Mechanizing Exploratory Game Design

Adam M. Smith

Contents

To Kathleen.

Abstract

Game design is an art form that deals with inherently interactive artifacts. Game designers craft games (assembled from rule systems and content), but they really seek to manipulate play: the interaction between games and players. When developing new games that are similar to past games, a designer may rely on previous experience with related designs and relatively easy access to players familiar with conventional design choices. When exploratorily venturing into uncharted territory, uncovering games that afford novel modes of play, there is a need for practical and technological interventions that improve a designer's access to feedback from the unfamiliar design scenario. In the interdisciplinary space between game design, design studies, computational creativity, and symbolic artificial intelligence (AI), my program of mechanizing exploratory game design aims to amplify human creativity in game design; enable new game designs through deep, play-time design automation; and demonstrate novel tools that respect the concerns of design problems.

This dissertation advances a practice of modeling design spaces as logic programs in the answer set programming (ASP) paradigm. Answer set programs can concisely encode the key conditions of artifact appropriateness, and, paired with state of the art algorithms for combinatorial search and optimization, they yield efficient and expressively sculptable artifact generators. I present three major applications of ASP-encoded design spaces to exploratory game design: a powerful machine playtesting tool for analyzing designer-specified games, a novel game prototype employing machine-generated rulesets bound by iteratively discovered constraints, and a suite of level design automation tools that offer unprecedented control over the aesthetic and pedagogical properties of puzzles for a widely deployed educational game. This practice is further developed in a series of introductory programming and advanced modeling tutorials. By formally modeling design spaces as concise and authorable logic programs (and without the need for original algorithm design and maintenance), designer-programmers are broadly empowered to quickly build and evolve the in-house automation required by their own exploratory game design projects. These claims are backed by a spreading adoption of this practice by others and deployed applications to at-scale game design projects.

Taken as a whole, this dissertation offers new insight into the nature of game design in relation to other design disciplines, a new design research practice for understanding how design thinking can and should be mechanized, a novel model of transformational creativity suitable for discussing both humans and machines, and a set of new applications for symbolic AI that simultaneously broaden the applicability and tax the limits of this automated reasoning infrastructure.

Acknowledgements

First, my deep thanks go to Michael Mateas, Jim Whitehead, and Noah Wardrip-Fruin for creating a vibrant intellectual environment within the Expressive Intelligence Studio and the larger Center for Games and Playable Media, where ideas outside of computer science proper could grow and refine through contact with those well versed in the arts and design.

I am immensely grateful for the volumes of input received from the subset of my labmates that have populated our lab IRC channel continuously for the last several years: Mark Nelson, Gillian Smith, James Skorupski, Anne Sullivan, and Ken Hullett. Together we've authored over 350,000 lines of dialog, and a fair fraction of that has been focused on scholarly subject matter. I take special pride in that the work presented in this dissertation is influenced by Mark's ideas on game design automation and Gillian's ideas on procedural content generation. The ideas I've co-developed with the others are reflected in publications that didn't quite fit into the story spun by this document.

Outside of academia, some of my personal friends in Silicon Valley have kept me in close contact with hacker culture, indie game design, and design in many other forms: Jeff Lindsay, Nathan Saichek, Joël Franusic, Adrian Perez, Jon Hull, and Russ Fan.

Linked primarily through the Internet, I want to acknowledge the significant inputs I've gotten from Martin Brain (my immediate upstream contact on all ideas related to answer set programming and parallel discoverer of this technology as an amazing tool for game content creation). I owe a similar debt to those on the Potassco team who have developed (and improve daily) the software tools on which this dissertation is founded. Special thanks go to Martin Gebser and Roland Kaminski for their clues on and off the `potassco-users` list that are propelling forward my post-dissertation research.

I would like to thank Zoran Popović at the University of Washington Center for Game Science for inviting me to the join Erik Andersen, Eric Butler, and others on the *Refraction* team in their on-going educational game development effort. The design automation tools I was able to make for *Refraction* made for a perfect at-scale application of my research and have directly inspired the research I will be continuing.

For their help in last-minute proof reading of this dissertation months into the start of my postdoc position, I want to thank Joe Osborn, Sherol Chen, Aaron Reed, Mark Chen, Matthew Burns, Aaron Bauer, and Alex Jaffe.

Finally, procedurally typeset words can only crudely approximate my gratitude for the companionship I've received from Kathleen Tuite, with

whom I've experienced an alternate, parallel life in computer science graduate school separated by significant measures of time and distance. Finishing this dissertation, we can now make our life together in Seattle.

Chapter 1

Introduction

The ability to create games is an ability to create human culture. Driven by curiosity about what is or what might be possible, exploratory game design continually seeks to extend our ability to author and engineer new kinds of games. The challenge I take on in this dissertation is that of accelerating exploratory design practices with the machine-supported reasoning of artificial intelligence.

Games have a property not clearly shared by many other cultural forms such as music, poetry, and sculpture; they are unavoidably interactive. The spaces of play afforded by a game, that by which we judge its impact on an audience, are the emergent result of entangled interaction between artifact and environment, between game and player. The appropriateness of a game is in its play, not in the structure of the game itself. Further, it is not enough for the rule system at the heart of a game to simply *allow* or be compatible with interesting play; it is the designer's responsibility to *drive* the emergence of desirable gameplay, a negotiation between a fixed game design and the variable whims of different players.

Designers craft rule systems with imperfect knowledge of their gameplay implications. What design knowledge they have is the product of creating, playing, observing, deconstructing, and reconstructing similar games. When staying within the confines of familiar game genres and implementation techniques, one designer may productively learn from the artifacts and experiences of another. However, when venturing into new design territory, the abundance of relevant examples to learn from vanishes. An exploratory designer must take on the responsibility of building up his or her own constellation of related prototypes before gaining the experience necessary to author games in this new space with intent and confidence.

Where transferring knowledge from pre-existing designs to an active

design project is a challenge of abstraction and analogical reasoning between familiar endpoints, the creation of new designs on the frontier is a challenge of concretion, the synthesis of sketches strictly outside of familiar territory. When the primitives under investigation are conditionally executing rules that update the abstract state of dynamic systems (as opposed to, say, strokes of pigment on canvas or musical notes on a staff), synthesis is clearly rooted in abstract symbol manipulation. Proposing a rule to allow a new kind of interaction, altering initial configurations to refine a challenge, or discovering plausible inputs (evidence of potential player strategies) that drive gameplay along interesting trajectories all exercise a designer's symbolic reasoning abilities, onerously at times.

Through artificial intelligence techniques, machines afford mechanization of symbolic reasoning. Given a theorem, one can set a machine to construct a proof of its truth or a counterexample evidencing its invalidity. Given a body of knowledge, one can ask a machine if a new statement is entailed by it. Further, given a set of observations, one can ask for the simplest novel explanation given a body of background knowledge. But what are the theorems of game design? What does game design knowledge look like? What are the observations in need of explanation? A mechanization of game design will shed light on these questions.

Game design is an activity that can clearly consume the best of human creativity, so marrying the concerns of game design to the affordances of machines is not a challenge to be taken lightly, nor is it one that will leave either game design or human creativity undisturbed. I do not intend to reduce game design to machinic procedures; instead I hope to demonstrate automated processes that reflect the previously unarticulated essence of several of the core elements of creativity in game design.

The mechanization of exploratory game design entails ***the introduction of machines into the game design process in a way that meaningfully reduces a designer's symbolic reasoning burdens and extends his or her creative reach***. This dissertation explores the following questions: How can we mechanize exploratory game design? What does such a mechanization imply for creativity in game design? And what does this say about machines when they begin to carry out significant design automation activities?

1.1 Goals

The goals of this dissertation are three-fold, revolving around the amplification of human game design creativity, the creation of new kinds of games that require significant design automation during their play, and the demonstration of software aides that respect the nature of design

problems as opposed to engineering problems.

1.1.1 Amplify creativity of human game designers

By revealing what creativity in game design is after, we can begin to identify bottlenecks and pair these bottlenecks with the abundant resources of modern-day computing. When technically literate designers can use technology to make creative leaps with reduced risk and cost, they are empowered to explore territory their un-augmented[1] counterparts cannot afford to attempt. The examples created by these forward-looking explorers (in the form of playable game prototypes, reusable clusters of game mechanics, and reference gameplay traces highlighting the impact of concrete design choices) will allow other designers to interpolate an understanding of the new territory, enabling the informed and intentioned design of new cultural artifacts.

This dissertation advances exploratory design practice that allows for new forms of feedback to be had even for games that are just emerging from the conceptual development stage.

1.1.2 Support deep, play-time design automation

Mapping the processes of game design to automated procedures allows a new class of games to be created: games that involve significant design effort by the game system itself during gameplay.

In table-top role playing games, the game master (sometimes called the dungeon master or referee) may design original content (maps, enemies, quests, etc.) on the fly and in response to the emergent trajectory of gameplay so far, and they may also negotiate with players to form new rules and exceptions to existing rules that directly steer gameplay along interesting lines that a strictly rigid world configuration and rule system would not allow. This kind of gameplay requires a designer integrated into the structure of the game.

