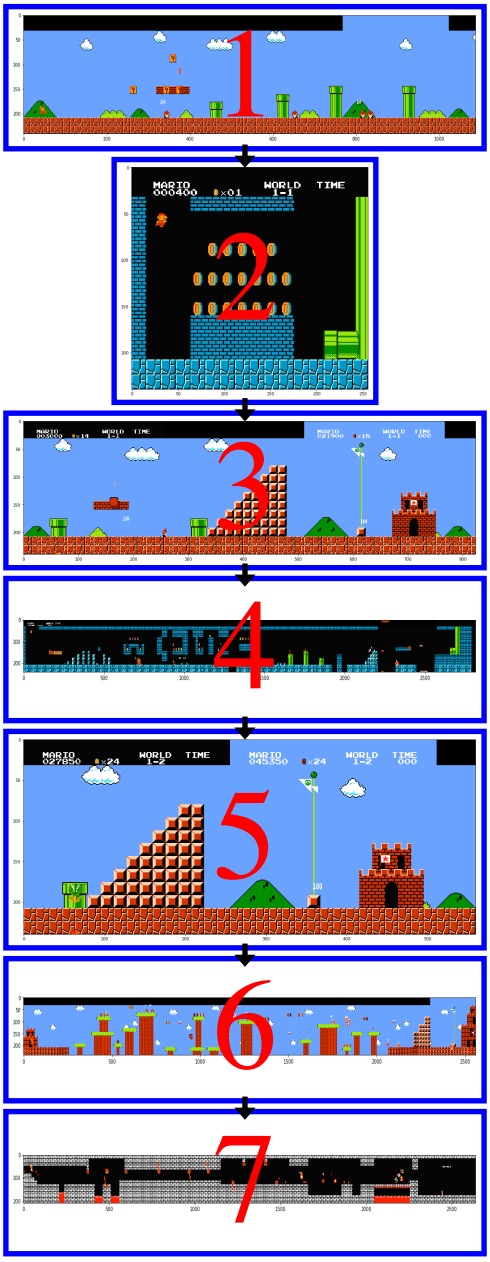

Figure 14.5: The first 4 levels extracted from *Super Mario Bros.*; Level 1-1 is comprised of rooms 1, 2, and 3.

through the level to determine this. In the future computer vision techniques (e.g. [161, 267]) could help with automatically merging the results of multiple play traces, so as to be able to fully map segments of a game that are mutually exclusive or are not likely to both appear in the same traversal.

There is not yet a pleasant user interface for this manual merging process, but I imagine a possible UI closely tied to the visualizations provided in this paper. A human analyst confronted with a map like Fig. 14.6 might begin by merging obviously identical rooms according to *Mappy's* suggestions. Selecting a cluster (e.g., $(0,4,5)$) by clicking on it and then hitting return (or double-clicking on the border of the cluster) would collapse these rooms into one, merging the links in and out at the same time. Next, the two clusters in the top-left corner are actually the same room—but the use of background tiles to render text (combined, perhaps, with loss of control when obtaining the sword) has confused *Mappy*. An analyst could shift-click to select both those clusters and hit return to merge them into one, and then double-click the blue border of that cluster to combine it into a single room. Finally, the treasure rooms of most dungeons in the *Legend of Zelda* look nearly identical. A user could split apart a candidate suggestion by shift-selecting individual rooms of one or more clusters and then hitting return to pull them into their own combined clusters (multiple clusters could simultaneously be combined in the same way, or a combination of clusters and rooms). In this way, individual treasure rooms or other similar-looking rooms could be kept separate.

The images shown in this paper visualize the first appearance of every game object (with the sprite image it appeared as at that time), but one is often really only interested in game objects that are arranged as part of the level (note that *Mappy* shows the first appearance of *game*

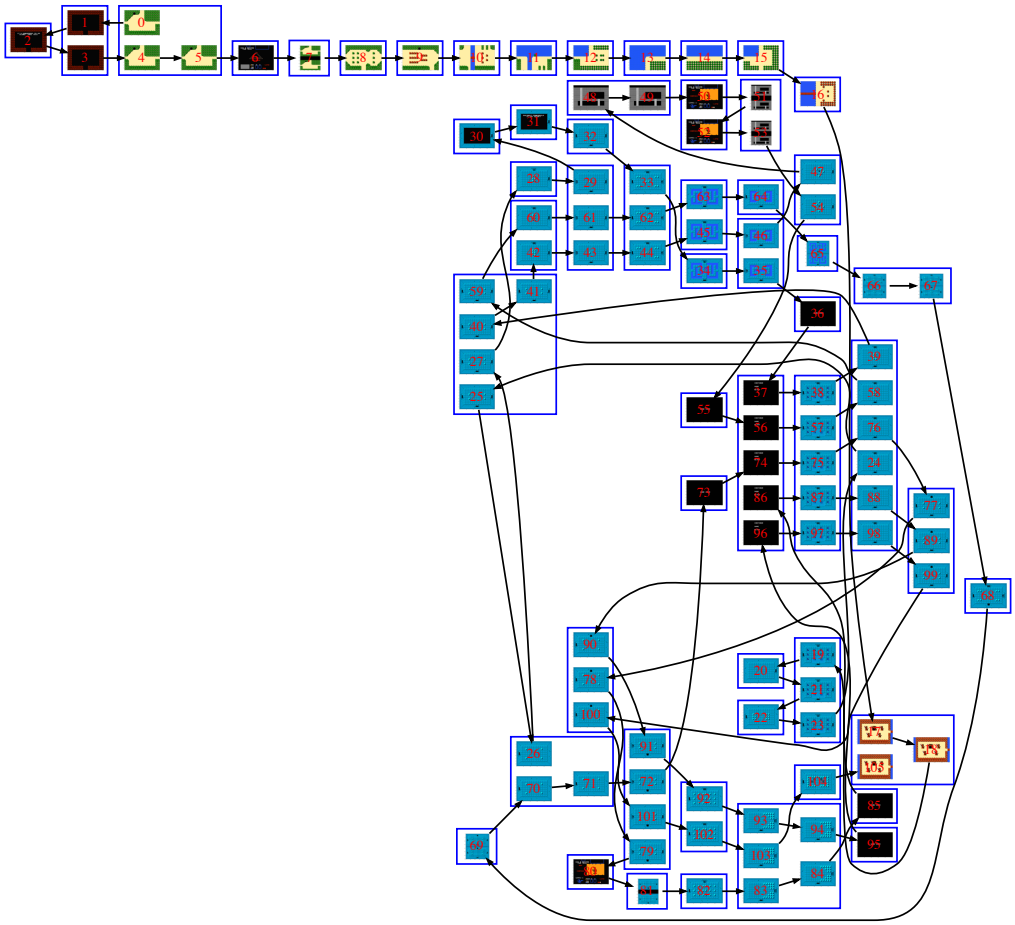

Figure 14.6: *Zelda* through the completion of Dungeon 1. The player (one of the authors) made numerous mistakes resulting in deaths (the cluster of black screens in the middle) which teleport the player to the beginning of the dungeon.

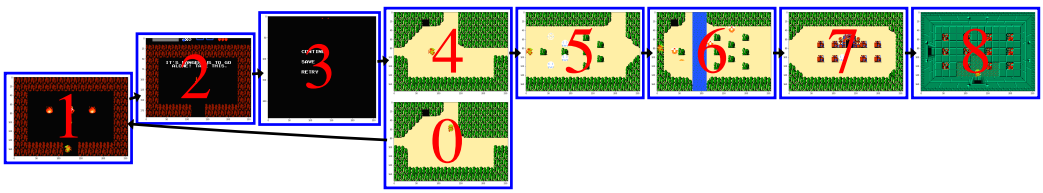

Figure 14.7: *Zelda* up to Dungeon 3, showing a map which is *true* but not *reasonable*.

objects such as sprites, but the 25%-time appearance of *tiles*). In *Super Mario Bros.,* one does not want the `100`-point indicators; in *Metroid* one does not want Samus's shots or the powerups dropped by defeated enemies. Because figuring out which game objects appear because of the evolution of game rules and which sprites appear as part of the level is not a well-defined problem (consider the mushrooms or stars coming out of question-mark blocks), it is left for now to human intuition; however, I hope to resolve this in future work, perhaps leveraging techniques for learning game rules [111] or interactions [241]. There is not yet a graphical user interface for indicating game objects to exclude from the map; currently a *Mappy* user removes unwanted game objects by examining their appearance and then eliminating the corresponding tracked game objects from future processing.

14.4 Discussion

Mappy makes several structuring assumptions about games and play, and it is informative to explore where and how these break down. I have already discussed limitations in scrolling detection and room merging, but there is another important assumption which has not been addressed yet: *Mappy* implicitly assumes it is observing *natural* play where a human explores the game in the way intended by designers and programmers. Here it is useful to distinguish *true* maps from *reasonable* ones. I call maps that accurately reflect game code *true,* even if they are inconsistent with players' expectations of the game's design. A *reasonable* map is one that matches these expectations but might be unfaithful to the source code (e.g., the maps from Zelda Elements mentioned above). Comparing Figs. 14.6 and 14.7 showcases this dis-

tinction. The former is a natural play of the game; but in the latter, a tool-assisted speedrunner utilizes multiple glitches to take an optimal (not at all natural) path. The first is the so-called "Up+A" trick, which causes a soft reset of the game when the player enters the eponymous command on the second controller. This covers the transitions from 2 (picking up the sword) to 3 (the soft-reset screen) to 4 (the player's initial location at game start). The second trick is "Screen Scrolling", which lets the player leave a screen and re-enter that same screen from the other side. This is how the player warps over the river in 6 and (due to collision detection failing when inside an object) passes back through the same wall to room 7.

All this is allowed by the code of the game, and the *true* map that *Mappy* collects captures the full behavior of that code; of course, this would be inappropriate for many of the use-cases suggested in the introduction. A human or AI player would probably want a map that characterizes their understanding of the game world. A user feeding this map to a machine learning algorithm for design knowledge extraction would likewise want a map that conveys the intended (if not actual) progressions in the game. It is also interesting to consider that an optimal AI will find such "secret passages" while an AI that does semantic hierarchical planning (e.g., planning sequences of platforms or rooms to traverse) will probably not. That said, *true* maps can be valuable to a game creator—particularly for highlighting areas where it differs from a *reasonable* map for applications like detecting bugs or sequence breaks.

As for learning map data proper, one important aspect of links which *Mappy* currently ignores is that links are embedded *in space.* In other words, the player usually travels between rooms because the character stepped on a staircase or crossed between rooms. Right now it does not learn the embedding of the network of links into the tilemaps, but this is important

future work. Notably, the same doorway might take the player to multiple different rooms (if, for example, certain game progress flags have been set) or the same room might be entered on the same side from multiple doorways (as in the Lost Woods).

As with AGDL in general, I see natural future work in extending the set of games which *Mappy* can address both on the NES and on other platforms (including black-box games without the hints from dedicated graphics hardware). Many of these techniques will transfer readily, but some of the low-level feature extraction must be adapted to work with additional context or on different hardware, perhaps incorporating more techniques from computer vision. The NES has been productive for these uses, but I do occasionally run into quirks of the hardware that would be avoided with pure computer vision approaches—for example, different games can include custom wiring or even additional memory that the internal tile renderer must handle properly.

Mappy must also be extended to find the scrolling sub-region of the screen automatically. This might be done by observing which portion of the screen seems to move around smoothly with respect to the whole viewport as the playthrough goes on; at any rate it is extremely important for games like *The Legend of Zelda 2: The Adventures of Link,* where the top-down overworld screen has no status bar while the side-scrolling screens do. It is especially important to handle game *modes,* including game-over and stage-start screens, battle versus field modes versus menus in role-playing games, and so on. This is complicated even in *The Legend of Zelda* where the menu activates by smoothly scrolling down and effectively pauses the action on the part of the screen *Mappy* should pay attention to.

I also hope to eventually track the *provenance* of *Mappy's* conclusions about maps.

In other words, I would like to identify for a given link, map, game object position, or other observation what game state (or sequence of game states) witnessed that fact. This could help improve the quality of the system's conclusions—in some cases one may want to interpret a screen transition as indicating either a change in room *or* merely a menu popping up, and tracking why the system believes one or the other conclusion seems useful for optimally resolving the ambiguity. Provenance also could improve the experience of merging rooms: being able to click and load up a pair of similar rooms in an emulator could help an analyst decide if they are indeed the same room. Moreover, this magnifies the utility of search and retrieval over concepts like level fragments, linking structures, or which sprites appear in which rooms. I believe a database that admits querying for cases like infinitely-looping hallways or Lost Woods-style trick dungeons could be useful for scholars of digital games as well as for data-driven PCG, and *Mappy* points the way to building tools like that at the same time it helps improve the coverage of the VGLC.

As mentioned earlier, *Mappy* ought to analyze several play-throughs of a game to get more complete maps. It could even borrow techniques from undirected or curiosity-driven search [237, 203] to reduce the need for human-provided play traces; this could take the form of automatic exploration off of the main branches given by provided traces or even fully automatic search. With my undergraduate independent study mentee Julius Mazer, I have explored some designs for merging rooms together and tracking provenance, using cues like relative location and sizes of rooms as well as similarities in the tilemaps. This work is promising and Adam Summerville and I will pursue it in detail soon; separately, Adam has been finding ways to handle game entities that transition between being tiles and being sprites.

Finally, as an un-optimized prototype, *Mappy's* runtime performance is acceptable but not superb. Mapping out a minute of play (3600 frames) takes between five and six minutes, mainly due to the expensive scrolling detection and game object tracking. Obviously this is an area for improvement and I am actively exploring ways both to parallelize *Mappy* and to bring down its constant factors. One easy way to increase the mapped frames per second (at the cost of missing short-lived tile changes) would be to only make map-related observations every few frames. Initial experiments here suggest that looking at every second frame is a good compromise that almost doubles the exploration speed without sacrificing too much accuracy; looking at every fifth frame roughly doubles the speed again but the results require postprocessing and cleanup to be made usable. This time skipping could also be made adaptive, taking longer steps when the visual appearance is not seen to change rapidly.