By employing significant play-time design automation, we can create videogames with the finely sculpted and player-adapted gameplay experiences that were previously only reachable with the integration of a dedicated, human designer into each and every gameplay instance. Depending on the nature of automation techniques employed, these new games have a potential to be radically more relevant and precisely tuned to an audience than the capabilities of any human game master. In this dissertation, I

[1] I refer to Engelbart's sense of augmentation [68]: "increasing the capability of a man to approach a complex problem situation, to gain comprehension to suit his particular needs, and to derive solutions to problems."

develop methods that can be used to produce play-time design automation systems and demonstrate them in an application to a widely deployed game.

1.1.3 Demonstrate tools that respect design problems

The available tools for game construction primarily support the construction of software artifacts by addressing the problems of software engineering. Integrated development environments may bundle testing toolkits, debuggers, and even formal verification machinery. However, these tools are mostly centered around the problem of helping an engineer bring their working software solution into compliance with a vetted, pre-defined specification.

A kind of ill-definedness is in the nature of all design problems, negating the direct use of specification-oriented tools. A new breed of tools for designers that respect design problems as design problems will treat problem specifications as untrusted and as fluid as the details of candidate solutions. These tools will emphasize discovering new requirements over repairing an artifact with respect to known requirements. By demonstrating tools that respect design problems, I will illuminate new uses of automation in the service of extending a game designer's creative reach and highlight new goals for future design-assisting tools.

1.2 Research Context

While the research described in this dissertation was carried out in the context of an academic computer science department, the set of fields my research draws from and contributes to reaches across traditional disciplinary boundaries. In particular, my work is relevant to game design, a field with significant development outside of academia; design studies, the scholarship of design processes across all domains of design; computational creativity, a synthesis of computation and the philosophy of human creativity; and symbolic artificial intelligence, the interface between computation and general symbolic reasoning.

1.2.1 Game Design

Game designers seek to design play, but they must settle for indirect control. The recorded knowledge of human game design efforts is split between postmortem analyses of existing games and the philosophies and practices for creating new games. Postmortems capture knowledge of which design choices worked or failed in creating an intended play experience, while design practices (such as prototyping and playtesting) and

philosophies (e.g. playcentrism, minimalism, proceduralism, etc.) are intended as guides to be followed when making design choices in future game projects. Anything that helps clarify a designer's path from the low level structural choices in a game (which may be invisible to players) towards realizing the high level gameplay experience they intend is a contribution to our shared knowledge of game design.

My research offers a preview of new kinds of tools for assisting game designers in configuring both a game's rules as well as its content, infrastructure for building new kinds of player-adapting games, and an explanation of the creative processes of game design (that goes beyond the simplistic model of optimizing a player's expected fun value).

1.2.2 Design Studies

Design studies seeks to understand the underlying structures of design processes as exemplified by the application of design across all domains, from intangible software systems and microscopic electronic systems through industrial product design and on to architectural and larger-scale urban design. The literature of design studies identifies distinct and universal phenomena of study, appropriate methods and philosophical values that are not reducible to either the sciences or the humanities. The third discipline, as design is sometimes called, deals with the development of technical solutions to necessarily ill-defined problems. Design research, a forward-looking branch of design studies, seeks to advance the processes of design and transform the space of artifacts that result from these processes.

My research builds on the vocabulary (such as ill-definedness, appositional reasoning, and design spaces) and universal strategies of design studies (such as iterative refinement, active learning in reflective practice, and leading with candidate solutions) to develop a broadly-informed understanding of game design. In my own design research practice, I ground these cross-domain notions in computational models that support the design of interactive artifacts.

1.2.3 Computational Creativity

The field of computational creativity overlaps with artificial intelligence, the philosophy of human creativity, and the practical domains of several cultural media. Computational creativity seeks to understand and explain how machines can be creative, to explicate human creativity in computational terms, and to demonstrate computational models of artifact generation in various realms traditionally dominated by human creativity:

music, poetry, painting, jokes, mathematics, science, and so on (certainly including game design).

Towards any of these goals, it is interesting to see demonstrations of machines that either augment or replace human creative efforts. In this dissertation, I offer this spectacle in the domain of game design, and I explain it in terms of a novel model of creativity. Defined at the knowledge level, this model speaks to the perception of sustained, transformational creativity in rational agents, regardless of whether they are human or machine.

1.2.4 Symbolic AI

The technical content of this dissertation is founded in formal logic, the currency of symbolic artificial intelligence (AI) techniques. Symbolic AI starts from the physical symbol system hypothesis (originally due to Allen Newell and Herbert Simon, two names that occur frequently in this dissertation): a physical symbol system has the necessary and sufficient means for general intelligent action. The machinery of symbol manipulation, they claim, is a workable stand-in for intelligence.

We need not accept this hypothesis at face value (indeed, this dissertation does not require it) to make abundant use of the ecosystem of formalisms and software tools that have clustered around symbolic AI. My use of logical predicates, connectives, quantifiers, entailment and proof are utilitarian. When I employ a symbolic knowledge representation it is not to represent the essence of the real world, but to explain my problems in sufficient detail to ensure its computed feedback is relevant to me. And, when I employ an exhaustive search process where humans seem to use only a spotty guess-and-check method, I do this to exploit an imbalance between human and machine costs of search, not to suggest the human's search, if perfected, should be exhaustive.

Though humans and machines may manipulate different symbols to reason in different ways, it is useful to imagine that they both *reason*. When I develop a game design automation system atop formal logic for pragmatic reasons, it is always possible to go back and inspect, through the lens of the physical symbol system hypothesis, what that system is saying about intelligence in design, artificial or otherwise.

1.3 Contributions

The core contributions of this dissertation are technical methods for carrying out several exploratory game design processes with a machine. Secondarily, situating these methods within a larger context of design and

creativity, I point a way towards a program of research that will yield a broader mechanization of creative, exploratory design.

1.3.1 Mechanizing Design Spaces

My primary contribution is the development of techniques, based on symbolic AI, for modeling and refining the structural design spaces that arise during exploratory game design. Design space models are declarative specifications of a set of artifacts that are distinctly appropriate or relevant to a working design scenario. The intent to model a design space as a whole (including the rationale for judging appropriateness) is an important departure from the traditions of computer-aided design (CAD) that are focused on modeling individual artifacts, often divorced from design rationale.

Capturing design spaces, which otherwise only exist as informal expectations and preferences in a designer's mind, as machine-manipulable structures allows the mechanical synthesis of new artifacts and the analysis of hand-created artifacts with respect to conditions of interest. Applied to specific spaces of game structure, it is possible to automatically sample both data-like content (e.g. puzzles and maps) and code-like rule sets (e.g. event handlers and victory conditions) with properties such as solvability, fairness, and conformance to genre conventions. Applied to spaces of play for a given game, it is possible to carry out a kind of machine playtesting that allows a designer to anticipate emergent player behavior and understand the elements of a game's structure that makes that kind of play possible and relevant. Finally, applied to spaces of play traces collected from human players, it is possible to explore the intersection between expected and observed play, finding patterns of empirical interest and then turning these patterns into new constraints on the design spaces for game structure and game play.

I propose and argue for answer set programming (ASP) as a practical reference model for the capture of design spaces. Answer set programming is a declarative programming paradigm focused on complex (NP-hard) combinatorial search and optimization problems with a symbolic logic representation. ASP-encoded design spaces support automation of complex synthesis and analysis tasks without the need to develop and maintain domain-specific search and inference infrastructure. To cash out this proposal, I describe several example design automation systems, noting both high-level modeling strategies and low-level programming idioms. Although many of the individual affordances of ASP can also be found in isolated symbolic AI tools (such as deductive databases and Boolean satisfiability or numerical constraint solvers), my comprehensive programming practice makes extensive use of the integrated nature of

ASP tools to gain rapid (and often visual) feedback on design problems.

Applying the concepts and representational techniques from the ASP-based reference model, I contribute several larger systems that automate symbolic reasoning in a variety of practical, exploratory game design problems. These applied systems forge a link between the abstraction of design spaces and the concrete processes of game design (including playtesting with human players and puzzle design for an educational game).

Having a formal and realistically authorable representation for design spaces that is shared between human and machine makes important progress towards my research goals. Offloading large chunks of the symbolic reasoning effort that is needed to synthesize new designs and diagnose flaws in current designs, designers are empowered to make larger creative leaps knowing that inspiring examples and counterexamples in the new territory can be had at a significant discount over lone contemplation. When the constraints on what makes an artifact relevant to a given play experience can be formally articulated, the same systems that assist a designer in offline exploration can be embedded into the game in the form of play-time design automation. Finally, tools that accept design space descriptions as input represent an important new class of tools that are informed by the nature of design problems.

1.3.2 Situating Mechanized Design

My secondary contribution situates the mechanization of exploratory game design within the larger context of design thinking and computational creativity. Starting from the vocabulary of design thinking, I relate the practice of problem shaping and the mode of design cognition called "appositional reasoning" to the affordances of existing symbolic AI tools. This link motivates the requirements of a formal representation for design spaces and suggests a role for mechanized design spaces outside of game design. Again, I use examples from ASP as the reference model for design spaces; however any number of alternative tools may satisfy the same requirements to be useful as a means to automate appositional reasoning.

With an understanding of how various aspects of design can be mechanized with the tools available today, I ask what a complete mechanization of creativity in exploratory game design would require of future tools. To transfer the goals and methods of game design out of an intrinsically human realm, I develop an explanation of creativity in game design that is cast at the knowledge level. The knowledge level is a systems description level, introduced by Newell, which defines a sense of rationality for agents independent of their symbol-level description. *Rationalizing* creativity in game design at the knowledge level opens the door to alternative *mecha-*

nizations of creativity at the level of physical symbol systems. Identifying the designer's goal as the satisfaction of curiosity about game design knowledge, this explanation suggests the need for future computational tools that could realize rational pursuit of this goal. Immediately, this points out several new classes of software tools that would be of use to creative game designers (e.g. design-pattern-aware decompilers). Further, it suggests an overall architecture for software systems that attempt to sustain transformational creativity in game design: that of discovery systems (a class of systems organized around knowledge discovery through open ended experimentation, previously only applied in science and mathematics).

I build the link between automation available today and a potential future of broader automation (one that would shed light on how creative human designers operate) through two diverse methods. The first is the use of the top-down theoretical explanations mentioned above, and the second is a kind of constructive, bottom-up method akin to creating science fiction. Through *computational caricature*, a design research practice I describe and motivate, others and I have developed working computational systems that provide grounded, executable arguments for how various aspects of creative game design might be automated in general without having to have solved every technical problem general automation would require.

By tentatively situating my design mechanization methods within the context of design thinking and computational creativity, I provide the theoretical background and research methods that others can follow to make further impact on the goals of this dissertation. With a general explanation of the goals of creative game design in hand, I highlight new ways in which automation might amplify human creativity (namely by removing bottlenecks in the discovery process). The desire for play-time design automation makes concrete the need for the machine-readable forms of design knowledge that a discovery system in game design would consume and produce–that is, the need for automation of creativity in game design is motivated by the practical need to produce the formalized knowledge required by future adaptive and self-configuring games. Finally, design thinking and computational creativity provide a motivation for design-respecting tools while computational caricature provides a method for prototyping in the development of these tools.

1.4 Approach

In making the above contributions, I have adopted five crosscutting strategies to scope my work and guide my exploration.

1.4.1 Design Spaces

My first research strategy is the emphasis on abstract design spaces over the concrete artifacts contained within them. I operate under the assumption that *designers are trying to refine their understanding of the space of appropriate artifacts*, not simply trying to design a single example. By focusing on a construct that spans many artifacts, knowledge from one artifact need not be explicitly transferred to the design of another artifact: we can speak of constraints on the design spaces that contain both of these artifacts (and myriad other artifacts with similar configurations). This focus avoids the trap of design automation mechanisms that are locked to the artifact level, forever manipulating individuals without the capacity to learn general design principles in a domain.

1.4.2 Computational Caricature

The pervasive use of computational caricature is my second strategy. To amplify designer creativity, for example, I need not develop a functional reference implementation for all of human creativity. Instead, my approach is to select key elements of game design that are poorly defined (when practiced by human designers) and to develop intentionally exaggerated and over-simplified computational systems that are designed to make my claims regarding the formalized essence of that element immediately recognizable.

1.4.3 Target Audience: Procedurally Literate Designer-Programmers

My third strategy is to target my computational caricatures for interpretation by a *procedurally literate* audience of designer-programmers. While there are many productive game designers who are not procedurally literate (i.e. they rely on others in a design team to craft the computational systems that support the designs they describe), I believe designer-programmers are the ones who have the most to gain from a mechanization of game design, and they, being intimately familiar with both the concerns of design and the affordances of programmed machines, are the ones who can provide the richest feedback for this research. Developing tools for novice designers may increase the size of the game creating population (eventually yielding more designer-programmers), but I estimate that further empowering the designer-programmers who are already able to probe the frontiers of game design is the best way to expand our shared creative reach.

1.4.4 Automating Logical Reasoning

My fourth strategy is a restriction on the strengths of computation that I intend to exploit. While modern-day computation offers mass communication and storage facilities as well as bulk number crunching abilities that far outstrip those of human users, I opt to primarily build on automated logical reasoning (symbol manipulation). In posing logical reasoning problems that are worth working on, humans have an advantage; in solving these problems, machines have the advantage (particularly when the problem boils down to tedious combinatorial search). Nonetheless, these problems are founded on an important common language: that of symbolic logic. Explaining essential elements of design thinking in terms of logical reasoning permits formalized problem specifications that either humans or computers may solve, allowing fluid reassignment of effort to bypass the bottlenecks that arise in particular design projects.

1.4.5 Artifacts as Communication

My final strategy is to use concrete artifacts (specific rule sets, detailed elements of game content, or replay-capable gameplay traces) as the primary form of communication from automated design tools back to human designers. That is, designers will speak in candidate design space models and listen for artifacts in response. When using existing formal verification tools, a designer hopes to learn that, yes, there is a proof (often by uninteresting exhaustion) that a complex set of carefully specified properties hold simultaneously for a single design in question. When using design tools designed with my artifacts-as-communication strategy, a designer hopes to learn that, no, their loosely and speculatively defined concept of interest (often regarding potential failure modes) is not empty. They hope to see several detailed traces evidencing how that interesting concept can be instantiated. Further, offering several detailed artifacts in response to a tentative question allows a kind of I'll-know-it-when-I-see-it exploration that is compatible with a designer's often informal curiosities.

1.5 Outline of Dissertation

Concluding my introduction, I now give an outline of the chapters to follow.

Chapters 2 through 6 cover background material, highlighting the key ideas and philosophies from each of the four disciplines I mentioned above. These chapters are organized as an opinionated reconstruction of the most relevant aspects of the related fields as opposed to a neutral review. In

Chapter 7, I synthesize the goals of this dissertation as concerns at the intersection of triples of these disciplines.

Chapter 8 unpacks appositional reasoning, a central process in design thinking, and reveals requirements on knowledge representation schemes that could satisfyingly serve an automation of this mode of cognition. Linking these requirements to previously demonstrated affordances of answer set programming, I describe how to map the concerns of design, in the vocabulary of design studies, onto machine-supported (abductive) logical reasoning.

Chapter 9 introduces computational caricature, the design research practice I have adopted in exploring the role of artificial intelligence in the game design process. In addition to directly reviewing one of my systems through the lens of caricature, I describe systems created by others that similarly function as caricatures of artificial intelligence in the design process.

Chapter 10 adopts the perspective of the designer-programmer applying answer set programming to design automation problems. This chapter introduces the syntax of logic programming, outlines an iterative process for defining and refining design space models as answer set programs, and provides line-by-line programming tutorial examples. It also presents mini-caricatures (rational reconstructions) of design automation systems created by others.

Chapters 12 through 15 present four applied design automation systems developed according to the practice presented previously: Ludocore (a queryable formal model of gameplay), Biped (a game sketching tool with equal support for human and machine playtesting), *Variations Forever* (a game prototype involving dynamic ruleset generation), and a suite of design automation tools for *Refraction* (a complex, pre-existing game that was not designed with the use of my tools in mind).

Chapter 16 describes rational curiosity, my working model of creativity that is defined in knowledge level terms, and applies it to describing creativity in game design. Although the theory of rational curiosity makes broader claims about human and machine creativity, I primarily focus on the game design impact of this theory.

Chapter 17 evaluates my impact on the goals of the dissertation outlined above, also examining the impact my developments have had on the next generation of students who will go on to develop their own design automation systems.

Finally, Chapter 18 briefly outlines a program of future work and offers my conclusions.

Chapter 2

Interdisciplinary Context & Motivations

From the perspective of game design alone, a program to mechanize exploratory game design first appears as a curiosity: a "wouldn't it be nice if–" project to be taken on someday in the far future. Zooming out to consider inquiry beyond game design, a broader motive and a number of promising means to pursue this program emerge.

In the next four chapters, I present an opinionated reconstruction of the interdisciplinary context of my research. I sketch out the philosophies and practices that I draw from and apply, motivate my choice of research methods, and setup how my work contributes to the disparate goals of several fields. My intention here is to map out four disciplines (game design in Chapter 3, design studies in Chapter 4, computational creativity in Chapter 5, and symbolic AI in Chapter 6) in enough detail to highlight important cross-links that promote new lines of thinking that would go unexplored from the perspective of any field in isolation. This involves looking at game design through the lens of general design, linking the study of human and machine creativity to design automation through the processes of artifact generation, and using the techniques of symbolic AI as the foundation for technical systems that address games, design, and creativity. Chapter 7 synthesizes the goals of this dissertation in the interdisciplinary space between these fields.

Chapter 3

Game Design

Game design, boiled down to a single statement,[1] is the art and craft of building playable systems so that the emerging player interaction with these systems meets with the designer's intent. This might be approximated as the configuration of rules and content so that the designer's intended audience will experience enjoyment, but this is just one perspective on a complex and subtle topic: the creation of material culture.