Chapter 15

Learning Character Behaviors with CHARDA

In chapter 12, I introduced hybrid automata as a convenient notation for describing action games. I showed their utility for model checking and design support in HyPED; but it may be a bit much to ask every action game designer to learn a new notation, modeling language, and toolset to learn about their games' dynamics. Fortunately, the framework of automated game design learning yields an approach to *learning* hybrid automata automatically from observations of game play. Learning (or recovering) HAs from existing systems yields convenient abstractions for human analysis and high-level automated planning; moreover, these abstractions can be refined, possibly automatically, by acquiring new data or performing additional experimentation. In this chapter I present my work with Adam Summerville on *CHARDA,* Causal Hybrid Automata Recovery via Dynamic Analysis, a non-parametric framework that learns an HA from observations of a dynamical system. [1]

CHARDA has two phases: mode identification and causal guard learning. I identify

[1] *Portions of this chapter originally appeared as "CHARDA: Causal Hybrid Automata Recovery via Dynamic Analysis" [244].*

modes via a dynamic programming approach that segments the trace and finds switchpoints where the dynamics of the system change. Then CHARDA learns causal guard conditions for mode-to-mode transitions using information-theoretic measures.

CHARDA's segmentation requires no prior knowledge of the number of potential modes or the location of switchpoints, requiring only a set of potential model templates (e.g., $\dot{x} = a$ or $\dot{x} := a; \ddot{x} = b$, read respectively as constant velocity or constant acceleration b starting from a reset velocity value a). Although the models can take any form (so long as a likelihood function is available), here I use general linear models (multivariate linear regressions). CHARDA performs model selection and segmentation via a principled penalty function. In this work, I tried both the Bayesian Information Criterion (BIC) and Minimum Description Length (MDL), but CHARDA is also penalty-function-agnostic.

CHARDA demonstrates excellent results in a novel domain: videogames, specifically *Super Mario Bros* (SMB). Games offer a unique set of challenges including non-physical dynamics and potentially very frequent mode transitions on the order of fractions of a second. As a domain, games lie somewhere between synthetic data and a physical robot or other cyber-physical system. Furthermore, games are interesting objects of analysis in their own right. In games specifically, CHARDA has some exciting applications:

- In the General VideoGame (GVG) playing domain, an AI could derive HA models for game entities and then do planning on this abstracted space without relying on a forward model [205]

- Model-checking/safety analysis of character automata without the overhead of manual

modeling by human game designers (as shown in Chapter 12)

- Extracting features for quantitative comparative analysis between games or game rules [97, 116]
- Automatic scraping of characters from existing games for a character behavior corpus, which could then be used for analysis or procedural generation as game levels are already [240]

The rest of the chapter is structured as follows. First, I discuss other approaches to learning dynamical system models and how CHARDA fits into the existing work here. I then briefly introduce the concrete domain of interest and explain CHARDA's design and implementation. Finally, I evaluate CHARDA in two domains: internally on the SMB domain, and externally in an aircraft tracking domain for comparison with another recent automaton learning algorithm.

15.1 Related Work

I introduced hybrid automata in Chapter 12. Given the desirable properties of these models, and the ready availability of tools for dealing with them, many researchers have explored automatically recovering these high-level models from real-world system behaviors. CHARDA shares motivations with HyBUTLA [184], which also aimed to learn a complete automaton from observational data. HyBUTLA seems able to learn only acyclic hybrid automata, since it works by constructing a prefix acceptor tree of the modes for each observation episode and then merges compatible modes from the bottom up. Moreover, HyBUTLA assumes that

the segmentation is given in advance and that all transitions happen due to individual discrete events, presumably from a relatively small set. The overall structure of both algorithms—split the observations into a number of intervals in which mode functions are fit, then merge redundant modes—is similar, but CHARDA learns a larger class of automata and does not require data to be pre-split into episodes or segments.

Santana *et al.* learned Probabilistic Hybrid Automata (PHA) from observation using Expectation-Maximization [213]. At each stage of the EM algorithm a Support Vector Machine was trained to predict the probability of transitioning to a new mode. Unlike CHARDA, their work requires a priori knowledge about the number of modes.

The closest work to ours is that of Ly and Lipson [160] which used Evolutionary Computation to perform clustered symbolic regression to find common modes with the Akaike Information Criterion uses to penalize model complexity. However, unlike CHARDA their work assumes *a priori* knowledge about the number of modes. Moreover, since their work assigns individual datapoints, not intervals, to a mode, their approach can only model stationary processes.

Several approaches have sought to learn models that describe dynamical systems' behavior. Hidden Markov Models [33] learn probabilistic state transitions between a hidden state and the observed data. The Infinite HMM [34] extends this to an unbounded number of states which assumes a Chinese Restaurant Process governs the state space. These approaches do not characterize guard *conditions*, but instead learn the *probability* of taking state transitions at each instant.

Data segmentation has a natural connection to automaton learning, and CHARDA

uses an approach based on least squares regression [37]. Model-based recursive partitioning [276] is an alternative family of techniques which fits a model to the entire dataset and then iteratively and greedily splits that model until reaching a threshold quality level or split count. Unfortunately, each split is only locally optimal so there are no guarantees about global optimality. The Forget-Me-Not-Process [173] finds a partitioning of time segments that allows for models to be repeated across different partitioned segments; however, it only works for stationary processes, i.e., distributions that do not change over time.

Some techniques have leveraged certain inductive biases from biological or other domains to obtain good results. Kukreja *et al.* [143] found models of switched mode systems given prior knowledge about the locations of the switchpoints, and Bridewell *et al.* [47] used exhaustive search over model structures to best explain observed data without assuming fixed switchpoints. CHARDA takes advantage of inductive biases around the behaviors of embodied agents, learning not just what dynamical modes there are but also *why* the character switches from one to another.

In terms of finding abstract models specifically of Nintendo games, I was inspired by Murphy's work [177] in automatically determining physical properties of game characters. That project, like this one, examined runtime memory structures to determine where objects were; they further explored, through experimentation, causal linkages between arbitrary locations in RAM and the visual position of characters on the screen. These relations were used to drive other experiments, e.g., to discover whether game characters fell due to gravity or whether their movement was obstructed by particular types of game objects. In a sense, their work is an *ad hoc* property-based testing approach to learning which of a fixed set of properties holds. CHARDA

requires less domain knowledge and captures the characters' behavior more precisely.

In the future I look forward to combining this more general approach with such knowledge-rich techniques to capture more complicated interactions between multiple agents and their environment. A recent publication by Summerville *et al.* [241] similarly used games as their domain, attempting to find causal interactions shared by different entities, and I have built on this approach for the causal guard learning.

15.2 Domain

CHARDA learns hybrid discrete/continuous behaviors of videogame characters or other agents whose inputs and movement behavior are observable. I obtain these inputs from an example playthrough of a game (e.g., SMB), assuming these inputs are representative of the character in question. Replaying this input sequence once through a software emulator of the game's hardware platform, I read out high-level features from the simulated graphics hardware and assemble those into distinct agents whose positions are tracked over time. Importantly, characters may pop in and out of existence, collide with fixed or moving obstacles of various types, or perform other arbitrary (often non-physical) behaviors. CHARDA can only observe characters' positions at a resolution of 1 pixel (a character is generally 8–32 pixels high); even then, the game world and CHARDA's sensing are at a $\frac{1}{60}$-second fixed discrete time interval. All the position readings are therefore inaccurate by up to one spatial unit, and these errors naturally propagate to velocity and other calculations.

The input to the automaton learning process for a single entity is: sequences of dis-

crete variable values that are possible control inputs (e.g., button presses), continuous variable values, and sets of predicates describing facts in external theories such as collision (e.g., the character was touching an object with appearance A at time t on one side or another). The goal is to go from that input data, presumed to be representative of the entity's "true" behaviors, to an abstraction suitable for planning or other purposes. This type of data is not hard to obtain for cyber-physical systems under the analyst's control or in cases where the possible causes for behavior change can be observed at some precision (even a probability distribution for these causes would suffice).

In this work, I look at learning a constrained class of hybrid automata from a combination of controlled (or at least witnessed) inputs and observed outputs. Specifically, though the learned automata may have any structure in terms of the number of modes and transitions, the modes may only have flows from a given set of model *templates*. In this specific work (and without loss of generality), every mode's flow condition is a specialization of $\ddot{x} = a, a \in \mathbb{R}$; moreover, all transitions leading into a given mode are forced to have the same update function, either $\dot{x} := n, n \in \mathbb{R}$ or the empty update. Finally, the set of guard conditions is currently assumed to be conjunctions of predicates from a given labeled set. The causal learning component learns which of these predicates is most associated with the transitions, and prefers those predicates which are more strongly causal. There is no reason these guards could not also be learned as (for example) linear inequalities, since the system knows the set of modes and their active intervals at the time of cause assignment. Again, I focus here on learning reasonably small over-approximations of the true model: these can always be refined, but I don't want to exclude any witnessed behaviors.

15.3 Method

I break down the hybrid automaton learning process into two parts: Identifying modes and determining causes for transitions. Again, these algorithms operate over a sequence of continuous variable values and a sequence of sets of predicates describing the automaton's environment at each instant. I roughly follow the classic dynamic programming solution to the segmented least squares problem [37] with a number of distinctions:

- Different model templates are considered for each segment, instead of a single least squares regression
- A principled penalty instead of a hand-chosen constant
- Merging segments if it results in a more optimal model

15.3.1 Mode Identification

The mode identification process first requires the construction of all possible models for all possible sub-intervals. Let T be a table of model parameters with one entry for each interval i, j and model template m. Then define T's entries as:

$$T[i,j,m] = \text{train}(m, d[i:j]) \quad \text{s.t.}$$

$$1 < i < j < n, m \in \mathcal{M}$$

where n is the number of potential switchpoints, $\mathcal{M}$ is the set of model templates, d is the dataset, and $T[i,j,m]$ is the model of template m trained on data from the interval of i to

j. For this work the set of models are all multivariate regressions, but the approach is general enough to work with any model that supports a likelihood function $\mathcal{L}(m|d)$.

The cost for a given model m for sub-interval i to j is therefore:

$$C[i,j,m] = -\log(\mathcal{L}(T[i,j,m]|d[i:j])) + \text{pen}(m, d[i:j])$$

given the penalty criterion pen. This work considers two penalties for model complexity. It requires a principled measure for model complexity for the selection of a given sub-model for an interval, for when a break should occur (due to the inclusion of a switch point increasing model complexity), and for when a merging of modes should occur (due to the inherent fact that two similar but distinct modes are more complex than one mode). To that end I considered both the Bayesian Information Criterion (BIC) [216] and the Minimum Description Length (MDL) [238].

$$\text{BIC} = -\log(\mathcal{L}(m|d)) + \dim(m)\log(n)/2$$

$$\text{So: } \text{pen}_{\text{BIC}} = \dim(m)\log(n)/2$$

Where $\dim(m)$ is the number of parameters in model m and n is the number of datapoints in dataset d.

$$\text{MDL} = -\log(\mathcal{L}(m|d)) + \dim(m)(1+\log(n)/2)$$

$$\text{So: } \text{pen}_{\text{MDL}} = \dim(m)(1+\log(n)/2)$$

The two measures are very similar, being asymptotically the same, but differ in the constants applied to the penalty term. BIC assumes a Bayesian standpoint and determines which model from a set of models is the true model. It operates asymptotically as n trends to ∞, given a fixed

loss for choosing the wrong model. MDL instead takes an information theoretic standpoint and assumes a spike-and-slab prior distribution for each parameter. Given that prior it takes approximately $(1+\frac{\log(n)}{2})$ bits to encode the parameter: 1 bit for whether the parameter $\theta = 0$ (whether it is a slab) and $\frac{\log(n)}{2}$ bits to encode its value (if it is a spike).

For all segments that end at point j CHARDA finds the optimal model and segmentation that leads to that point. $O[j]$ is the optimal cumulative cost of models across segments up to datapoint j.

$$O[j] = \begin{cases} 0 & \text{if } j = 0 \\ \text{argmin}_{0 \leq i \leq j, m}\ C(i,j,m) + O[i-1] & \text{if } j > 0 \end{cases}$$

The system uses dynamic programming to work backwards from the last switch point, finding the optimal sequence of segments that produces the optimal set of models.

After segmentation, the segments' models are merged if this will improve the overall attractiveness of the entire model, namely by reducing the number of parameters in the overall model by a large enough amount that the decrease in complexity is greater than the decrease in likelihood.

This is accomplished by constructing a new model from the data for segments s_1, s_2 concatenated to $d[s_1 s_2]$:

$$m_{s_1 s_2} = \text{argmin}_{m \in \mathcal{M}}\ \log \mathcal{L}(\text{train}(m, d[s_1 s_2]) | d[s_1 s_2])$$

The overall sequence of models is improved by the merging if the following inequality holds:

$$-\log(\mathcal{L}(m_{s_1 s_2} | d[s_1 s_2])) + \text{pen}(m_{s_1 s_2}) <$$

$$-\log(\mathcal{L}(O[s_1])) - \log(\mathcal{L}(O[s_2])) + \text{pen}(O[s_1]) + \text{pen}(O[s_2])$$

15.3.2 Guard Learning

From these merged modes, causal guarded transitions between modes are learned by finding probabilistically likely conditions where the direction of causality is known. The target domain comes with some advantages for ascribing causality, namely that it offers inputs supplied by a player and that the system can be sure of the direction of causality regarding them; however, any domain that allows for instrumentation of exogenous inputs can utilize the same methodology. Another potential source of causal transition guards in this domain is collisions between visible entities, of which, again, the system can be sure of the direction of causality. CHARDA also looks at endogenous variables as a last resort (and then mainly qualitatively), since causality is much harder to ascertain: for example, if the character enters a mode with flow $\dot{y} = -4$ it could be that $\dot{y}$ is saturating at a terminal velocity, or it might be for some other reason.