In this chapter, I review game design roughly from the perspective of the lead designer on a videogame, someone who is responsible for crafting the core interactive systems and supporting details of a game that will be realized in software. This perspective downplays opportunities to relate game design to literature and visual art, and it largely ignores the traditions of tabletop game design. However, I claim this perspective is useful for developing a mechanization of game design that advances the goals of this dissertation.

In the rest of this chapter, I review game design as it is taught in textbooks and discussed at industry conferences and other community forums. Then I review a call for more structure and formal understanding of the game design process by some, but importantly not all, of the game design community. Next, I survey published games and academic research that employs procedural content generation (the mechanized creation of game content either at design-time or play-time) and then review various attempts at a general formalization and automation of game design (ranging from generalizations of procedural content generation to opera-

[1] My intent is not to provide a final definition for game design; instead, I want to highlight the importance of a designer's intent. When overzealous application of playcentric design philosophies break into player-centric design, the role of the game designer as an artist is unduly diminished. User-centered design of pleasure machines is something distinct from game design.

tionalizations of design philosophies). Finally, I offer some perspective on game design as a design discipline, able to influence and be influenced by the practices of design in other domains.

3.1 Best Practices

Designers and developers of early videogames cobbled together development practices and creative inspiration from their experience at hand: playing sports and tabletop games, manipulating physical puzzles, and programming computers to run mathematical simulations. Today, the novice designer can readily access an accumulation of videogame-centric wisdom in the form of influential textbooks such as *Game Design Workshop* [73], *Rules of Play* [176], and *The Art of Game Design* [179]. Community websites like Gamasutra[2] archive project-by-project postmortems that record the successes and failures of well-known games. Finally, the latest speculations and emerging philosophies of game design spread at industry conferences like the annual Game Developers Conference.[3]

Amongst a wide array of best practices, three broad categories are relevant for discussion in the context of mechanizing exploratory game design: early-stage prototyping, playtesting across all stages of game design, and late-stage balancing.

3.1.1 Prototyping

The practice of prototyping seeks a way to get feedback on a game design without the effort to design and implement a complete game. Different types of prototypes hit different points on a tradeoff between quality of feedback and the effort required to get that feedback. Seeking insight that will refine and evolve their working game concept, designers usually start with relatively cheap, low-commitment prototypes that afford broad exploration and then shift to more expensive and better representative methods as the design project converges.

Paper prototyping [189] is a common technique for testing risky game concepts with very little construction effort. Paper prototypes are, in essence, partial tabletop games that are played with materials recycled from other games (playing cards, dice, miniature figures, etc.) and sundry items from home or office (graph paper, whiteboards, coins, paperclips, etc.). In contrast with complete tabletop games, the rules of play for paper prototypes may or may not ever be written down and are subject to change at the designer's whim. When playing a paper prototype, the

[2] `http://www.gamasutra.com/features/postmortem/`
[3] `http://www.gdconf.com/`

designer may step in to amend die rolls, introduce exceptions to rules, or rearrange the game world in an effort to guide gameplay along trajectories that explore phenomena of interest to the designer.

Where paper prototyping gives a designer agility (in terms of being able to quickly jump between alternate designs without major construction efforts), it burdens them with mental storage and computation during play. The designer, acting as a game master (or GM, in the sense of tabletop role-playing games), must track the state of any game elements that are not fully represented by physical objects or symbols on paper as well as compute updates to on and off-paper state according to consistent rules (all the while contemplating whether to introduce alternative rules). When the pace at which a gameplay prototype can react to player choices is bounded by a single human's computation ability, certain design questions, such as those relating to pacing and twitch reactions, are unanswerable with paper prototypes. Nonetheless, insight regarding even complex, real-time videogames can be gained through paper prototyping [73, chap. 7].

In computational prototypes (also called digital prototypes [73, chap. 9]), fragments or approximations of a candidate game design are realized as computer programs. By automating much of the manual bookkeeping associated with paper prototyping and enforcing rules in a rigid manner, computational prototypes allow for observation of play that is not directly mediated by the designer (and thus subject to the designer's bias and play-time manipulations). The cost of this automation, however, is both the effort required to develop and maintain the necessary systems and the effort required to cast previously informal and negotiable rule systems in formal terms. In exchange for engineering work, computational prototypes offer far more representative feedback from a design scenario. Many computational prototypes incrementally approach the final game in complexity and fidelity. Along the way, computational prototypes make use of placeholder content and "programmer art" as a way of fleshing out a playable rule system without involving a much larger creative team.

In reality, there is a continuous and evolving spectrum of methods between paper and computational prototypes. For example, Fullerton [73, p. 9]) suggests using a spreadsheet to inexpensively automate only the most tedious numerical bookkeeping of an otherwise paper prototype, retaining agility for other design elements (such as map design and non-player character behavior). Joris Dormans' game feedback diagrams (or "Machinations") [58, chap. 4] are executable pictures of the key resource flows in a game design. Simple diagrams allow a designer to reflect on feedback structures (potentially relating them to general feedback design

patterns[4]) or run basic simulations, and more complex diagrams can implement playable mini-games.[5]

Towards the goal of expanding a designer's creative reach and demonstrating tools that respect design problems, part of my research explores the development of new prototyping methods. New kinds of prototypes have the potential to offer designers new points on the feedback–effort tradeoff, allowing them access to richer feedback on a fixed budget or reducing the cost of answering specific design questions. The Biped system presented later in this dissertation in Chapter 13 offers one such new point between traditional paper and computational prototypes: it allows automated testing (which is rare for any kind of game prototype) for designs expressed with relatively little engineering effort. Meanwhile, the general strategy of prototyping, building partial systems that elicit feedback (also called "backtalk") from a design scenario, will be reviewed more broadly from the perspective of design studies in Chapter 4.

3.1.2 Playtesting

The practice of playtesting aims to bring game prototypes into contact with players in a way that results in insights not derivable from inspection of the prototype on its own. In particular, playtesting can illuminate the ways in which interactions between structures in a game bring about different spaces of play.

The result of playtesting is not simply an evaluation or judgment of how well a game plays (as one might summarize with a score or marks on a survey); instead it results in a collection of evidence that a designer may interpret in many different ways. Regardless of whether the player involved in a playtest reports experiencing "fun," the designer might gain insight from the set of choices the player objectively made or did not make, any neutral expectations the player verbalizes or suggests through in-game actions, or the propensity for the player to transfer skills and strategies from related games. Observing these details in depth and in context permits credit and blame assignment of a player's overall experience to the particular subsystems encountered. For example, a player might report that the combat mechanic in a prototype was "boring and repetitive," but the designer can notice that the player continually ignored a certain action (perhaps a powerful combination attack) which sets this game apart from others in the same genre—but the player likely had no knowledge of this action because it was introduced in a tutorial sequence that the player had

[4] `http://www.jorisdormans.nl/machinations/wiki/index.php?title=Pattern_Library`

[5] `http://www.jorisdormans.nl/machinations/wiki/index.php?title=Case_Study:_SimWar`

quickly skipped through. In this example, playtesting reveals a potential problem with tutorial design by way of charged comments made on a seemingly unrelated mechanic.

As with prototyping, there is a tradeoff in playtesting between the accuracy of feedback and the cost required to elicit it. *Game Design Workshop* [73, chap. 9] highlights three stops on this spectrum: self-testing, testing with confidants, and testing with the target audience.

Self-testing, where the designer plays his or her own game, occupies the most lightweight end of the spectrum. Self-testing best affords working through the fundamentals of a game where sweeping changes are made to the design and the result would need to be refined more before others could understand it. In self-testing, it is easy to apply new gameplay constraints or game goals on the fly—a designer might think "I'll try to play this level without ever jumping" without formulating a compelling reason why that condition should be tested before simply attempting it. In this way, self-testing can be applied to rule systems that are not quite games yet (perhaps lacking clear outcomes). Even at later stages in the design process, self-testing is valuable for providing feedback immediately after a change is made.

Self-tests are biased by a designer's hopes and desires for a game and their knowledge of the game's internal workings. The least costly way around these biases is to perform playtesting with confidants (friends, family, and coworkers who may or may not be game designers themselves). Confidants can be trusted to see past many incomplete elements of a gameplay prototype (e.g. they can excuse your programmer art or lack of tutorial materials). Likewise, they are able to focus their feedback on the elements that have changed since the last version they tested. Although confidants are unlikely to be biased by knowledge of implementation details, the pre-existing personal relationship between designer and tester will still bias feedback.

Truly unbiased tests of a game design must involve members of the game's target audience. This kind of playtest provides an accurate simulation of play by the future first-time consumers of the game product. This form of testing is usually reserved for late-stage refinement and polish because of the costs involved in bringing playtesters into an environment that is realistic enough for feedback to be trusted and sufficiently well instrumented that the required feedback can be gained without disrupting the player in an approximation of their natural habitat. The very best kind of playtester is called a "Kleenex tester" [174, p. 446] in the sense that they are thrown away after a single use. The use of Kleenex testers ensures that players have no bias from experience with previous iterations of the game's design (though these players' experience with similar games in the same genre or even games in the same franchise is a valuable part

of what makes them representative of the target audience).