For the SMB domain CHARDA considers the following set of predicates for guard condition learning:

- *Control I* (Pressed; Held; Released) — A change in the binary control input *I* — *Exogenous*
- *Collision* with *X* from direction *Y* — Collision with another entity, *X*, from a given direction *Y* — *Exogenous*
- 0-*in,* 0-*out by Sign* - A zero crossing or touching in velocity and its characteristics (e.g., from negative to positive, or vice versa) — *Endogenous*

- *Velocity Extremum* - $\dot{x} = \text{ext}_{\pm}(s)$ - the velocity is roughly equal to the extremum for a given mode s — *Endogenous*
- *Acceleration Sign* — $\ddot{x}$ has the sign -1, 0, or 1 — Endogenous
- *Velocity Sign* — $\dot{x}$ has the sign -1, 0, or 1 — *Endogenous*

The *Control* and *Collision* predicates are given priority as CHARDA can be sure of their direction of causality.

Summerville *et al.* had previously used Normalized Pointwise Mutual Information (NPMI) to learn semantic information about game objects [241], which led me to believe that CHARDA could determine transition guards using a similar technique. CHARDA calculates the NPMI of each transition from a predecessor mode to a successor mode with each predicate active during the predecessor mode. NPMI is a scaling of pointwise mutual information defined as:

$$\text{npmi}(x,y) = \frac{\text{pmi}(x,y)}{-\log(p(x,y))} = \frac{\log(\frac{p(x,y)}{p(x)p(y)})}{-\log(p(x,y))}$$

NPMI for two events is -1 when they never co-occur, 0 when independent, and 1 when they always co-occur. CHARDA applies two different thresholds for NPMI: 0.9 for *universal* (present all, or nearly all, the times that transition is taken) events and 0.4 for *relevant* events. For example, to learn the cause for transitions from hypothetical mode **A** into mode **B**, CHARDA looks at all time intervals where **A** is active, determines for each predicate how strongly correlated it is with the transition event $\mathbf{A} \Rightarrow \mathbf{B}$, and takes all those passing a threshold to be causes. These correspond to conjuncts in the guard condition. Those correspondences which are high enough

to be of interest but do not meet the threshold are called *relevant* and are possible disjuncts in the guard condition (assuming it has the form $c_1 \wedge \ldots \wedge c_i \wedge (d_1 \vee \ldots \vee d_j)$). If CHARDA has an exogenous explanation, it discards endogenous explanations.

There may be cases where out-transitions of a mode are non-deterministic: they have identical causes, or one's causes subsume another. In these situations the offending target modes are merged, one pair at a time, re-connecting edges as necessary until a fixpoint is reached. This merging greedily abstracts the true automaton, but in practice it seems to work well for domains like game characters whose discrete state changes are generally strongly tied to control inputs or collisions; future work will explore more sophisticated approaches to resolving non-determinism.

15.4 Evaluation

This work has been evaluated in two domains: Aircraft Dynamics Modeling and Mario's Jump Dynamics from SMB.

First, I explore the use of CHARDA in aircraft modeling for a direct comparison with Santana *et al.* [213]. Their approach used Expectation Maximization [84] to recover a hybrid automaton from observational data by iteratively refining an Interactive Multiple Model. Guard conditions were learned by applying support vector machines. As in Santana's work, I also include results for a Jump Markov Linear System (JMLS) which assumes Markovian transitions.

The aircraft model is given in two distinct scenarios: the first, "Lawnmower" (see Fig.

15.1), features an aircraft moving in a constant velocity for some period of time and then making a constant-rate turn to reverse heading, repeating this pattern for some number of iterations. In the second scenario, "Random," the aircraft makes a given maneuver (either constant heading or constant turn) for 50 time steps and then changes to a random maneuver; this is repeated 17 times. I must note that this portion of the evaluation is only based on CHARDA's segmentation algorithm and does not employ transition guard learning. As the observational data offers no causal information indicating why a mode transition might be made, CHARDA does not learn any causal transition guards (which would simply overfit the given observations).

As in Santana's work, I ran 32 trials and discarded the best and worst runs; the results are shown in Table 15.1. For the Lawnmower domain, CHARDA outperforms Santana *et al.*, but both are close enough to the ground truth that the difference is negligible. In the Random domain CHARDA outperforms the prior work dramatically because the segmentation is not based on learning linear guards; CHARDA instead finds an optimal segmentation based on model accuracy and complexity. I must note again that there are no real causes for why the aircraft changes maneuvers, so it is impossible to learn true causal guards. Santana *et al.* learn correlative guards for a given training instance, but their learned guards are not applicable to unseen data because they are tuned to that specific training instance (for example, if the aircraft's flight pattern was rotated or translated, all of their learned guards would be invalidated due to their training domain and linear nature). As such, I feel that it is only relevant to compare the segmentation portion of CHARDA to the prior work. CHARDA would be better-suited if the domain were framed as a control problem and the dataset contained features like operator controls and aircraft sensors.

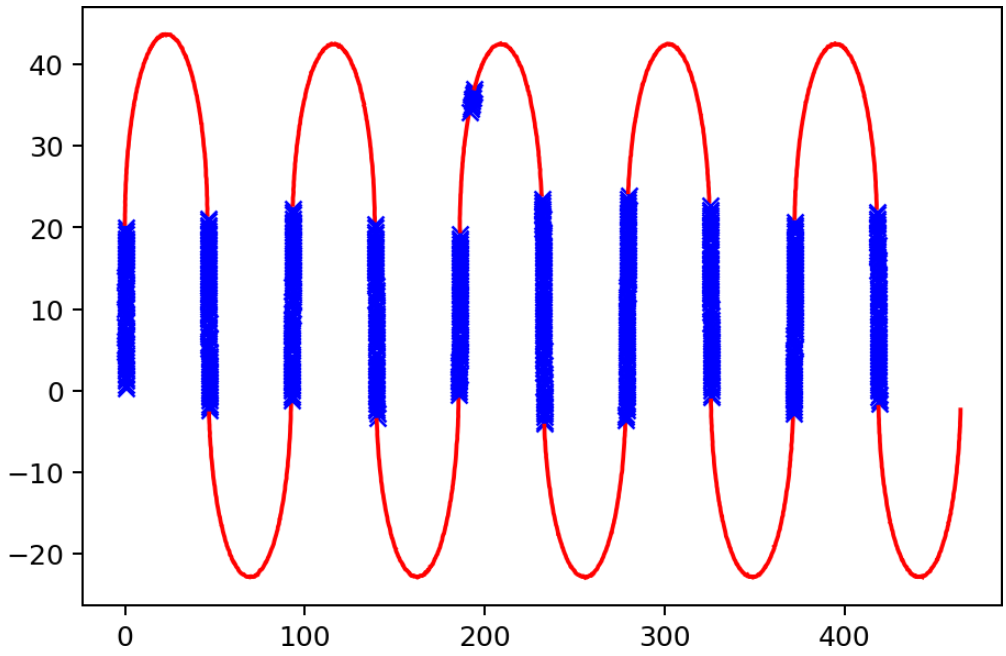

Figure 15.1: The Lawnmower data, as segmented by CHARDA with BIC. Beyond having slight errors on the beginning and end of the turns, there is one turn where it incorrectly reverts to a constant velocity in the middle. *Only the switch point detection portion of CHARDA is used.

Method	Data	Attribution Error
JMLS	Lawnmower	53.93%
PHA	Lawnmower	3.33%
CHARDA	Lawnmower	**2.75%***
JMLS	Random	58.91%
PHA	Random	63.2%
CHARDA	Random	**4.10%***

Table 15.1: Percentage of modes misattributed for CHARDA, PHA, and JMLS. The results shown here are only based on the segmentation portion, and do not include causal guard learning as there are no causal reasons for the mode transitions.

On Ground $\dot{y} = 0$ — Caused by Mario colliding with something solid from above

Jump(1,2,3) Three jumps with parameters:

- $\dot{y} := 4, \ddot{y} = -\frac{1}{8}$
- $\dot{y} := 4, \ddot{y} = -\frac{31}{256}$
- $\dot{y} := 5, \ddot{y} = -\frac{5}{32}$

Entered from **On Ground** when the **A button** is pressed and $|\dot{x}| < 1$, $1 \leq |\dot{x}| < 2.5$, or $2.5 < |\dot{x}|$, respectively

Release(1,2,3) $\dot{y} := min(\dot{y}, 3)$ — Entered from the respective **Jump** when the **A** button is released; $\ddot{y}$ same as respective **Jump**.

Fall(1,2,3) Falling at one of three rates: $\ddot{y} = -\frac{7}{16}$, $-\frac{3}{8}$, or $-\frac{9}{16}$; entered from the respective **Jump** or **Release** mode when the apex is reached ($\dot{y} \leq 0$)

Terminal Velocity(1,2,3) $\dot{y} = -4$ - Entered from **Fall** when $\dot{y} \leq -4$. The initial timestep in the **Terminal Velocity** state is actually $\dot{y} = -4 + \dot{y} - \lfloor \dot{y} \rfloor$ before being set to -4.

Bump(1,2,3) $\dot{y} := 0$ — Entered from a **Jump** or **Release** when Mario collides with something hard and solid from below; $\ddot{y}$ same as respective **Jump** or **Release**

SoftBump(1,2,3) $\dot{y} := 1 + \dot{y} - \lfloor \dot{y} \rfloor$ — Entered from a **Jump** or **Release** when Mario collides with something soft and solid from below; $\ddot{y}$ same as respective **Jump** or **Release**

Bounce(1,2,3) $\dot{y} := 4, \ddot{y} := a$ — Entered when Mario collides with an enemy from above; a is given by the respective **Jump**, **Release**, **Fall**, or **Terminal Velocity** state

Figure 15.2: The true HA for Mario's jump in *Super Mario Bros.* $:=$ represents the setting of a value on transition into the given mode, while $=$ represents a flow rate while within that mode.

For the Mario domain, CHARDA makes no assumptions about the number of true modes and lets the non-parametric approach recover the correct modes on its own. This means that it cannot be compared directly to Santana *et al.* as their work requires the number of modes a priori, so instead I compare these results to a manually-defined automaton based on human reverse-engineering of the game's program code [128] (see Fig. 15.2). I present the HAs learned by CHARDA in Figures 15.4 and 15.5. The Mario trace used for this work was 3772 frames in

length, 63 seconds. The learned HAs are over-approximations of the true HA. Whereas the true HA has 3 separate jump modes based on the state of $\dot{x}$ at the time of transition, the learned HAs have only one such jump whose parameters are averages of the parameters of the true modes. Following from learning just one jump, CHARDA learns only a single falling mode. MDL does learn that releasing the **A** button while ascending leads to a different set of dynamics, but it considers this a change in gravity as opposed to a reset in velocity.

MDL produces the more faithful model of the true behavior, but is overzealous in its merging of the distinct jump mode chains into a single jump mode chain. As such, it only recovers 7 of the 22 modes; however, abstracting away the differences between the jump chains it learns 7 of 8 modes, only missing the distinction between hard bump and soft bump. A comparison of the modeled behaviors and the truth can be seen in figure 15.3.

15.5 Conclusion

I have presented CHARDA, a novel combination of techniques (dynamic programming with a grounded penalty for data segmentation, causal relationship learning) that can recover hybrid automata from observations of a dynamical system. CHARDA outperforms an existing HA learning algorithm in data segmentation, and in a well-suited domain can find causal (not merely correlative) transition guards. I have also shown hybrid system identification in a novel domain, videogames, that comes with an interesting set of challenges (short time durations, non-physical dynamics) and benefits (full access to all command inputs).

The use of a well-founded penalty criterion in conjunction with the dynamic program-

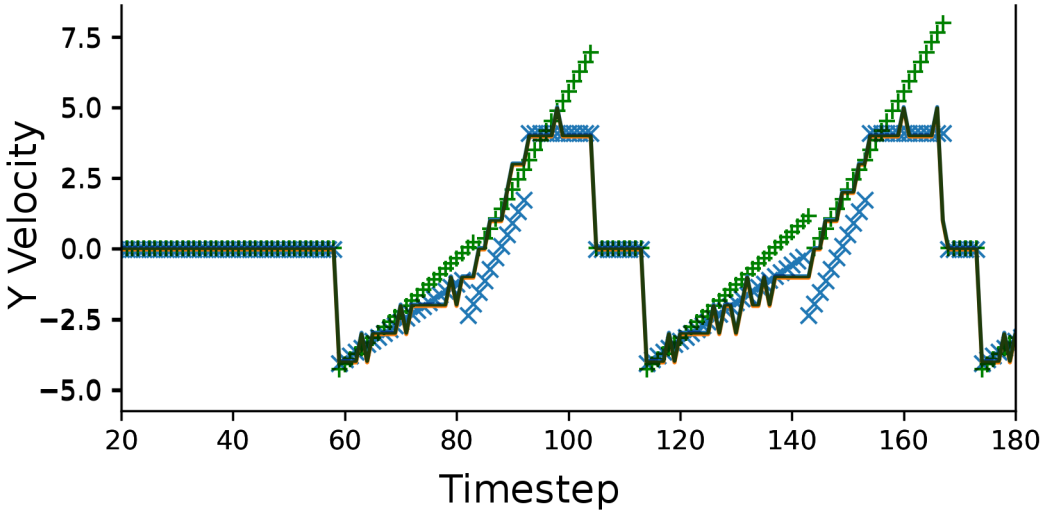

Figure 15.3: Modeled behavior using MDL criterion (Blue X) and BIC (Green +) vs true behavior (Black Line). MDL's largest error source is resetting to an specific value when the true behavior involves clamping to that value, whereas since BIC learns to transition at the 0 crossing it has a more accurate reset velocity. BIC does not learn the transition from **Falling** to **Terminal Velocity**. MDL has a Mean Absolute Error (MAE) of 0.522 while BIC has an MAE of 0.716.