The textbook's categories leave no room for non-human (machine) playtesting, however I imagine automated playtesting to fall somewhere between self-testing and testing with confidants on the scale of accuracy and effort. Machine playtesting allows a kind of simulated playtesting with hypothetical audiences that could never be brought in for testing in person. For example, testing with strategically perfect machine players might lead to insights that traditional human testing could never reveal (because, for example, certain subtle strategies may never appear within the brief runs of human playtesting sessions). The hypothetical audience under test in machine playtesting is one of the designer's own crafting, so biases will abound. Nonetheless, the fact that machines think in very different ways than humans do makes them an interesting source of second opinions that (non-android) confidants cannot fully replace.

3.1.3 Balancing

Balancing is a practice usually reserved for the later stages of typical game design processes [73, chap. 10]. It often involves adjusting numerical parameters (e.g. the relative strength or cost of two actions) so as to tune the length and difficulty of typical gameplay sessions. Beyond numerical tweaking, balance can include adding or removing exceptions to general rules (such as giving certain characters special abilities or weaknesses) or other structural changes.

For most videogame projects, balancing must be completed before a game is shipped as a commercial product. This is unfortunate because having large interacting communities playing a game for long durations is perhaps the best way to get feedback on balancing choices. Newer development and deployment technology is making post-deployment balancing a possibility (it is typical for balancing updates to be distributed as software patches that are automatically applied). Even when eased by technological means, post-deployment balancing work can trigger player backlash when the familiar rules of a game change without notice nor means of reversion.

In the traditions of tabletop roleplaying games, it is the expected role of the GM to carry out significant customization and balancing of the game on the fly, altering default rules and the configuration of the game in light of the current play experience. In a potential future of player-adapting games that make heavy use of play-time design automation, I suspect that players will benefit from being explicitly informed that the game is being adapted to them. Even when players know to expect autonomous changes to the game, they should know whether the changes are applied only to them or to the player community as a whole—many

player-created online resources for games are centered around publishing and sharing the exactly those canonical values of many of the parameters that post-deployment balancing would aim to adjust. For example, an article on the *Terraria* wiki[6] includes a table noting that the maximum horizontal velocity of player characters who are either *well fed* or *wearing an Anklet of the Wind* (but not both) is exactly 12.50 blocks per second.

Although this dissertation does not advance any new practices specific to balancing, the concerns of balancing that arise in the design of specific puzzles and maps (e.g. for approximate difficulty and strategic fairness) will inspire constraints that we would like to model with with formal design space models. Similarly, the technology I develop for machine playtesting is applicable to some balancing concerns, but my focus is on the earlier stages of the game design process.

3.2 Folk Psychology in Game Design

Beyond concrete practices for how to build prototypes and carry out playtests, the shared wisdom of game design is flavored by the inclusion of folk psychology in the form of commonly held beliefs and convictions (when applied to game design) about players' mental processes. I refer to elements of knowledge that are relevant to (and often productive for) design without scientific validation in the specific context of game design. It is not my intent to cast doubt on the application of folk psychology; instead I note that these ideas are informative and productive of insight in game design projects despite their non-empirical nature.

A prime example of this is widespread (e.g. in Jenova Chen's MFA thesis "Flow in Games" [28] and Jesper Juul's *Half Real* [102]) reference to Mihály Csíkszentmihályi's theory of *flow* [50], a concept originally derived from the psychological study of art students [49]. Stated simply, flow is a mental state in which an individual's skill level is well matched to the challenge presented by a scenario. Flow is associated with intrinsic reward, a distorted sense of time, and action–awareness merging—often desirable effects for a game to have on players. Flow is a property of individual players in specific situations in play, not a property of the game itself. Thus, designing games for flow requires either very indirect shaping of play through the adjustment of the game itself (precise balancing) or the inclusion of additional play-time logic that will, in effect, rebalance the game to bring about flow for the current player.

Much of game design's folk psychology is not derived from scientific processes. Instead, much is developed through an expert designer's re-

[6] `http://wiki.terrariaonline.com/Player_stats`

flection[7]. A well-known example of reflection-inspired[8] folk psychology in game design is Raph Koster's "theory of fun" [114]. Koster claims games are machines for teaching and that players have fun when they successfully learn in the game. A good game, he says, is "one that teaches everything it has to offer before the player stops playing" [114, p. 46]. This designerly theory of fun is very productive of questions that drive the design of level progression in games (particularly in my own experience; see Chapter 15 in which educational goals drive puzzle design concerns): What are we teaching in the second level of the game? In which level will we later check that they can combine that knowledge with the new knowledge presented in level three?

Where many game design automation systems (such as some reviewed later in this chapter) try to optimize a proxy measure for "fun" in an overall way, I claim it is more interesting to build systems that respect the same strategies and understandings that a human game designer might be using when they design a game and its content. That is, I trust designers to have an accurate enough (or at least incrementally improving) understanding of their own audiences that I will not reach to the literature of psychology to justify the metrics and constraints used in automating the design appropriate rule systems and game content.

3.3 Call for Structure

As game design is nominally about the production of games, it is not a standard practice to collect and share structured knowledge of game design and design processes. Some of this happens organically (informal knowledge is commonly shared at conferences and less commonly, but slightly more formally, in articles and books), however many game designers wish it happened more and, further, that it be adopted as a practice by all designers.

In the introduction to his *400 Project*[9], a project to catalog a large number of unexpressed, informal rules of game design, Hal Barwood writes the following:

> Game design is an uncertain and murky endeavor. In this it resembles art, architecture, writing, moviemaking, engineer-

[7]The fringe status of reflection as a method of inquiry in science makes interesting contrast with the elevated status of "reflective practice" in professional design. See Chapter 4 for a discussion of reflective practice in a design studies context.

[8]Koster's theory of fun is indirectly inspired by psychological and cognitive science research. However, when we read his book, it is the clarity of the game-related examples and the resonance with the anecdotes he offers that persuades us, not the rigor of his research methods or source literature.

[9]`http://www.finitearts.com/Pages/400page.html`

> ing, medicine, and law. All of these fields do their best to reason through their problems, but all have found it necessary to develop practical rules of thumb as well. Hmm—could game developers benefit from a similar approach?

The rules collected so far, while indeed quite informal, do suggest that there is a wealth of design knowledge that can be shared. To catalog more rules, the project demands designers reflect on their own practice and, in effect, to become design researchers.

Barwood is not alone, nor is his bias towards informal knowledge universal. In "Formal Abstract Design Tools" [31], a call for a unified formal language for discussing games, Doug Church claims: "Not enough is done to build on past discoveries, share concepts behind successes, and apply lessons learned from one domain or genre to another." Church hopes by encoding discoveries in a common language that developers could share design knowledge as easily as they do the software libraries that make realistic graphics and physical simulations possible in modern videogames. Since the original article, Barwood's plea has been answered, in small part, by formal notation systems such as those of Koster [113] and Cook [41]. For the most part, however, Church's FADTs remain to be invented.

In "The Case for Game Design Patterns" [115], Bernd Kreimeier builds on the cry of Barwood and Church to further "establish a formal means of describing, sharing and expanding knowledge about game design" in the form of libraries of design patterns. This push has met with better success, as in the book *Patterns in Game Design* [12], the first (2012) International Workshop on Design Patterns in Games[10], and Ken Hullett's dissertation [95] which deeply investigates structured design patterns. Design patterns offer a powerful vocabulary for talking about design choices (particularly how and why to make certain choices), but there are as yet no standard game design tools that accept design patterns as input. Thus far, these patterns are descriptive (allowing reflection on existing designs), and they cannot be used prescriptively (as formulae for producing new designs) in an automated way.

In academic circles, the effort to build shared knowledge around game design has been adopted by some in the field of game studies. For example, The Game Ontology Project[11] [231], is simultaneously advancing a structured language for capturing game design knowledge and gathering a body of knowledge in that language. Similarly, there is effort to flesh out more precise terminology than that of common game design discussion such as "game mechanics." For example, "operational logics" describe the way in which meaning can be authored with the primitives available

[10] `http://dpg.fdg2012.org/`
[11] `http://www.gameontology.com/`

at the code/platform level [137].

The call for structure has not been universal. In the scathing article[12] "Against a Formal Language for Game Design," an anonymous industry veteran offers insightful counterpoint. The author succinctly summarizes the utopian aims of the above calls for structure in game design:

> The benefits [of formalization], its proponents claim, will be that game design will finally be rendered a field of study within which theories can be hypothesized, tested, and either proved or disproved. With these established knowledge systems in hand, then, we will be able to discuss, study, and teach game design more easily and reliably than we are currently capable of doing.