On Ground $\dot{y} = 0$ — Caused by Mario colliding with something solid from above

Jump $\dot{y} := [3.97, 4.10], \ddot{y} = [-0.140, -0.131]$ — Entered from **On Ground** when the **A button** is pressed

Release $\dot{y} := [2.10, 2.54], \ddot{y} = [-0.430, -0.384]$ — Entered from **Jump** when the **A button** is released

Fall $\dot{y} := 0, \ddot{y} = [-0.373, -0.359]$ — Entered from **Jump** or **Release** when the apex is reached

Bump $\dot{y} := [-1.85, -1.27], \ddot{y} = [-0.324, -0.238]$ — Entered from **Jump** when something solid is collided with from below

Bounce $\dot{y} := [3.51, 3.82], \ddot{y} = [-0.410, -0.378]$ — Entered from **Jump** when an enemy is collided with from above

Terminal Velocity $\dot{y} = [-4.15, -4.06]$ — Entered from **Jump** or **Fall**

Figure 15.4: HA with MDL as the penalty.

On Ground $\dot{y} = 0$ — Caused by Mario colliding with something solid from above

Jump $\dot{y} := [4.19, 4.42], \ddot{y} = [-0.195, -0.181]$ — Entered from **On Ground** when the **A button** is pressed

Fall $\dot{y} := 0, \ddot{y} = [-0.356, -0.338]$ — Entered from **Jump** when the apex is reached

Bump $\dot{y} := [-2.37, -1.67], \ddot{y} = [-0.289, -0.188]$ — Entered from **Jump** when something solid is collided with from below

Bounce $\dot{y} := [3.52, 3.88], \ddot{y} = [-0.424, -0.391]$ — Entered from **Jump** when an enemy is collided with from above

Terminal Velocity $\dot{y} = [-4.16, -4.05]$ — Entered from **Jump** when the threshold of -4 is reached.

Figure 15.5: HA with BIC used as the penalty

ming approach is only one of many possible segmentation techniques, and it remains future work to test the general framework of *Segmentation + Guarded Transition* learning with other techniques. However, the biggest source of error in the learned HAs comes not from mistakes in segmentation, but rather from overzealous merging of modes. The learned parameters at segmentation in fact do describe modes in line with **Jump1** and **Jump3** (i.e., $\dot{y} = 4$ vs $\dot{y} = 5$), but these modes are merged together since it improves the overall learned model according to the criterion. It remains for future work to determine if there is a different principled way to learn these similar but distinct modes. It is also future work to incorporate techniques from other approaches, such as mode assignment via a Chinese Restaurant Process or the Forget-Me-Not Process, to pool modes at segmentation time instead of a post-segmentation merge process.

Beyond improving segmentation, there are also possible improvements to learning guarded transitions. Assuming CHARDA had perfect segmentation and mode assignment, it

would still not be able to fully capture the guarded transitions of Mario given that its transition learning scheme does not have knowledge of Mario's horizontal velocity, nor is it able to learn transitions based on comparisons to arbitrary thresholds. In some domains, experimentation is possible: a system might be able to control the dynamical system in question or to put it into situations where its behavior could be informative. I would like to explore this to improve the precision of the analysis, either by helping to split truly distinctive merged modes or by testing hypothesized guard conditions.

My results on *Mappy* and CHARDA, which were obtained (along with several other projects) within just a few months, shows that operational logics are a powerful tool for knowledge representations in automated game design learning. These are just the beginnings of what could be a complete research agenda in AGDL: to achieve a new result in specification recovery, select a composition of operational logics, find a corresponding knowledge representation and learning strategy for capturing the design information relevant to those logics, and then just *run it* against games of interest.

This is a *general-purpose* strategy for coming up with useful knowledge representations for AGDL projects. It pairs well with the general-purpose strategies for game modeling and support tools outlined in Chapters 9 and 10: one could readily imagine round-tripping a learned model from an arbitrary game into those analysis tools and then back out to enable lightweight verification of games *in general.*

Chapter 16

Conclusion

This part of the dissertation has explored the use of operational-logics-based knowledge representations in two main areas: design support via model checking and automated game design learning. Some work has yielded convincing successes (e.g., CHARDA) while other efforts have opened up new areas of inquiry and opportunities to follow up (e.g., Reductionist). Once I began refining and working from the theory of operational logics I was able to produce significant new work in these areas much more quickly than before; I believe that they are of general interest and use to others as well.

I have argued throughout this dissertation that operational logics are an ideal starting point for projects in game analysis and game modeling. I have shown my work from the last four years (since I began to explore operational logics) in the context of the dissertation's new theoretical developments. To repeat the claims of my introduction, I make four key contributions to game studies and game design support:

1. The refinement of operational logic theory (Chapters 1 – 6), including rephrasings of

existing critical tools and computational systems in those terms.

2. The operationalization of operational logics, down to the level of specifying complete games in terms of how they compose operational logics (Chapters 7 – 8).

3. The exposition of strong connections between operational logics and formal logics, as a starting point for knowledge representation work (Chapters 9 – 10).

4. The elaboration of five significant computational-intelligence systems, all of which I constructed in the past two to three years, based on such knowledge representations as evidence for an operational logics-first approach (Chapters 11 – 15).

Each of these contributions has a through-line which does not necessarily map onto the chapter order (these are summarized in Table 16.1). The first such theme is the development of a complete account of operational logics, both in the sense that (as presented here) they can describe a complete game design and in the sense that they are shown as a possible underpinning of two productive approaches to constructing game studies arguments: proceduralist readings and playable models (this amounts to the entirety of the first part of the dissertation). This completes the game studies project initiated in the introduction, expanding those theories in useful new ways and yielding new approaches to extending Gemini in Chapter 8. Moreover, their use in Part 2 acts as a kind of further validation: of course Mappy is confused by game menus (as discussed in Chapter 14), because it does not know about game mode logics—Mappy could naturally be enhanced to deal with menus properly by extending its knowledge representation to account for game mode logics.

The second main theme in this dissertation is the theory and construction of game

modeling languages and design support tools, grounded in operational logics. A key move here was refounding the project of game design support (which, for me, began in building game-making tools for colleagues during my MFA) on the basis of operational logics, instead of on concepts like game genre or on specific games or game engines. The work from the OL catalog (Chapter 3) and theory on OL composition (Chapter 4) up through defining games in terms of OLs (Chapter 7) and the direct procedural operationalization (in Chapter 8) branched out into model-checking work via Game Design Modulo Theories (Chapter 9) and my research on game design support tools (Chapter 10) into Reductionist (Chapter 11) and HyPED (Chapter 12). These latter two tools were explicitly grounded in operational logics and tested the notion that operational logics' composition could be mapped onto the composition of logical theories. This intuition also seems to hold for statistical machine learning techniques and perhaps even abductive logic, as evidenced in the third major theme of this dissertation.

This third theme is Automated Game Design Learning, which had been published separately and was presented on its own in Chapter 13, but for me always seemed a natural dual to my work on modeling languages and game design support. This theme itself has two key areas: first, the logical operationalization in Chapter 8 yields Constellation as a future development of Gemini. Constellation, being grounded in OL theory, enjoys the same orthogonality and compositionality possessed by OL theory (and which eluded Gemini); moreover, its double nature as both a generator and a mechanical deep-reader means it could support a variety of yet unexplored approaches to AGDL based on logical inference and optimization. I therefore see Constellation as a purely symbolic analogue to the statistical methods employed in the automated game design learning examples in Mappy (Chapter 14) and CHARDA (Chapter 15).

Mappy and CHARDA extract truthful and useful game design information just by observing game play—other projects in AGDL could conceivably perform active learning or automated experimentation to resolve ambiguities and refine their learned models. The key invention of this dissertation for the purposes of this theme, as it was for the second theme, is the establishment of the connections between operational logics and formal models in Chapter 9. This yields an invaluable tool for coming up with AI knowledge representations that are both relevant to the operational logics employed in games of interest *and* computationally feasible to learn or approximate automatically.

The final theme is a slice through the second part of the dissertation: Mappy (Chapter 14), CHARDA (Chapter 15), and HyPED (Chapter 12) make up the pieces (yet unassembled) of a complete circuit for round-tripping game levels and character behaviors in and out of model-checking and verification, opening up such tools to audiences who would never want to take the time to learn a game modeling language. Given a graphical logics game, Mappy can turn visual phenomena into a (part of a) game level as linked spaces; CHARDA can learn the behaviors of game characters in a hybrid automata format and determine which types of level terrain are (for example) solid for different characters; and HyPED takes those linked-spaces and hybrid automata representations as input, allowing for both interactive play and for directly checking properties of the observed game design. HyPED models can then readily be exported into other game engines or game-making tools (perhaps the very tool from which the HyPED model was originally harvested via CHARDA). Such an end-to-end tool could leverage the insights of Chapter 10 in an even more generic way, using some mechanism like Playspecs (or a metric extension to Playspecs that considers events in continuous time) to address arbitrary

graphical logic games. As in the discussion above around improving Mappy by expanding its set of addressable operational logics, it stands to reason that HyPED or CHARDA could be incrementally extended with new logics—as long as the axes of operational logic composition can be left to a designer or analyst, this could lead to extremely general-purpose yet *structured* game prototyping tools, without requiring that game developers learn yet another programming language.

The work I have presented here is significant, substantial, and most surprisingly *recent:* all these projects had their initial inception just a few years ago and have already yielded over a dozen publications in the last two years alone.

Chapter	OL Theory	Game Modeling	AGDL	LW Verification
Operational Logics	Chapter 2	Chapter 2	Chapter 2	Chapter 2
Cataloging OLs	Chapter 3	Chapter 3	Chapter 3	Chapter 3
Composing OLs	Chapter 4	Chapter 4	Chapter 4	Chapter 4
Proceduralist Readings	Chapter 5		Chapter 5	
Playable Models	Chapter 6			
Defining Games via OLs		Chapter 7		Chapter 7
Operationalizing OLs		Chapter 8	Chapter 8	Chapter 8
GDMT		Chapter 9	Chapter 9	Chapter 9
Design Support Tools		Chapter 10		Chapter 10
Reductionist		Chapter 11		
HyPED		Chapter 12		Chapter 12
AGDL			Chapter 13	Chapter 13
Mappy			Chapter 14	Chapter 14
CHARDA			Chapter 15	Chapter 15

Table 16.1: Major themes of the dissertation, and chapters in which they are elaborated. Abbreviations: OL for Operational Logics, AGDL for Automated Game Design Learning, LW Verification for Lightweight Verification, GDMT for Game Design Modulo Theories.

16.1 Open Questions

Still, important open questions remain: How can continuous and discrete aspects of game play like resource flows and game-modes be usefully composed? Can a design support system analyze idle games or strategy games where aggregates and expected values are as important as (if not more important than) specific routes to a given outcome? Is it possible to generalize compositional analysis to account for compositions of arbitrary logics? Or must compositional analysis always proceed logic-by-logic or composition-by-composition, as in Chapter 12, where one could assume that linked spaces could be simulated independently?

As for automated game design learning, could one do CHARDA or Mappy-like work directly from vision and still benefit from operational logics-inspired knowledge representations? Could one teach Mappy to learn the difference between foreground and background and, indeed, to handle games that don't use tiled graphics? How can modular and compositional knowledge representations on the game design learning side tackle problems like general RPG playing [187]?

Could this work help move design support beyond solvability? Game design support generally has focused on the automated search for (and visualization of) solutions or paths through levels or other scenarios assuming an idealized player. This is a well-defined problem which admits approaches from general game-playing, motion planning, and other areas of artificial intelligence; playing the game is reduced to winning or solving the game. Such solution-oriented approaches, while straightforward, are only one way of implementing design support.

Play traces are interesting to study because the set of traces produced by a game in some sense defines that game [179]; the problem becomes one of sampling from the space of possible traces in the most useful way [227]. Design support work in the dynamics-oriented category usually finds specific sets of traces (e.g., winning, losing, or problematic traces) directly [230] or explores the possibility space as widely as it can [32, 263]. Both approaches assume an *idealized player*—either a wholly synthetic player or one constructed by amalgamating real-world play data. This might be the player who always plays optimally, the player who never uses a particular ability [125], or players with knowledge of the right policy but degraded reaction time [123].

If one wants to automate parts of the playtesting process [181], the prime candidate for mechanization is the player; but it is likely that dynamics-oriented approaches are not mainly limited by the considerable technical challenge of general game playing, but by the theoretical challenge of finding interesting and useful idealized players. One often assumes that the player optimizes their score or time; and it is generally thought that if such a player can beat a game and obtain a reward then the game possesses a necessary (but not sufficient) property for the game to be good. This is a low bar for the research community to set for ourselves—designers likely already have a good sense of whether a game or level is solvable, because they know much more about the design than an algorithm could. Finding distinctive ways to solve a level can be useful [263], but it seems unlikely to be especially surprising. Even worse, when I conducted an empirical evaluation of this kind of design support system I found that a relatively polished puzzle solver integrated into a game development tool *did not improve* (novice) designers' ability to fix bugs in puzzle levels [191]. Was this study flawed, was the tool flawed, or

is there actually a risk that solvers do not help designers fix design problems?

What else can automated playtesting ask of a game besides "can it be won?" Solvability with constraints on acceptable solutions (as in the *Refraction* work) is more interesting than mere solvability, but it requires well-defined design properties and is often limited to local problems of relatively small size. One can also imagine a system which determines whether a game level is solvable under some (synthesized) assumptions—in other words, what is true of some, most, or all solutions? If I have a character in a platformer, I am less interested in the availability of a path across the entire level than I am in the question of whether the character could get to point B from some point (or platform) A—overall reachability can easily be obtained by combining these more designer-relevant partial solutions. Relatedly, I might want to know if a level segment is *probably passable* only if the character enters it with a certain amount of health or if the player has a certain bound on reaction time.

Tools like these may give some useful feedback to designers, but is reachability essentially *boring* as a universal design criterion? Designers might prefer to work, for example, in terms of odds and expected values and not binary success flags—to see how the probability of success correlates to parameters on the character's starting condition or the player's level of fatigue. Many games have some slack in their design and admit broad varieties of solutions, of which some are intended and some are unintended (perhaps they are safe but too tedious to enjoy), and naïve solution search does not help characterize or prevent such problems. Designers may want to know whether the solution is tedious or the level is hard; it does not matter if there are billions of solutions if they are all unpleasant to play. Detecting when a previously available solution is no longer feasible, as in Inform 7's Skein [182], also seems promising, and I hope to

explore this type of lightweight design support.

Stepping away from solvability, what are some other roles for design support? Zook *et al.* phrased playtesting as an active learning problem, automating away the *designer* as opposed to the player for constrained parameter tuning problems [279]. The key insight is that a playtest is a source of design-relevant information, and an AI system could take steps to maximize the information gain of each playtest. This could potentially be further improved by automated playtesting of the sort described above—designers can avoid playtesting unbeatable games or games unbeatable within a time limit or under some other constraint.

It is worth further exploring the set of ways in which a playtest could be derailed by a (design or programming) bug—and how to fix such issues automatically when they occur, as in the Zenet fault-recovery project [151]. Game makers could build playtest facilitation systems which notice and intervene when a player gets stuck (either because they fail to see a solution or the solution has become unachievable). This is a somewhat different question from reachability and is perhaps more useful and interesting. The opposite question—sequence breaking, where the player follows a supposedly impossible or unexpected sequence of actions—is also potentially interesting to designers.

Player modeling work could also be applied here. Considering what knowledge the player has access to (given what the game has displayed to them so far) and what knowledge the game requires to progress, a system could form a model of what the player actually does know. This could form the basis for in-game assistance and adaptive difficulty, and it could also be valuable for ensuring that playtests do not go too far afield from the questions designers want to answer. The goal of a human playtest is to debug the connection between the designer's mental

model of a game and that of players. Designers are surprised when players don't perceive the game properly; how could someone design automated playtests to highlight these gaps when they occur?

Designers and players both achieve high-level goals by abstracting away low-level details of game rules. For example, a curious player might plan to bomb every wall in *Zelda,* ignoring the details of navigating to each wall-adjacent tile, pausing to collect more bombs, and so on. To some designers, the agglomeration of higher-level skills through play is the essential task of learning to play a game [136, 66]; this was recently reified by analogy to programming language theory [163]. Could designers annotate their game code or rules to facilitate the automatic discovery of such high-level actions and chains? Could AIs search with these representations and highlight new abstract skills that designers did not expect to exist (for example, infinite bomb-jumping in *Super Metroid*)?

Many successful game design support systems visualize the behavior of subsets of a game's dynamics so that designers can consider game systems independently. Spatiotemporal visualizations of observed play activity have long been used to support design decisions in the iterative game design process [93]. The combinatorial story game Ice-Bound was developed alongside a visualization which highlighted possible story-object interactions, showing at a glance which objects and story events needed more text to be authored to avoid repetition or error states [102]. Competitive game players will often set up play environments which control for or abstract away significant portions of a game's state space to experiment with and understand the game's design. Fighting game players build spreadsheets enumerating the possible *combo moves* of each character against each other character and their outcomes; sometimes

these are visualized in the style of motion planning, where each move is an arc through possibility space and combos become strings of such steps [248]. Could one gather data and generate visualizations like these automatically? How do design diagnostics like these generalize to other sorts of game systems?

This leads to a fascinating and productive question: What sorts of (relatively game-independent) visualizations and algorithmic support would be most useful to designers—and how can the academic community evaluate such design process interventions? If one hopes to support creativity by reducing the gap between a design intuition and its concrete, playable realization (or to shrink the probability that a promising design is abandoned because of combinatorial design space exploration problems) then one *must* study the space of possible interactions between game designers and design support technologies more closely and more broadly than the community has so far [181, 154].

16.2 Moving Forward

I belive that making progress on all these fronts is only a matter of further exploring and applying the concepts introduced in this dissertation, synthesizing them with results from computer vision and other fields. As I write this, my colleague Adam Summerville has managed to relax many of Mappy's assumptions and has captured character and tile animation cycles; in fact, he has been able to identify where a game entity transitions between being drawn as a sprite and being drawn as a tile (as in Mario's bricks or Zelda's pushable blocks). Separately, the hybrid systems group at UCSC has taken an interest in HyPED, translating its representation

of hybrid automata into an alternative formalism amenable to model-predictive control, and is seeing promising initial results on motion planning problems in action games *in general* thanks to the expressive knowledge representation.

To repeat my opening argument: The theory and applications presented in this dissertation are powerful and can drive research agendas in game design support, automated game design learning, and even game studies. The schema of finding formal analogues to game design concepts is fruitful and effective, and I hope that, with or without operational logics as an intellectual core, it becomes a commonly accepted way of exploring the possibilities of game design space.

Bibliography

[1] The spriter's resource, 2003. URL https://www.spriters-resource.com.

[2] FCEUX - The all in one NES/Famicom/Dendy Emulator, 2017. URL http://www.fceux.com.

[3] ROMHacking.net, 2017. URL https://www.romhacking.net/utilities/.

[4] The Video Game Atlas, 2017. URL http://www.vgmaps.com/.

[5] Eric Aaron, Franjo Ivančić, and Dimitris Metaxas. Hybrid system models of navigation strategies for games and animations. In *International Workshop on Hybrid Systems: Computation and Control*, pages 7–20. Springer, 2002.

[6] Ernest Adams and Joris Dormans. *Game Mechanics: Advanced Game Design*. New Riders, 2012. ISBN 978-0-321-82027-3.

[7] Philip Agre. *Computation and Human Experience*. Cambridge University Press, 1997. ISBN 978-0-521-38603-6.

[8] Ian Albert. Super mario bros. maps - ian-albert.com, 2017. URL http://ian-albert.com/games/super_mario_bros_maps/.

[9] et al Albin, K. Property specification language reference manual, 2003.

[10] Zeyad Abd Algfoor, Mohd Shahrizal Sunar, and Hoshang Kolivand. A comprehensive study on pathfinding techniques for robotics and video games. *International Journal of Computer Games Technology*, 2015:7, 2015.

[11] Rajeev Alur and Parthasarathy Madhusudan. Visibly pushdown languages. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 202–211. ACM, 2004.

[12] Rajeev Alur, Costas Courcoubetis, Thomas A Henzinger, and Pei-Hsin Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid systems*. Springer, 1993.

[13] Rajeev Alur, Radu Grosu, Insup Lee, and Oleg Sokolsky. Compositional refinement for hierarchical hybrid systems. In *International Workshop on Hybrid Systems: Computation and Control*, pages 33–48. Springer, 2001.

[14] Rajeev Alur, Salar Moarref, and Ufuk Topcu. Compositional synthesis with parametric reactive controllers. In *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control*, pages 215–224. ACM, 2016.

[15] Erik Andersen, Sumit Gulwani, and Zoran Popovic. A trace-based framework for analyzing and synthesizing educational progressions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 773–782. ACM, 2013.

[16] Thomas H Apperley. Genre and game studies: Toward a critical approach to video game genres. *Simulation & Gaming*, 37(1):6–23, 2006.

[17] Ronald C Arkin. Path planning for a vision-based autonomous robot. In *Cambridge Symposium_Intelligent Robotics Systems*, pages 240–250. International Society for Optics and Photonics, 1987.

[18] Stephan Arlt, Evren Ermis, Sergio Feo-Arenis, and Andreas Podelski. Verification of GUI applications: A black-box approach. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, number 8802 in Lecture Notes in Computer Science, pages 236–252. Springer Berlin Heidelberg, 2014. ISBN 978-3-662-45233-2 978-3-662-45234-9. URL http://link.springer.com/chapter/10.1007/978-3-662-45234-9_17.

[19] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. Testing telecoms software with quviq quickcheck. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*, ERLANG '06, pages 2–10, New York, NY, USA, 2006. ACM. ISBN 1-59593-490-1. doi: 10.1145/1159789.1159792. URL http://doi.acm.org/10.1145/1159789.1159792.

[20] Fatemeh Asadi, Massimiliano Di Penta, Giuliano Antoniol, and Yann-gaël Guéhéneuc. A heuristic-based approach to identify concepts in execution traces. In *Proceedings of the 14th European Conference on Software Maintenance and Reengineering*, pages 31–40, 2010. doi: 10.1109/CSMR.2010.17.

[21] The Video Game Atlas. The video game atlas - gb/gbc maps, 2017. URL http://vgmaps.com/Atlas/GB-GBC/index.htm.

[22] Anna Atramentov and Steven M LaValle. Efficient nearest neighbor searching for motion planning. In *Robotics and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on*, volume 1, pages 632–637. IEEE, 2002.

[23] Abdulbaki Aydin, Lucas Bang, and Tevfik Bultan. Automata-based model counting for string constraints. In *International Conference on Computer Aided Verification*, pages 255–272. Springer, 2015.

[24] Christel Baier. On algorithmic verification methods for probabilistic systems. *Universität Mannheim*, 1998.

[25] Jorge A Baier and Sheila A McIlraith. Planning with first-order temporally extended goals using heuristic search. In *Proceedings of the National Conference on Artificial Intelligence*, volume 21, page 788. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2006.

[26] Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. A decade of software model checking with SLAM. *Communications of the ACM*, 54(7):68–76, 2011.

[27] Bikramjit Banerjee, Gregory Kuhlmann, and Peter Stone. Value function transfer for general game playing. In *ICML workshop on Structural Knowledge Transfer for Machine Learning*, June 2006. URL `http://www.cs.utexas.edu/users/ai-lab/?ICML06-bikram`.

[28] Nikola Banovic, Tovi Grossman, Justin Matejka, and George Fitzmaurice. Waken: Reverse engineering usage information and interface structure from software videos. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*, UIST '12, pages 83–92. ACM, 2012. ISBN 978-1-4503-1580-7. doi: 10.1145/2380116.2380129. URL `http://doi.acm.org/10.1145/2380116.2380129`.

[29] Jiří Barnat, Luboš Brim, Vojtěch Havel, Jan Havlíček, Jan Kriho, Milan Lenčo, Petr Ročkai, Vladimír Štill, and Jiří Weiser. Divine 3.0–an explicit-state model checker for multithreaded c & c++ programs. In *Computer Aided Verification*, pages 863–868. Springer, 2013.

[30] Clark W Barrett, Roberto Sebastiani, Sanjit A Seshia, and Cesare Tinelli. Satisfiability modulo theories. *Handbook of satisfiability*, 185:825–885, 2009.

[31] Richard Bartle. Hearts, clubs, diamonds, spades: Players who suit muds. *Journal of MUD research*, 1(1):19, 1996.

[32] Aaron William Bauer and Zoran Popović. Rrt-based game level analysis, visualization, and visual refinement. In *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2012.

[33] Leonard E Baum and Ted Petrie. Statistical inference for probabilistic functions of finite state markov chains. *The annals of mathematical statistics*, 1966.

[34] Matthew J Beal, Zoubin Ghahramani, and Carl Edward Rasmussen. The infinite hidden markov model. *Advances in neural information processing systems*, 2002.

[35] Marc Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[36] Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, pages 253–279, 2012. doi: 10.1613/jair.3912.

[37] Richard Bellman and Robert Roth. Curve fitting by segmented straight lines. *Journal of the American Statistical Association*, 1969.

[38] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[39] Alexis Bès. An application of the feferman-vaught theorem to automata and logics for words over an infinite alphabet. *Logical Methods in Computer Science*, 4, 2008.

[40] Ted J Biggerstaff. Design recovery for maintenance and reuse. *Computer*, 22(7):36–49, 1989.

[41] Staffan Björk and Jussi Holopainen. *Patterns in Game Design*. Charles River Media, 2004.

[42] Yngvi Björnsson. Learning rules of simplified boardgames by observing. In *Proceedings of the 20th European Conference on Artificial Intelligence*, pages 175–180. IOS Press, 2012.

[43] Ian Bogost. *Persuasive games: The expressive power of videogames*. Mit Press, 2007.

[44] SRK Branavan, David Silver, and Regina Barzilay. Non-linear monte-carlo search in civilization ii. In *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.

[45] Michael S Branicky, Michael M Curtiss, Joshua A Levine, and Stuart B Morgan. Rrts for nonlinear, discrete, and hybrid planning and control. In *Decision and Control, 2003. Proceedings. 42nd IEEE Conference on*, volume 1, pages 657–663. IEEE, 2003.

[46] Alexander Eric Braylan and Risto Miikkulainen. Object-model transfer in the general video game domain. In *Twelfth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2016.

[47] Will Bridewell, Pat Langley, Ljupčo Todorovski, and Sašo Džeroski. Inductive process modeling. *Machine learning*, 2008.

[48] Cameron Browne. *Evolutionary Game Design*. Springer Science & Business Media, 2011. ISBN 978-1-4471-2179-4.

[49] Stéphane Bura. Emotion engineering in videogames, 2008. URL `http://www.stephanebura.com/emotion/`.

[50] Eric Butler, Adam M Smith, Yun-En Liu, and Zoran Popovic. A mixed-initiative tool for designing level progressions in games. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*, pages 377–386. ACM, 2013.

[51] Rogelio E. Cardona-Rivera. Cognitively-grounded procedural content generation. In *Proc. What's Next for AI in Games*, 2017.

[52] Bert Chang and Paul du Bois. Robotic testing to the rescue. In *Game Developers Conference*, 2009.

[53] Peng Cheng and Steven M LaValle. Resolution complete rapidly-exploring random trees. In *Robotics and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on*, volume 1, pages 267–272. IEEE, 2002.

[54] Elliot J. Chikofsky and James H Cross. Reverse engineering and design recovery: A taxonomy. *IEEE software*, 7(1):13–17, 1990.

[55] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. *ACM SIGPLAN Notices*, 46(3):265–278, 2011.

[56] Doug Church. Formal abstract design tools. In Katie Salen Tekinbas and Eric Zimmerman, editors, *The Game Design Reader: A Rules of Play Anthology*. MIT Press, 2005.

[57] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. Nusmv: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[58] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 268–279. ACM, 2000. ISBN 1-58113-202-6. doi: 10.1145/351240.351266. URL `http://doi.acm.org/10.1145/351240.351266`.

[59] Clang. Clang static analyzer, 2008. URL `http://clang-analyzer.llvm.org/`.