This seems a fair characterization, but consider the author's critical aims. He states his claim: "A formal language for game design will not lead to generating better game design; nor will it create consensus on topics in the field; nor will it make communication of game design concepts easier. In fact, a formal language will make it more difficult to spread knowledge in the field of game design." Citing a broad basis in philosophy and literary theory, he claims the establishment of universal truths of game design (particularly with regards to the appeal of a game for different audiences) is both undesirable and impossible. "The 'truths'," he continues, "expressed in a formal language of game design, in other words, would only be valid for a particular audience in relation to a specific conjunction of terms." Later, "the only people who could use a formal language to talk to each other," he pontificates, "would be the elite group who had invested sufficient time and money to learn the language."

To avoid a fall into any one of the multitude of dead-ends highlighted by this polemical essay requires that the end product of any proposed general structure for game design be considered. For the purposes of advancing the practice of game design, we should prefer formalisms that are productive, even if that productivity comes at the cost of scientific rigor or formal elegance. Workable formalizations of game design, it seems, should approach the messiness and situation-dependence of game design head-on. Abandoning the search for universal truths in game design, we can instead seek representations that efficiently express the local truths in specific design scenarios ("a specific conjunction of terms"). For such knowledge engineering to be worthwhile, the "time and money" invested must pay dividends in expanded creative reach and insight. When formal communication will only take place between a designer and his or her

[12] `http://www.micrysweb.com/office/formallanguage.html`

automation tools, the scope of topics to be formalized and the number of parties who need to intelligibly communicate is immediately reduced to a realistic scope. The situation-specific focus is the foundation of my artifacts-as-communication strategy.

Thus, while there is plentiful insight to be gained from browsing, for example, the Game Ontology Project wiki, I claim where structured knowledge is needed most is at the project-specific scale. In exploratory design, we may quickly uncover new territory for which the patterns fit to more common games break down. On this fringe, we want the support of external tools to help us organize and apply freshly uncovered knowledge of interesting and reusable structures.

3.4 Procedural Content Generation

One area where rigid rules for design and prescriptive formulae for design choices has gained acceptance is in procedural content generation (PCG). The label "procedural content generation" is a term borrowed from computer graphics; however, the generation of game content via stored procedures long predates this nomenclature. Canonical early examples of PCG in games are the generated dungeons of *Rogue* (Toy, Wichman, and Arnold 1980) and the generated of star systems in *Elite* (Acornsoft 1984). Today, many game titles feature generated content as a primary selling point such as *Spore* (Maxis 2008), *Minecraft* (Mojang 2011), and *Diablo III* (Blizzard 2012).

When compared with the alternative of creating game content via traditional means (manually, using data-oriented authoring tools), the unreliably-delivered promise of PCG is access to more content than could feasibly be hand-authored or the same quantity of content content with less effort. Blindly chasing this promise can lead to problems. Larger spaces of content are much harder to validate than hand-created collections, leading to practices like that used in *Spore* [40] where the output of the generative procedures that shipped with the game was limited to what was generated from a relatively small set of random seed values verified to generate appropriate content in in-house tests before shipping. Further, the effort to create a generator that produces content competitive with hand-authored content is high and may even exceed the effort to simply hand-author the required amount of content. This is not to say that generators cannot meet or beat the quality of hand-authored content. Of DIORAMA, a particularly sophisticated map generator for the real-time strategy game *Warzone 2100* (Pumpkin Studios 1999), one player writes:[13] "Looks to make as good or better maps than 90% of the

[13] `http://forums.wz2100.net/viewtopic.php?f=3&t=3011#p29075`

mappers efforts over the last 10 years. Congrats. :)"

3.4.1 PCG Meets Game Design

Used as a generative subsystem in larger games, PCG clearly has uses for improving replay value and supporting player-adaptation in traditional game design. However, in examining the motivations and methods of PCG (particularly those overlapping with AI), PCG sublimates into the broader reflective practice of game design.

From *Rogue* (Toy and Wichman 1980) to *Canabalt* (Adam Atomic 2009), games using PCG for replay value aim to reap the benefits of potential cost reductions and increased ranges of content. When players play through a game several times, an abundance of controlled variation in game content challenges them to master the general principles behind a game's systems instead of simply memorizing a single solution script. Where PCG cannot be applied in a game (perhaps the content is too subtle to be reliably generated), a common strategy for building gameplay value through content is to make the first (and perhaps only) playthrough of a game as exciting as possible through intense hand-authoring (using linear story-arcs, fixed cut scenes, or dramatic twists that yield the bulk of their impact during the first time they are experienced).

Using PCG for adaptation implies augmenting a generator with some means of maintaining relevance to a particular player's history. The effectiveness of PCG-based adaptation hinges on both controllability of the generator (i.e. its ability to reliably produce relevant content) and an understanding of the mechanisms of relevance. For example, in the abstract space shooter *Warning Forever* (Hikware 2003), the generator for the next boss ship to encounter is driven by knowledge of which weapon type last destroyed the player's ship and the order in which the player destroyed the components of the previously generated boss.

Adaptation, applied to its logical extreme, promises all of the benefits of a human game master and more. However, short of solving artificial general intelligence, adaptation introduces the potential for a game to fail in undiagnosable ways. When players get stuck in an adapting game, they may not be able to consult wikis, walkthroughs and other shared repositories of player knowledge that would help them get unstuck. Thus, adaptation should be considered when a specific aspect of the (game) mastered experience is desired and when the cost to potential player culture-building is acceptable.

Considering PCG as a first-class design element instead of a supporting technology is a key tenet of *PCG-based game design* [201], a concept defined by parallels to *AI-based game design* [65]. Instead of using "smoke and mirrors" to create the illusion of intelligence in game characters (ex-

ploiting the *Eliza effect* [226]) or attempting to create "black box" replacements for dedicated human level designers [40], these philosophies and methods suggest examining the affordances of particular systems and using them to deeply influence game design choices. Instead of trying to hide the functional quirks and ways in which a nominally supporting subsystem fails to support the desired illusion, these details can be exploited to unlock the potential for entirely new kinds of game mechanics. When supporting systems are developed in an attempt to solve well-defined or game-independent problems (e.g. attempting to make a fully-generic level generator), massive opportunities are missed to expand the space of reachable game designs with the surplus affordances of existing systems (such as by giving the level generator a means to communicate with a music generator, which many games might not have). Further, overly generic content generation attempts fail to expand the usefulness of PCG systems by bringing them into contact with the context-dependent concerns of specific game projects (e.g. allowing generated levels to support a game's fixed storyline instead of distracting from it). PCG-based game design moves beyond an engineering technique to a full-fledged game design research practice. *Inside a Star Filled Sky* (Jason Rohrer 2011) and *Endless Web*[14] are highly novel game designs that are simultaneously supported by and thematically oriented around unbounded spaces of game content.

In an invited talk [112] on the AI of *Darkspore* (Maxis 2011), Dan Kline offered his perspective on the strategy of replacing randomness (a low-sophistication component of many PCG and AI systems) with intentional AI direction[15] (in the sense of a film director). The design of *Darkspore* demanded significant replay value from a relatively small space of hand-authored missions (levels) as well as the automatic adaptation of these missions to player choices (such as how they customized their character). The introduction of AI into mission generation (an initially PCG-like goal) quickly broadened into a general strategy touching many other areas of the game. AI direction offers a way to steer both content and mechanics (such as simulated die rolls in combat resolution) into territory that is more interesting and relevant than would result from "fair" randomness. Kline captures a suggested progression of development with this pithy sequence: "`0, 1, Rand(), AI`"—the introduction of a new gameplay-relevant feature (perhaps a large boss character at the end of every mission) should proceed from being not present at all, hard-wired

[14] `http://endlessweb.soe.ucsc.edu/`

[15] Though I use Kline as the spokesman for AI direction, the idea precedes him and *Darkspore*. For example, *Left 4 Dead* (Valve 2008) included an "AI Director" as a much hyped feature. Much earlier, *Pac-Man* (Namco 1980) made heavy use of similar ideas to orchestrate ghost behavior. *Rubber-banding* and *dynamic difficulty adjustment* (DDA) are very similar ideas with long histories in game design.

to always be present, then present in an interestingly unpredictable manner, and finally present in exactly those conditions the designer deems appropriate to the player's ongoing experience (via conditions managed by an AI system).

3.4.2 Typical Architectures for Content Generators

The space of content generation procedures available for use in games continually evolves as new kinds of content are tackled and more efficient and more controllable procedures are discovered for the generation of familiar kinds of content. The broad categories describing the architecture adopted by each of these procedures, however, is much more stable.

Procedures such as the diamond-square algorithm [144] for terrain generation or *Elite*'s star system generator are said to be *constructive* [220]: they are (potentially) randomized procedures that directly construct a single content artifact without explicitly considering alternative designs or backtracking on unfortunate choices. *Composite* generators are said to have an overall constructive (or pipelined) architecture if they contain several sub-procedures organized in a feed-forward manner where the output artifact of one system becomes the input specification for another without forming any cycles. In Chapter 15, I compare a declaratively-specified generator that I created with answer set programming to an imperatively-specified generator consisting of seven pipelined stages.

Inventing a custom algorithm that will generate exactly the space of artifacts that a designer has in mind becomes increasingly difficult as new requirements are discovered. A way around this is to factor the content generation problem into two parts: developing a simpler-to-validate constructive process that overestimates the range of the generative space and pairing it with a filtering process that tests the validity of candidates proposed by the generator. In this *generate-and-test* architecture, it is possible to select for criteria (such as solvability of puzzles or a lack of easily detected design flaws) that would be unreasonable to forbid through clever constructive procedure design alone.

A rich example of this architecture is the Launchpad platformer level generator [203]. Launchpad uses a generative procedure (itself a composite of smaller constructive generators) to propose undecorated level designs. A series of critics evaluate candidate levels for adherence to a target trajectory and over or under-use of specific design components. Levels emerging from the generate-and-test system are piped through two final passes that decorate the level with coins (thus yielding a constructive architecture for the outermost system).

Ideally, new constraints can be added to generate-and-test systems by simply layering on more testing phases. However, efficiency concerns

often require major redesigns of these systems to push elements of the test process deeper and deeper into the generate process (a phenomenon noted as early as 1969 [190]). Preferably, the candidate generation process should adapt itself to feedback produced by the testing process automatically and on the fly. This is the strategy adopted in uses of evolutionary computation in content generators. While sometimes misleadingly classified as generate-and-test systems [220], evolutionary systems make the proposal of new candidate artifacts causally dependent on the result of testing previous artifacts, often by producing syntactic mutations and combinations of artifacts that have tested well in the past.

In a racetrack generation application, Julian Togelius et al. [217] defined the drivable areas of a track by connecting a set of control points by a thick spline. Because different control point locations lead to different player driving experiences, an optimization procedure is used to select a track that scores highly when evaluated by an algorithm trained to simulate the driving style of a particular player. Initially, the algorithm starts with a population of 100 sets of randomly placed control points. The simulated player drives each track, and the 20 best (according to the authors' cascading elitism strategy) are retained. Copies and mutations (via random perturbation) of these candidate tracks are generated to create the population of 100 tracks for use in the next iteration of the algorithm. Thus, evolutionary generators have components that are recognizable as filling either a generator or a testing role, however the overall architecture's use of test-contingent generation would seem to place it in an important category of its own.

In each of the architectures above, the generative space (the set of artifacts that a procedure might generate) is implicitly captured in the design of an algorithm. Understanding why a particular artifact would or would not be generated by an algorithm requires walking through the steps taken by the algorithm. Another architecture for content generators (one which currently lacks a standard name) attempts to make the definition of a system's generative space explicit. I have called this category "*declarative, solver-based*" [195] as it pairs a declarative definition of the generation problem with domain-independent solving algorithms (the choice of which does not affect the generative space beyond running time and statistical biases). In declarative, solver-based generative systems, it is a designer's job to specify *what* they want to be generated instead of *how* it should be generated. As with generate-and-test systems, this allows the generation of content for which a designer cannot imagine constructive algorithms, however it does require more critical thinking about what exactly is desired from that content. One of the primary contributions of this dissertation is a technique for building declarative, solver-based generators that capture various design spaces of interest in various

exploratory game design scenarios. See Chapter 10 for more detail.

3.5 Attempts at Formalization and Automation

Thus far, work that attempts to formalize or automate aspects of game design is still very preliminary, and, generally, the more technical the approach, the less connected it is with the textbook, best practices of game design.

3.5.1 MDA

Robin Hunicke, Marc LeBlanc, and Robert Zubeck offer the MDA (for mechanics, dynamics, and aesthetics) framework as a formal approach to game design and game research [96]. The framework defines mechanics to be the rules at the level of data representation and algorithms. Dynamics refer to how a game functions as a system including the run-time behavior of mechanics acting on player inputs over time. Aesthetics captures ideas like "fun" and other desirable emotional responses evoked in the player in reaction to gameplay.

MDA is a "lens" or "view" on game design—something that helps a designer understand their responsibility and opportunities in design as opposed to a final definition of the nature of game design. The intent of the MDA framework is to encourage new designers to formulate and track personal goals for the aesthetics of the game while mechanics are being adjusted. In an iterative design process, the framework suggests continually moving between the three levels of abstraction in response to experience gained.

To better understand the intangible levels (dynamics and aesthetics), the progenitors of this framework suggest building models (mathematical, graphical, etc.) that capture the intended functionality of the game at that level. Dormans' feedback diagrams mentioned in § 3.1.1 and Koster's and Daniel Cook's notations mentioned in § 3.3 are examples of graphical models with some support for mathematical modeling. My Ludocore system (described in Chapter 12) is an example of one way to produce a model of game dynamics that both humans and machines can reason over to gain insight in a design scenario. Ludocore models, in addition to containing details pertaining to game mechanics, may also include rich models of expected player choice, giving these models leverage for understanding dynamics.

3.5.2 A Factoring of the Concerns in Game Design

In "Towards Automated Game Design" [154], Mark Nelson and Michael Mateas suggest a description of game design as a problem solving activity. In place of reducing game design to a single problem (per the strategy of some systems described later in this section), they offer a factoring of the concerns of game design into four largely orthogonal categories: abstract mechanics, concrete presentation, thematic content, and control mapping. These categories strongly influenced the design of my BIPED system (described in Chapter 13), which was created in collaboration these authors [198].

In this factoring, abstract mechanics specify how game state evolves over time with player interaction, including goal conditions for a game. For example, the abstract mechanics might specify that your character should move through a network of rooms to collect all treasures and return them to the starting location. Alternatively, they might simply express "avoid being hit for five seconds," where the means of avoidance and the conditions for being hit are addressed in response to the other concerns below. In the sport of Tennis, abstract mechanics might refer to rules that say to increment the score when the ball bounces into certain zones, not describing how to calculate the ball's momentum or construct the court. Steenberg's concept of the "pivot state" [209] of a game is primarily concerned with abstract mechanics.

Concrete representations provide an audio-visual representation of the abstract state of the game. For example, the choice between depicting time remaining as a clock or a bar graph is a problem to be solved in this concern. This concern also addresses mechanics that directly support representations, e.g. how an abstract resource-gathering mechanic is realized in terms of a top-down 2D grid space (where tile adjacency might be important) or a first-person 3D space representation (where direct lines of sight are important).

The thematic content of a game is the body of real-world references made by the game content (e.g. visual art, story, or familiar behaviors mimicked in the game). For example, depicting an element of player state with the label "health" references the real-world concept of the health of an organism, something to be preserved (perhaps via an in-game verb that references the concept of "healing"). Thematic references set up expectations that are not coded anywhere else in a game's formal design choices, and they depend on a player's familiarity with the referent to have their desired impact.

Control mappings express the relationship between physical input devices (mice, keyboards, joysticks, etc.) and the state exposed by the abstract mechanics and concrete representation. For example, in most plat-

former games, pressing a dedicated physical button triggers the abstract jumping action that is realized concretely by adding an impulse to the player character's vertical velocity state and perhaps playing a sound.

Towards automated game design, the authors describe a system that uses AI-based automated reasoning to configure the thematic content of generated mini-games. To ensure the games would "make sense," a common-sense knowledgebase (the combination of ConceptNet [127] and WordNet [145]) was checked to make sure the objects referenced by mini-games conceptually supported the required actions. Given "pheasant" and "shoot" as input concepts, the system could reason that a "dodging" game template was applicable for instantiation with a "bullet" and "duck" sprite.

This factoring primarily speaks to concerns from the M and A in the MDA framework; it adds incremental formality to a slice of larger game design. The design of abstract mechanics is just one avenue of exploratory game design, whether undertaken for its own sake or to seek incremental novelty within the confines of established genre conventions. Thus, while this dissertation focuses most heavily on the abstract mechanics of a game and how those mechanics are enacted game content, the reader should keep in mind that this effort covers only a slice of a slice of the broader concerns of game design.

3.5.3 Ludi

In Cameron Browne's dissertation "Automated Generation and Evaluation of Recombination Games" [22], Browne proposes the idea of automatic recombination of *ludemes* ("fundamental units of play, often equivalent to a rule" [21]) as the basis for a filtering and playtest-based optimization process. His system, Ludi, makes use of general game playing (GGP) agents (discussed later in § 6.4) to test generated games. These automated players sample the space of play supported by the rules. In place of a statistically uniform sampling of play (which would likely be uninteresting), the use of strategic agents is an attempt to generate behavior closer to the realities of human play. The automated players use game-independent heuristics to search ahead by several moves, each trying to play the game as well as possible.

Later (in Chapter 14), I describe a project in which I automate search through a space of rulesets. However, instead of adopting Browne's hypothesis "that there exist fundamental (and measurable) indicators of quality" that can be computed, I opt to bring a human designer into the loop, adding and removing constraints on patterns of interest and general failure modes as they are incrementally discovered.

Though Browne is careful not to claim that ruleset optimization is an

accurate model of game design (rather, he says "game design is something of an art"), others are not so careful. Vincent Hom and Joe Marks [94] also apply a genetic algorithm to optimize over a space of rulesets using GGP for automated playtesting. They effectively define "automated game design" to mean the automatic balancing (for challenge and enjoyment) of games specifically at the rule-level (as opposed to the level of tweaking numerical parameters).