[60] Edmund Clarke and Daniel Kroening. Hardware verification using ANSI-c programs as a reference. In *Proceedings of ASP-DAC 2003*, pages 308–311. IEEE Computer Society Press, 2003. ISBN 0-7803-7659-5.

[61] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and Aravinda Prasad Sistla, editors, *Computer Aided Verification*, number 1855 in Lecture Notes in Computer Science, pages 154–169. Springer Berlin Heidelberg, 2000. ISBN 978-3-540-67770-3 978-3-540-45047-4. URL `http://link.springer.com/chapter/10.1007/10722167_15`.

[62] James Clune. Heuristic evaluation functions for general game playing. In *Proceedings of the 22Nd National Conference on Artificial Intelligence - Volume 2*, AAAI'07, pages 1134–1139. AAAI Press, 2007. ISBN 978-1-57735-323-2. URL `http://dl.acm.org/citation.cfm?id=1619797.1619828`.

[63] Kate Compton. So you want to build a generator... http://galaxykate0.tumblr.com/post/139774965871/so-you-want-to-build-a-generator, February 2016.

[64] Kate Compton, Ben Kybartas, and Michael Mateas. Tracery: an author-focused generative text tool. In *International Conference on Interactive Digital Storytelling*, pages 154–161. Springer, 2015.

[65] Daniel Cook. The chemistry of game design. *Gamasutra*, 2007.

[66] Daniel Cook. Loops and arcs, 2012. URL http://www.lostgarden.com/2012/04/loops-and-arcs.html.

[67] Michael Cook. Alien languages: How we talk about procedural generation. *Gamasutra*, 2016.

[68] Michael Cook and Simon Colton. Multi-faceted evolution of simple arcade games. In *2011 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 289–296, 2011. doi: 10.1109/CIG.2011.6032019.

[69] Michael Cook and Simon Colton. From mechanics to meaning and back again: Exploring techniques for the contextualisation of code. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2013. URL http://www.aaai.org/ocs/index.php/AIIDE/AIIDE13/paper/view/7420.

[70] Michael Cook, Simon Colton, and Jeremy Gow. Initial results from co-operative co-evolution for automated platformer design. In Cecilia Di Chio, Alexandros Agapitos, Stefano Cagnoni, Carlos Cotta, Francisco Fernández de Vega, Gianni A. Di Caro, Rolf Drechsler, Anikó Ekárt, Anna I. Esparcia-Alcázar, Muddassar Farooq, William B. Langdon, Juan J. Merelo-Guervós, Mike Preuss, Hendrik Richter, Sara Silva, Anabela Simões, Giovanni Squillero, Ernesto Tarantino, Andrea G. B. Tettamanzi, Julian Togelius, Neil Urquhart, A. Şima Uyar, and Georgios N. Yannakakis, editors, *Applications of Evolutionary Computation*, number 7248 in Lecture Notes in Computer Science, pages 194–203. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-29177-7 978-3-642-29178-4. URL http://link.springer.com/chapter/10.1007/978-3-642-29178-4_20.

[71] Michael Cook, Simon Colton, and Alison Pease. Aesthetic considerations for automated platformer design. In *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2012. URL http://www.aaai.org/ocs/index.php/AIIDE/AIIDE12/paper/view/5463.

[72] Michael Cook, Simon Colton, Azalea Raad, and Jeremy Gow. Mechanic miner: Reflection-driven game mechanic discovery and level design. In Anna I. Esparcia-Alcázar, editor, *Applications of Evolutionary Computation*, number 7835 in Lecture Notes in Computer Science, pages 284–293. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-37191-2 978-3-642-37192-9. URL http://link.springer.com/chapter/10.1007/978-3-642-37192-9_29.

[73] Michael Cook, Simon Colton, and Jeremy Gow. Automating game design in three dimensions. In *ccg.doc.gold.ac.uk*, 2014.

[74] Michael Cook, Jeremy Gow, and Simon Colton. Danesh: Helping bridge the gap between procedural generators and their output. In *Proc. Procedural Content Generation*, 2016.

[75] Greg Costikyan. I have no words and i must design. interactive fantasy# 2. *British roleplaying journal*, 1994.

[76] Costas Courcoubetis, Moshe Vardi, Pierre Wolper, and Mihalis Yannakakis. Memory-efficient algorithms for the verification of temporal properties. In *Computer-aided Verification*, pages 129–142. Springer, 1993.

[77] Ben Cousins. Elementary game design. *Develop magazine*, 2004.

[78] Russ Cox. Implementing regular expressions. https://swtch.com/ rsc/regexp/, 2011.

[79] Chris Crawford. *The Art of Computer Game Design*. McGraw-Hill/Osborne Media, 1984.

[80] Valentin Dallmeier, Nikolai Knopp, Christoph Mallon, Sebastian Hack, and Andreas Zeller. Generating test cases for specification mining. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 85–96. ACM, 2010.

[81] Loris D'Antoni. A symbolic automata library. https://github.com/lorisdanto/symbolicautomata, 2017.

[82] Loris D'Antoni and Rajeev Alur. Symbolic visibly pushdown automata. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 209–225. Springer, 2014. ISBN 978-3-319-08866-2. doi: 10.1007/978-3-319-08867-9_14. URL https://doi.org/10.1007/978-3-319-08867-9_14.

[83] Loris D'Antoni and Margus Veanes. Static analysis of string encoders and decoders. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 209–228. Springer, 2013.

[84] Arthur P Dempster, Nan M Laird, and Donald B Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the royal statistical society. Series B (methodological)*, 1977.

[85] The Specs Project Developers. Specs - parallel ecs, 2018. URL https://github.com/slide-rs/specs.

[86] Bruno Dias. Procedural meaning: Pragmatic procgen in Voyageur. *Gamasutra*, 2016.

[87] Bruno Dias. Procedural generation in voyageur. https://vimeo.com/182465861, September 2016.

[88] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25: 53–95, 2013. doi: 10.1002/smr.567.

[89] Eurico Doirado and Carlos Martinho. I mean it!: detecting user intentions to create believable behaviour for virtual agents in games. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*, pages 83–90. International Foundation for Autonomous Agents and Multiagent Systems, 2010.

[90] J Dormans. Machinations: Elemental feedback structures for game design. In *Proceedings of the GAMEON-NA Conference*, pages 33–40, 2009.

[91] Joris Dormans. Simulating mechanics to study emergence in games. In *Workshops at the Seventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2011. URL `http://www.aaai.org/ocs/index.php/AIIDE/AIIDE11WS/paper/view/4093`.

[92] Alexandre Duret-Lutz and Denis Poitrenaud. Spot: an extensible model checking library using transition-based generalized büchi automata. In *Proceedings of the IEEE Computer Society's 12th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, pages 76–83. IEEE, 2004.

[93] Magy Seif El-Nasr, Anders Drachen, and Alessandro Canossa. *Game analytics: Maximizing the value of player data*. Springer Science & Business Media, 2013.

[94] Zelda Elements. Link's awakening maps | zelda elements, 2017. URL `http://www.zeldaelements.net/games/c/links_awakening/maps`.

[95] Roger Evans and Emily Short. Versu—a simulationist storytelling system. *Computational Intelligence and AI in Games*, 2014.

[96] N Falstein. Better by design: The 400 project. *Game Developer magazine*, 9(3), 2002. URL `http://www.theinspiracy.com/400_project.htm`.

[97] Martin Fasterholdt, Martin Pichlmair, and Christoffer Holmgård. You say jump, i say how high? operationalising the game feel of jumping. In *DiGRA/FDG*, 2016.

[98] Daniel Fava, Dan Shapiro, Joseph Osborn, Martin Schaef, and E James Whitehead. Crowdsourcing program preconditions via a classification game. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 1086–1096. IEEE, 2016.

[99] Jordan Fisher. How to make insane, procedural platformer levels. *Gamasutra*, 2012.

[100] Frame Trapped Software Group. Hit Box Viewer, 2017. URL `http://frametrapped.com/`.

[101] Ted Friedman. The semiotics of simcity. *First Monday*, 4(4), 1999.

[102] Jacob Garbe, Aaron A Reed, Melanie Dickinson, Noah Wardrip-Fruin, and Michael Mateas. Author assistance visualizations for ice-bound, a combinatorial narrative. In *Proceedings of the Ninth International Conference on the Foundations of Digital Games*, 2014.

[103] N. E. Gold, M. Harman, D. Binkley, and R. M. Hierons. Unifying program slicing and concept assignment for higher-level executable source code extraction. *Software: Practice and Experience*, 35:977–1006, 2005. doi: 10.1002/spe.664.

[104] Nicolas Gold. Hypothesis-based concept assignment to support software maintenance. In *2013 IEEE International Conference on Software Maintenance*, volume 0, page 545. IEEE Computer Society, 2001. doi: 10.1109/ICSM.2001.972768.

[105] David Graham. In-game debugging and visualization tools. In *Game Developers Conference*, 2012.

[106] Daniele Gravina, Antonios Liapis, and Georgios Yannakakis. Surprise search: Beyond objectives and novelty. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference*, pages 677–684. ACM, 2016.

[107] Peter Gregory, Henrique Coli Schumann, Yngvi Björnsson, and Stephan Schiffel. The grl system: learning board game rules with piece-move interactions. In *Workshop on Computer Games*, pages 130–148. Springer, 2015.

[108] Matthew Guzdial. Video parser, 2017. URL `https://github.com/mguzdial3/VideoParser`.

[109] Matthew Guzdial and Mark Riedl. Game level generation from gameplay videos. In *Twelfth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2016.

[110] Matthew Guzdial and Mark O. Riedl. Toward game level generation from gameplay videos. In *Proceedings of the Sixth Workshop on Procedural Content Generation*, 2015.

[111] Matthew Guzdial, Boyang Li, and Mark Riedl. Game engine learning from video. In *26th International Joint Conference on Artificial Intelligence*, 2017.

[112] Kent Hansen. Metroid level data explained, 2017. URL `http://www.metroid-database.com/m1/lvldata.php`.

[113] Mark Harman, Nicolas Gold, Rob Hierons, and Dave Binkley. Code extraction algorithms which unify slicing and concept assignment. In *Ninth Working Conference on Reverse Engineering*, pages 11–20, 2002. doi: 10.1109/WCRE.2002.1173060.

[114] Thomas A Henzinger. The theory of hybrid automata. In *Verification of Digital and Hybrid Systems*, pages 265–292. Springer, 2000.

[115] Thomas R. Hinrichs and Kenneth D. Forbus. Analogical learning in a turn-based strategy game. In *International Joint Conference on Artificial Intelligence*, pages 1–6, 2006.

[116] Xavier Ho, Martin Tomitsch, and Tomasz Bednarz. Finding design influence within roguelike games. *Meaningful Play 2016 Conference Proceedings*, 2016.

[117] Gerard J Holzmann. Design and validation of protocols: a tutorial. *Computer Networks and ISDN Systems*, 25(9):981–1017, 1993.

[118] Gerard J Holzmann. The model checker spin. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.

[119] Ian D Horswill. Architectural issues for compositional dialog in games. In *Proc. GAMNLP*, 2014.

[120] William H. Huber. Epic spatialities: The production of space in the final fantasy games. In Pat Harrigan and Noah Wardrip-Fruin, editors, *Third Person: Authoring and Exploring Vast Narratives*. MIT Press, 2009.

[121] John Hughes, Ulf Norell, and Jérôme Sautret. Using temporal relations to specify and test an instant messaging server. In *Proceedings of the 5th Workshop on Automation of Software Testing*, pages 95–102. ACM, 2010.

[122] Robin Hunicke, Marc LeBlanc, and Robert Zubek. Mda: A formal approach to game design and game research. In *Proceedings of the AAAI Workshop on Challenges in Game AI*, 2004.

[123] Aaron Isaksen, Daniel Gopstein, and Andrew Nealen. Exploring game space using survival analysis. In *FDG*, 2015.

[124] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[125] Alexander Jaffe, Alex Miller, Erik Andersen, Yun-En Liu, Anna Karlin, and Zoran Popovic. Evaluating competitive game balance with restricted play. In *Proceedings of the Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2012.

[126] Léonard Jaillet, Judy Hoffman, Jur Van den Berg, Pieter Abbeel, Josep M Porta, and Ken Goldberg. Eg-rrt: Environment-guided random trees for kinodynamic motion planning with uncertainty and obstacles. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pages 2646–2652. IEEE, 2011.

[127] Ryan James, Kaltman Eric, Hong Timothy, Isbister Katherine, Mateas Michael, and Wardrip-Fruin Noah. Gamenet and gamesage: Videogame discovery as design insight. In *DiGRA/FDG*, Dundee, Scotland, August 2016. Digital Games Research Association and Society for the Advancement of the Science of Digital Games. ISBN ISSN 2342-9666. URL `http://www.digra.org/wp-content/uploads/digital-library/paper_201.pdf`.

[128] jdaster64. SMB physics spec. http://forums.mfgg.net/viewtopic.php?p=346301, 2012. Accessed: 2017-02-13.

[129] Jesper Juul. The open and the closed: Games of emergence and games of progression. In *CGDC Conf.*, 2002.

[130] Jesper Juul. *Half-real: Video games between real rules and fictional worlds*. MIT press, 2005.

[131] Jesper Juul. Swap adjacent gems to make sets of three: A history of matching tile games. *Artifact*, 1(4):205–216, 2007.

[132] Daniel Kästner, Jörg Barrho, Ulrich Wünsche, Marc Schlickling, Bernhard Schommer, Michael Schmidt, Christian Ferdinand, Xavier Leroy, and Sandrine Blazy. CompCert: Practical Experience on Integrating and Qualifying a Formally Verified Optimizing Compiler. In *ERTS2 2018 - Embedded Real Time Software and Systems*, Toulouse, France, January 2018. 3AF, SEE, SIE. URL `https://hal.inria.fr/hal-01643290`.

[133] Jongwoo Kim and Joel M Esposito. Adaptive sample bias for rapidly-exploring random trees with applications to test generation. In *American Control Conference, 2005. Proceedings of the 2005*, pages 1166–1172. IEEE, 2005.

[134] Kenneth Kolson. The politics of simcity. *PS: Political Science & Politics*, 29(1):43–46, 1996.

[135] Tolga Könik, Paul O'Rorke, Dan Shapiro, Dongkyu Choi, Negin Nejati, and Pat Langley. Skill transfer through goal-driven representation mapping. *Cognitive Systems Research*, 10:270–285, 2009. doi: 10.1016/j.cogsys.2008.09.008.

[136] Raph Koster. A grammar of gameplay. In *Game Developers Conference*, 2005. URL `http://www.raphkoster.com/gaming/atof/grammarofgameplay.pdf`.

[137] Raph Koster. *Theory of Fun for Game Design*. O'Reilly Media, Inc., 2013. ISBN 978-1-4493-6319-2.

[138] Bartosz Kostka, Jaroslaw Kwiecien, Jakub Kowalski, and Pawel Rychlikowski. Text-based adventures of the golovin ai agent, 2017.

[139] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-time systems*, 2(4):255–299, 1990.

[140] Fabian Kratz, Oleg Sokolsky, George J Pappas, and Insup Lee. R-charon, a modeling language for reconfigurable hybrid systems. In *International Workshop on Hybrid Systems: Computation and Control*, pages 392–406. Springer, 2006.

[141] Fred Kröger and Stephan Merz. First-order linear temporal logic. In *Temporal Logic and State Systems*, pages 153–179. Springer, 2008.

[142] Gregory Kuhlmann and Peter Stone. Automatic heuristic construction in a complete general game player. In *AAAI*, volume 6, pages 1457–1462, 2006.

[143] Sunil L Kukreja, Robert E Kearney, and Henrietta L Galiana. A least-squares parameter estimation algorithm for switched hammerstein systems with applications to the vor. *IEEE Transactions on Biomedical Engineering*, 2005.

[144] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN 0-321-14306-X.

[145] Frank Lantz, Aaron Isaksen, Alexander Jaffe, Andy Nealen, and Julian Togelius. Depth in strategic games. In *AAAI Workshop on What's Next for AI in Games*, 2017.

[146] Steven M. Lavalle. Rapidly-exploring random trees: A new tool for path planning. Technical report, Iowa State University, 1998.

[147] S. Lavelle. Puzzlescript. http://puzzlescript.net, 2013.

[148] Linda Lemieux, Brian Buhr, Michael Souto, Marietta Pashayan, Dyan Jodoin, and Steve Fowler. Automation in the technically challenging world of game development. In *Game Developers Conference*, 2013.

[149] Jonathan Lessard. Designing natural-language game conversations. In *Proceedings of the First International Joint Conference of DiGRA and FDG*. Digital Games Research Association and Society for the Advancement of the Science of Digital Games, 2016.

[150] Jonathan Lessard, Etienne Brunelle-Leclerc, Timothy Gottschalk, Marc-Antoine Jetté-Léger, Odile Prouveur, and Christopher Tan. Striving for author-friendly procedural dialogue generation. In *Proc. Non-Player Characters and Social Believability in Games*, 2017.

[151] Chris Lewis. Zenet: Generating and enforcing real-time temporal invariants. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pages 329–330. ACM, 2010.

[152] Chris Lewis and Jim Whitehead. *Runtime repair of software faults using event-driven monitoring*, volume 2. ACM, 2010. ISBN 978-1-60558-719-6.

[153] Mike Lewis. Stochastic grammars: Not just for words! In Steve Rabin, editor, *Game AI Pro 3*. CRC Press, 2017.

[154] Antonios Liapis, Georgios N Yannakakis, and Julian Togelius. Designer modeling for personalized game content creation tools. In *Proceedings of the AIIDE workshop on artificial intelligence & game aesthetics*, 2013.

[155] Antonios Liapis, Georgios N Yannakakis, and Julian Togelius. Sentient sketchbook: Computer-aided game level authoring. In *Proceedings of the Eighth International Conference on the Foundations of Digital Games*, pages 213–220, 2013.

[156] Vladimir Lifschitz and Wanwan Ren. A modular action description language. In *AAAI*, volume 6, pages 853–859, 2006.

[157] Chong-u Lim and D.F. Harrell. An approach to general videogame evaluation and automatic generation using a description language. In *2014 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8, 2014. doi: 10.1109/CIG.2014.6932896.

[158] Yun-En Liu, Erik Andersen, Richard Snider, Seth Cooper, and Zoran Popović. Feature-based projections for effective playtrace analysis. In *Proceedings of the Sixth International Conference on the Foundations of Digital Games*, pages 69–76. ACM, 2011.

[159] Noel Llopis. Data oriented design: Now and in the future. *Game Developers Magazine*, 17(8):31–33, 2010.

[160] Daniel L Ly and Hod Lipson. Learning symbolic representations of hybrid dynamical systems. *Journal of Machine Learning Research*, 13(Dec):3585–3618, 2012.

[161] Steve Mann. Compositing multiple pictures of the same scene. In *Proceedings of the 46th Annual IS&T Conference*, volume 2, pages 319–25, 1993.

[162] Chris Martens. Ceptre: A language for modeling generative interactive systems. In *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2015.

[163] Chris Martens and Matthew A. Hammer. Languages of play: Towards semantic foundations for game interfaces. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, 2017.

[164] Chris Martens, Adam Summerville, Michael Mateas, Joseph Osborn, Sarah Harmon, Noah Wardrip-Fruin, and Arnav Jhala. Proceduralist readings, procedurally. In *Experimental AI in Games Workshop*, volume 3, 2016.

[165] Michael Mateas. Expressive ai: A semiotic analysis of machinic affordances. In *3rd Conference on Computational Semiotics for Games and New Media*, page 58, 2003.

[166] Michael Mateas and Noah Wardrip-Fruin. Defining operational logics. *Digital Games Research Association (DiGRA)*, 4, 2009.

[167] Michael L Mauldin, Guy Jacobson, Andrew W Appel, and Leonard GC Hamey. Rog-o-matic: a belligerent expert system. Technical report, Carnegie-Mellon University, 1983.

[168] Peter Andrew Mawhorter. *Artificial Intelligence as a Tool for Understanding Narrative Choices*. PhD thesis, University of California, Santa Cruz, 2016.

[169] Joshua McCoy, Mike Treanor, Ben Samuel, Aaron A Reed, Michael Mateas, and Noah Wardrip-Fruin. Prom week: Designing past the game/story dilemma. In *FDG*, pages 94–101, 2013.

[170] Joshua McCoy et al. Social story worlds with Comme il Faut. *Computational Intelligence and AI in Games*, 2014.

[171] Soumaya Medini, Philippe Galinier, Massimiliano Di Penta, Yann-gaël Guéhéneuc, and Giuliano Antoniol. A fast algorithm to locate concepts in execution traces. In *Lecture Notes in Computer Science*, volume 6956, pages 252–266. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-23715-7.

[172] Florian Merz, Stephan Falke, and Carsten Sinz. LLBMC: Bounded model checking of c and c++ programs using a compiler IR. In Rajeev Joshi, Peter Müller, and Andreas Podelski, editors, *Verified Software: Theories, Tools, Experiments*, number 7152 in Lecture Notes in Computer Science, pages 146–161. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-27704-7 978-3-642-27705-4. URL http://link.springer.com/chapter/10.1007/978-3-642-27705-4_12.

[173] Kieran Milan, Joel Veness, James Kirkpatrick, Michael Bowling, Anna Koop, and Demis Hassabis. The forget-me-not process. In *Advances in Neural Information Processing Systems*, 2016.

[174] Sergey Mohov. Turning a chatbot into a narrative game: Language interaction in Event[0]. In *nucl.ai*, 2015.

[175] Casey Muratori. Mapping the Island's Walkable Surfaces, 2012. URL http://the-witness.net/news/2012/12/mapping-the-islands-walkable-surfaces/.

[176] Tom Murphy, VII. learnfun & playfun: A general technique for automating nes games. *SIGBOVIK*, April 2013.

[177] Tom Murphy, VII. The glEnd() of Zelda. *SIGBOVIK*, April 2016.

[178] Graham Nelson. Natural language, semantic analysis, and interactive fiction. *IF Theory Reader*, page 141, 2006.

[179] Mark J Nelson. *Representing and reasoning about videogame mechanics for automated design support*. PhD thesis, Georgia Institute of Technology, 2015.

[180] Mark J Nelson and Michael Mateas. An interactive game-design assistant. In *Proceedings of the 13th international conference on Intelligent user interfaces*, pages 90–98. ACM, 2008.

[181] Mark J. Nelson and Michael Mateas. A requirements analysis for videogame design support tools. In *Proceedings of the 4th International Conference on Foundations of Digital Games*. ACM, 2009. ISBN 978-1-60558-437-9.

[182] Nelson, Graham. §1.8: A short skein tutorial, 2014. URL http://inform7.com/learn/man/WI_1_8.html.

[183] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How amazon web services uses formal methods. *Communications of the ACM*, 58, 2015. doi: 10.1145/2749359.2699417.

[184] Oliver Niggemann, Benno Stein, Asmir Vodencarevic, Alexander Maier, and Hans Kleine Büning. Learning behavior models for hybrid timed systems. In *AAAI*, 2012.

[185] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[186] Peter Ohmann, Alexander Brooks, Loris D'Antoni, and Ben Liblit. Control-flow recovery from partial failure reports. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, Barcelona, Spain*, 2017.

[187] Joseph Osborn, Ben Samuel, Adam Summerville, and Michael Mateas. Towards general rpg playing. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2017. URL `https://aaai.org/ocs/index.php/AIIDE/AIIDE17/paper/view/15899`.

[188] Joseph C. Osborn. Operational logics catalog, 2017. URL `https://github.com/JoeOsborn/ol-catalog`.

[189] Joseph C. Osborn. Reductionist. https://github.com/joeosborn/reductionist, 2017.

[190] Joseph C Osborn and Michael Mateas. A game-independent play trace dissimilarity metric. *Proceedings of the Ninth International Conference on the Foundations of Digital Games*, 2014.

[191] Joseph C. Osborn and Michael Mateas. Evaluating a solver-aided puzzle design tool. In Sebastian Deterding, Jonathan Hook, Rebecca Fiebrink, Marco Gillies, Jeremy Gow, Gillian Smith, Kate Compton, and Memo Akten, editors, *Proceedings of the CHI'17 Workshop on Mixed-Initiative Creative Interfaces co-located with ACM CHI Conference on Human Factors in Computing Systems (CHI 2017), Denver, USA, May 7, 2017.*, volume 1907 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2017. URL `http://ceur-ws.org/Vol-1907/3_mici_osborn.pdf`.

[192] Joseph C. Osborn, Dylan Lederle-Ensign, Noah Wardrip-Fruin, and Michael Mateas. Combat in games. In *Proceedings of the Tenth International Conference on the Foundations of Digital Games*, 2015.

[193] Joseph C Osborn, Ben Samuel, Michael Mateas, and Noah Wardrip-Fruin. Playspecs: Regular expressions for game play traces. In *11th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2015.

[194] Joseph C. Osborn, Brian Lambrigger, and Michael Mateas. HyPED: Modeling and analyzing action games as hybrid systems. In *Thirteenth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2017.

[195] Joseph C. Osborn, James Ryan, and Michael Mateas. Analyzing expressionist grammars by reduction to symbolic visibly pushdown automata. In *Proceedings of the Tenth International Workshop on Intelligent Narrative Technologies*, 2017.

[196] Joseph C. Osborn, Adam Summerville, and Michael Mateas. Automated game design learning. In *Proceedings of the Conference on Computational Intelligence in Games*, 2017.

[197] Joseph C. Osborn, Adam Summerville, and Michael Mateas. Automatic mapping of nes games with mappy. In *Proceedings of the 2017 Workshop on Procedural Content Generation*, 2017.

[198] Joseph C. Osborn, Noah Wardrip-Fruin, and Michael Mateas. Refining operational logics. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, 2017.

[199] Joseph Carter Osborn, April Grow, and Michael Mateas. Modular computational critics for games. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2013. URL `http://www.aaai.org/ocs/index.php/AIIDE/AIIDE13/paper/view/7389`.

[200] Joseph Carter Osborn, Benjamin Samuel, and Michael Mateas. Visualizing the strategic landscape of arbitrary games. *Information Visualization*, page 1473871617718377, 2017.

[201] pannenkoek2012. SM64 - Watch for Rolling Rocks - 0.5x A Presses (Commentated), 2016. URL `https://www.youtube.com/watch?v=kpk2tdsPh0A`.

[202] pannenkoek2012. SM64 - Watch for Rolling Rocks - 0.5x A Presses, 2016. URL `http://imgur.com/gallery/rK24p`.

[203] Deepak Pathak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In *ICML*, 2017.

[204] Diego Pérez-Liébana, Spyridon Samothrakis, Julian Togelius, Tom Schaul, and Simon M Lucas. Analyzing the robustness of general video game playing agents. In *Computational Intelligence and Games (CIG), 2016 IEEE Conference on*, pages 1–8. IEEE, 2016.

[205] Diego Perez-Liebana, Spyridon Samothrakis, Julian Togelius, Tom Schaul, and Simon M Lucas. General video game ai: Competition, challenges and opportunities. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[206] Diego Perez-Liebana, Spyridon Samothrakis, Julian Togelius, Tom Schaul, Simon M Lucas, Adrien Couëtoux, Jerry Lee, Chong-U Lim, and Tommy Thompson. The 2014 general video game playing competition. *IEEE Transactions on Computational Intelligence and AI in Games*, 8(3):229–243, 2016.

[207] Brian Provinciano. Automated testing and instant replays in retro city rampage. In *Game Developers Conference*, 2015.

[208] James Ryan, Tyler Brothers, Michael Mateas, and Noah Wardrip-Fruin. Juke joint: characters who are moved by music. *Proc. Experimental AI in Games*, 2016.

[209] James Ryan, Michael Mateas, and Noah Wardrip-Fruin. Characters who speak their minds: Dialogue generation in talk of the town. *Proc. AIIDE*, 2016.

[210] James Ryan, Ethan Seither, Michael Mateas, and Noah Wardrip-Fruin. Expressionist: An authoring tool for in-game text generation. In *Proc. ICIDS*, 2016.

[211] Katie Salen and Eric Zimmerman. *Rules of play: Game design fundamentals*. MIT press, 2004.

[212] Ben Samuel, Aaron A. Reed, Paul Maddaloni, Michael Mateas, and Noah Wardrip-Fruin. The scored rule engine: Next-generation social physics. In *Proceedings of the 10th International Conference on the Foundations of Digital Games*, 2015.

[213] Pedro Santana, Spencer Lane, Eric Timmons, Brian Williams, and Carlos Forster. Learning hybrid models with guarded transitions. In *AAAI Conference on Artificial Intelligence*, 2015.

[214] T. Schaul. A video game description language for model-based or interactive learning. In *2013 IEEE Conference on Computational Intelligence in Games (CIG)*, pages 1–8, 2013. doi: 10.1109/CIG.2013.6633610.

[215] Stephan Schiffel and Michael Thielscher. Fluxplayer: A successful general game player. In *AAAI*, volume 7, pages 1191–1196, 2007.

[216] Gideon Schwarz et al. Estimating the dimension of a model. *The annals of statistics*, 1978.

[217] Mohammad Shaker, Mhd Hasan Sarhan, Ola Al Naameh, Noor Shaker, and Julian Togelius. Automatic generation and analysis of physics-based puzzle games. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, pages 1–8. IEEE, 2013.

[218] Mohammad Shaker, Noor Shaker, and Julian Togelius. Ropossum: An authoring tool for designing, optimizing and solving cut the rope levels. In *Proceedings of the Ninth Aaai Conference on Artificial Intelligence and Interactive Digital Entertainment*. AAAI Press, 2013.

[219] Alexander Shkolnik, Matthew Walter, and Russ Tedrake. Reachability-guided sampling for planning under differential constraints. In *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, pages 2859–2865. IEEE, 2009.

[220] Emily Short. Procedural text generation in IF. https://emshort.wordpress.com/2014/11/18/procedural-text-generation-in-if/, November 2014.

[221] Emily Short. Bowls of oatmeal and text generation. https://emshort.blog/2016/09/21/bowls-of-oatmeal-and-text-generation/, September 2016.

[222] Emily Short. Visualizing procgen text. https://emshort.blog/2016/06/13/visualizing-procgen-text/, June 2016.

[223] Emily Short. Visualizing procgen text, part two. https://emshort.blog/2016/09/12/visualizing-procgen-text-part-two/, September 2016.

[224] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[225] Yasser Shoukry, Pierluigi Nuzzo, Indranil Saha, Alberto L Sangiovanni-Vincentelli, Sanjit A Seshia, George J Pappas, and Paulo Tabuada. Scalable lazy smt-based motion planning. In *Decision and Control (CDC), 2016 IEEE 55th Conference on*, pages 6683–6688. IEEE, 2016.

[226] Kristin Siu, Eric Butler, and Alexander Zook. A programming model for boss encounters in 2d action games. In *Experimental AI in Games Workshop*, 2016.

[227] Adam M Smith. Open problem: Reusable gameplay trace samplers. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2013.

[228] Adam M. Smith and Michael Mateas. Variations forever: flexibly generating rulesets from a sculptable design space of mini-games. In *In Proc. of the 2010 IEEE Conf. on Computational Intelligence and Games (CIG*, 2010.

[229] Adam M Smith, Mark J Nelson, and Michael Mateas. Computational support for play testing game sketches. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2009.

[230] Adam M. Smith, Mark J. Nelson, and Michael Mateas. Ludocore: A logical game engine for modeling videogames. In *Proceedings of IEEE Conference on Computational Intelligence and Games*, 2010.

[231] Adam M Smith, Chris Lewis, Kenneth Hullet, Gillian Smith, and Anne Sullivan. An inclusive view of player modeling. In *Proceedings of the 6th International Conference on Foundations of Digital Games*, pages 301–303. ACM, 2011.

[232] Adam M Smith, Erik Andersen, Michael Mateas, and Zoran Popović. A case study of expressively constrainable level design automation tools for a puzzle game. In *Proceedings of the Seventh International Conference on the Foundations of Digital Games*, pages 156–163. ACM, 2012.

[233] Adam M Smith, Eric Butler, and Zoran Popovic. Quantifying over play: Constraining undesirable solutions in puzzle design. In *FDG*, pages 221–228, 2013.

[234] Gillian Smith and Jim Whitehead. Analyzing the expressive range of a level generator. In *Proc. Procedural Content Generation in Games*, page 4, 2010.

[235] Gillian Smith, Jim Whitehead, and Michael Mateas. Tanagra: A mixed-initiative level design tool. In *Proceedings of the Fifth International Conference on the Foundations of Digital Games*, pages 209–216. ACM, 2010.

[236] Spirit AI. http://spiritai.com/, 2017.

[237] Christopher Stanton and Jeff Clune. Curiosity search: Producing generalists by encouraging individuals to continually explore and acquire skills throughout their lifetime. *PloS one*, 11(9):e0162235, 2016.

[238] Robert A Stine. Model selection using information theory and the mdl principle. *Sociological Methods & Research*, 2004.

[239] Adam Summerville, Matthew Guzdial, Michael Mateas, and Mark O Riedl. Learning player tailored content from observation: Platformer level generation from video traces using lstms. In *Twelfth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2016.

[240] Adam Summerville, Sam Snodgrass, Michael Mateas, and Santiago Ontanón. The vglc: The video game level corpus. *arXiv preprint arXiv:1606.07487*, 2016.

[241] Adam Summerville, Morteza Behrooz, Michael Mateas, and Arnav Jhala. What does that ?-block do? learning latent causal affordances from mario play traces. *Proceedings of the first What's Next for AI in Games Workshop at AAAI 2017*, 2017.

[242] Adam Summerville, Chris Martens, Sarah Harmon, Michael Mateas, Joseph Carter Osborn, Noah Wardrip-Fruin, and Arnav Jhala. From mechanics to meaning. *IEEE Transactions on Computational Intelligence and AI in Games*, 2017.

[243] Adam Summerville, Joseph C. Osborn, Christoffer Holmgård, Daniel Zhang, and Michael Mateas. Mechanics automatically recognized via interactive observation: Jumping. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, 2017.

[244] Adam Summerville, Joseph C. Osborn, and Michael Mateas. Charda: Causal hybrid automata recovery via dynamic analysis. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 2017.

[245] Adam Summerville, Sam Snodgrass, Matthew Guzdial, Christoffer Holmgård, Amy K. Hoover, Aaron Isaksen, Andy Nealen, and Julian Togelius. Procedural content generation via machine learning (PCGML). *CoRR*, abs/1702.00539, 2017. URL `http://arxiv.org/abs/1702.00539`.

[246] Adam James Summerville, James Ryan, Michael Mateas, and Noah Wardrip-Fruin. CFGs-2-NLU: Sequence-to-sequence learning for mapping utterances to semantics and pragmatics. Technical Report UCSC-SOE-16-11, UC Santa Cruz, 2016.

[247] Steve Swink. *Game feel: a game designer's guide to virtual sensation*. Morgan Kaufmann, 2009.

[248] Richard Terrell. Design over time: Smash fundamentals part 2, 2016. URL https://www.youtube.com/watch?v=TmAG_l7FHD4.

[249] Terrell, Richard. One Smash, 2016. URL http://www.onesmash.net/.

[250] Michael Thielscher. GDL-II. *Künstliche Intelligenz*, 25(1):63–66, 2010. ISSN 0933-1875, 1610-1987. doi: 10.1007/s13218-010-0076-5. URL http://link.springer.com/article/10.1007/s13218-010-0076-5.

[251] Michael Thielscher. The general game playing description language is universal. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, page 1107, 2011. URL http://www.cse.unsw.edu.au/~mit/Papers/IJCAI11.pdf.

[252] Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968. ISSN 0001-0782. doi: 10.1145/363347.363387. URL http://doi.acm.org/10.1145/363347.363387.

[253] David Thue, Vadim Bulitko, and Marcia Spetch. Passage: A demonstration of player modeling in interactive storytelling. In *AIIDE*, 2008.

[254] Nikolai Tillmann and Jonathan de Halleux. Pex - white box test generation for .NET. In Bernhard Beckert and Reiner Hähnle, editors, *Tests and Proofs*, volume 4966, pages 134–153. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-79123-2 978-3-540-79124-9. URL http://link.springer.com/10.1007/978-3-540-79124-9_10.

[255] Julian Togelius, Noor Shaker, and Joris Dormans. Grammars and l-systems with applications to vegetation and levels. In *Procedural Content Generation in Games*, pages 73–98. Springer, 2016.

[256] Paul Tozour. Decision modeling and optimization in game design, part 1: Introduction, 2013. URL http://www.gamasutra.com/blogs/PaulTozour/20130707/195718/Decision_Modeling_and_Optimization_in_Game_Design_Part_1_Introduction.php.

[257] Mike Treanor and Michael Mateas. BurgerTime: A proceduralist investigation. In *Proceedings of the 2011 DiGRA International Conference*, 2011. URL http://www.digra.org/wp-content/uploads/digital-library/11307.07106.pdf.

[258] Mike Treanor, Michael Mateas, and Noah Wardrip-Fruin. Kaboom! is a many-splendored thing: An interpretation and design methodology for message-driven games using graphical logics. In *Proceedings of the Fifth International Conference on the Foundations of Digital Games*, pages 224–231. ACM, 2010.

[259] Mike Treanor, Bobby Schweizer, Ian Bogost, and Michael Mateas. Proceduralist readings: How to find meaning in games with graphical logics. In *Proceedings of the 6th International Conference on Foundations of Digital Games*, pages 115–122. ACM, 2011.

[260] Mike Treanor, Bryan Blackford, Michael Mateas, and Ian Bogost. Game-o-matic: Generating videogames that represent ideas. In *Proceedings of the The Third Workshop on Procedural Content Generation in Games*, PCG'12, pages 11:1–11:8. ACM, 2012. ISBN 978-1-4503-1447-3. doi: 10.1145/2538528.2538537. URL http://doi.acm.org/10.1145/2538528.2538537.

[261] Mike Treanor, Bobby Schweizer, Ian Bogost, and Michael Mateas. The micro-rhetorics of game-o-matic. In *Proceedings of the International Conference on the Foundations of Digital Games*, pages 18–25. ACM, 2012.

[262] Jonathan Tremblay, Pedro Andrade Torres, Nir Rikovitch, and Clark Verbrugge. An exploration tool for predicting stealthy behaviour. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2013. URL http://www.aaai.org/ocs/index.php/AIIDE/AIIDE13/paper/view/7435.

[263] Jonathan Tremblay, Alexander Borodovski, and Clark Verbrugge. I can jump! exploring search algorithms for simulating platformer players. In *Proceedings of the Tenth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2014.

[264] RAM Van der Linden. *Designing procedurally generated levels*. PhD thesis, TU Delft, Delft University of Technology, 2013.

[265] Riemer Van Rozen and Joris Dormans. Adapting game mechanics with micro-machinations. In *Foundations of Digital Games*, Proceedings of the 9th International Conference on the Foundations of Digital Games. Society for the Advancement of the Science of Digital Games, 2014. URL https://hal.inria.fr/hal-01110847.

[266] Tom Murphy VII. T in y world, 2012. URL http://tinyworld.spacebar.org/.

[267] Greg Ward. Hiding seams in high dynamic range panoramas. In *Proceedings of the 3rd Symposium on Applied Perception in Graphics and Visualization*, pages 150–150. ACM, 2006.

[268] N. Wardrip-Fruin. Playable media and textual instruments. *Dichtung Digital*, 34:211–253, 2005.

[269] Noah Wardrip-Fruin. Expressive processing: On process-intensive literature and digital media, 2006.

[270] Noah Wardrip-Fruin, Michael Mateas, Steven Dow, and Serdar Sali. Agency reconsidered. *Breaking New Ground: Innovation in Games, Play, Practice and Theory. Proceedings of DiGRA 2009*, 2009.

[271] Stefan Winkler and Jens Pilgrim. A survey of traceability in requirements engineering and model-driven development. *Software and Systems Modeling*, 9(4):529–565, 2010. ISSN 1619-1366. doi: 10.1007/s10270-009-0145-0. URL http://dx.doi.org/10.1007/s10270-009-0145-0.

[272] Nick Yee. Motivations for play in online games. *CyberPsychology & behavior*, 9(6): 772–775, 2006.

[273] Fang Yu, Muath Alkhalaf, and Tevfik Bultan. Stranger: An automata-based string analysis tool for php. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 154–157. Springer, 2010.

[274] José P. Zagal and Amy Bruckman. The game ontology project: Supporting learning while contributing authentically to game studies. In *Proceedings of the 8th International Conference for the Learning Sciences*, ICLS'08, pages 499–506. International Society of the Learning Sciences, 2008. URL http://dl.acm.org/citation.cfm?id=1599871.1599933.

[275] José P Zagal, Michael Mateas, Clara Fernández-Vara, Brian Hochhalter, and Nolan Lichti. Towards an ontological language for game analysis. *Worlds in Play: International Perspectives on Digital Games Research*, 21:21, 2007.

[276] Achim Zeileis, Torsten Hothorn, and Kurt Hornik. Model-based recursive partitioning. *Journal of Computational and Graphical Statistics*, 2008.

[277] Alexander Zook and Mark O. Riedl. Automatic game design via mechanic generation. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence*, pages 530–537, 2014.

[278] Alexander Zook and Mark O Riedl. Generating and adapting game mechanics. In *Proceedings of the 2014 Foundations of Digital Games Workshop on Procedural Content Generation in Games*, 2014.

[279] Alexander Zook, Eric Fruchter, and Mark O. Riedl. Automatic playtesting for game parameter tuning via active learning. In *Proceedings of the 9th International Conference on the Foundations of Digital Games, FDG 2014, Liberty of the Seas, Caribbean, April 3-7, 2014.*, 2014. URL http://www.fdg2014.org/papers/fdg2014_paper_39.pdf.