3.5.4 A Generic Framework

In perhaps the most formalized (thus directly reusable) and least practice connected (thus less productive of insight) attempt of which I am aware, Nathan Sorenson and Philippe Pasquier offer "a generic framework for automated video game level creation" [205]. Noting the common requirement of custom algorithm development in PCG systems and the ineffectiveness of standard genetic algorithms (GAs) to handle constraint-solving tasks, they describe a generic content system built around a single, carefully selected GA.

The Feasible-Infeasible Two-Population genetic algorithm (FI-2Pop) [110] is a variation on the standard GA setup to maintain separate populations of constraint-abiding and constraint-violating individuals. Infeasible individuals are judged, instead of by the normal fitness function, by the number or severity of constraints they violate. The hope is that the algorithm can use the mutations and crossovers with individuals in the infeasible population to "tunnel" across regions of the search space that variations taken only from the feasible population could not (at least not efficiently).

Where the use of a GA usually requires the design of problem-specific data structures and operations, this generic framework applies a fixed representation in terms of "design elements" (sets of typed record structures such as "*Platform(x,width,height)*") with game independent mutation and crossover operations (defined over sets of typed records). The fitness function used to evaluate feasible candidates is also game-independent, however it does make use of game-specific annotations (a scalar noting the challenge presented by each design element in isolation). Constraints, such as that there be at least a certain number of some kind of element or that there is no spatial overlap between elements of a pair of types, are expressed declaratively as part of the input to the generation system. Some constraints, such as that a generated level is "traversable," are dependent on the structure of elements in a way that narrows the scope of the framework (though both platformer and 2D adventure game examples are demonstrated).

Thus, this framework offers the ability to produce generators for new

kinds of content by specifying only non-procedural details. Applying this framework to a new type of content requires a declarative grammar of design elements, global constraints expressed in a language of six (mostly) generic templates, numerical challenge annotations, and one more auxiliary parameter used by the fitness function to evaluate the "fun" of a candidate by a flow-inspired computation over challenge values [204].

The practice of creating declaratively modeled design spaces that I develop in Chapter 10 effectively generalizes Sorenson and Pasquier's framework. With the constructs of answer set programming, a designer can declaratively define domain-specific constraints (e.g. variations on "traversable") at will as well as define alternative optimization criteria that may or may not be recognizable as proxy measures for "fun." Although designers must learn a more complex language than the above framework's constraint expressions, the designer is still shielded from the design details of the underlying generative procedure. Where Sorenson and Pasquier adopted a general-purpose combinatorial optimization procedure from the traditions of computational intelligence (CI), I chose to recycle infrastructure from symbolic AI. From the perspective of a designer utilizing either framework, however, this difference in choice between philosophies is largely unimportant.

Stepping back, Sorenson and Pasquier's "generic framework" is a technical solution for the specific problem of level generation. Towards mechanized solutions for other concerns of game design, a broader perspective must be taken on what could play the role of a design element, where constraints come from and when they there are applied, and what, if any, criteria warrant the inclusion of optimization as part of a problem definition. Such a treatment of game design in general terms is not part of standard practices in game design, at least not yet.

3.6 Game Design as a Design Discipline

It seems that most work that has engaged videogame design from a formal approach has treated it either as an idiosyncratic craft practice (with traditional techniques and domain-specific vocabularies) or as an engineering practice (reduced, in part, to rigorous analysis of alternatives by science-derived or science-inspired methods). The key philosophical turn taken in this dissertation is to consider game design from the unique and general perspective of *design thinking* [48], a perspective that is distinct from the traditions of the sciences and the humanities. The next section lays out the foundations of design as a general discipline in order to link the situation in game design with knowledge coming from other design domains such as architectural, industrial, and electronic design.

The promise for game design to learn from the current state of the art in design research recently began to be explored by others (concurrently with this dissertation). In "Some Notes on the Nature of Game Design" [118], Jussi Kuittinen and Jussi Holopainen's aim "is not to create yet another prescriptive framework for game design," referring to textbooks like *Game Design Workshop* (which prescribes the "playcentric" brand of game design methodology), "but rather to connect the game design studies to general design studies in a stimulating way." Drawing a number of the same parallels I highlight in the next chapter, they also uncover a number of biases and blind spots of the almost-exclusively playcentric philosophy common in the literature of game design.

This is not to suggest that the only fruitful interaction between game design and design studies is a kind of learning from one's elders. Game design possesses a number of unique properties and crystalizes other properties (namely interactivity) that are only present in other domains in a shadowy way. The practices that currently seem tied to games (e.g. playtesting or the creation of player models) may yet provide broad insight for our general understanding of design. However this will only happen if we recognize game design as a design discipline, able to inform and be informed by the lessons learned in other design domains.

Chapter 4

Design Studies

"Design studies" is not the name of a field, per se, but it is the name of an influential journal of research that studies the design process. The work published in *Design Studies* resides within the umbrella of design research. However, "design research" (amongst other interpretations [121]) can refer to research undertaken *within* design processes (i.e. within a particular design project) or *through* design (i.e. borrowing the practice of prototyping for use in, say, a scientific setting). Thus, I use "design studies" to refer to the work that is about design generally—the study of design processes.

Design Studies was the first academic journal of design (established in 1979), and it has published a number of the seminal articles that I cite later in this section. I have adopted the journal's statement of aims and scope[1] as the reference definition of work that I classify as design studies:

> *Design Studies* is the only journal to approach the understanding of design processes from comparisons across all domains of application, including engineering and product design, architectural and urban design, computer artefacts and systems design. It therefore provides a unique forum for the analysis, development and discussion of fundamental aspects of design activity, from cognition and methodology to values and philosophy. The journal publishes new research and scholarship concerned with the process of designing, and in principles, procedures and techniques relevant to the pedagogy of design.

In this chapter, I survey historically influential and personally relevant results of design studies. This survey is intended to situate my attempt to

[1] `http://www.journals.elsevier.com/design-studies/`

explain and automate the game design process in the context of broader attempts to explain and automate design in general. The acknowledgement that design, as a discipline, has structure and general phenomena of study is a strong motivator for many of the methods applied and theories advanced later in this dissertation.

4.1 The Relationship between Science and Design

Design studies has a relation to design similar to that which science studies (such as in the influential work of Thomas Kuhn [117]) has to science. That is, both are interdisciplinary research areas that seek to situate a particular kind of professional expertise in a broad social, historical, and philosophical context. Further, the bodies of expertise they study are overlapping: scientists design theories and experimental apparatuses while designers formulate hypotheses and carry out and analyze experiments. The relationship between science and design, between scientists and designers, is rich and multifarious; "scientific design," the "science of design," and "design science" are each interesting and *distinct* categories of study [47, chap. 7]. Touching on each of these areas, my research consistently approaches the science–design complex from the design end.

Perhaps the most obvious way science and design come into contact is *scientific design*: design using the knowledge of science (e.g. materials science or behavioral science). Scientific design, approximately synonymous with engineering, uses the data, models, and instruments of science while not necessarily using any processes that have been motivated by or tested through science. Church's formal abstract (game) design tools (from § 3.3) may someday allow a kind of scientific game design in the future. In the mean time, scientific design in games (temporarily ignoring software engineering) most often appears in the form of A/B testing or the import of science-derived models from other fields (such as the importation of flow from psychology or the use of statistical modeling in matchmaking and economic modeling in virtual world management). Currently, game designers are a long distance off from trusting their tools and theories to the same level that, say, civil engineers trust finite element analysis and the theory of stresses and strains that underlies that model.

The *science of design* is the scientific study of design and designers. Although it is not the only valid way of studying design, the science of design is in fact responsible for a significant fraction of what is known in design studies. Thus, it is natural to ask about the status of a science of game design. The scientific end of the field of game studies (anchored in psychology and sociology, e.g. Sherry Turkle's work on player identity

[224]) seems to focus on the post-deployment aspects of games: the relation between game, player, and player culture. A fitting science of game design, however, would speak to the relation between game, designer, and designer culture (perhaps seeking explanation for how *400 Project*-style rules of thumb arise).

Finally, *design science*, a term closely associated with R. Buckminster Fuller [72], is the systemization of the goals and methods of design with science-equivalent rigor. The *design method* [86] would be design's answer to the scientific method, an attempt to drive out intuitive or craft-oriented processes lacking motivation beyond pragmatism or tradition. Despite recurring attempts to systematize design in general, design studies records no dominating framework. Further, in light of ideas such as "wicked problems" (discussed shortly in § 4.3), we should no longer expect the existence of such a systemization. Nonetheless, the failed program of design science contributed immensely to the discussion of design's methods and motivations. Kreimeier's call for game design patterns (which is meeting growing responses) is evidence of an emerging game design *protoscience* (in a neutral sense [119] and as distinguished from *pseudoscience*).

4.2 Methods of Design Studies

How does one go about the work of design studies? In *Design Thinking: Understanding how Designers Think and Work*, veteran design studies researcher Nigel Cross [48] surveys five methods for researching the nature of design ability that have made impacts throughout the history of design studies.

- *Interviews with designers*: asking successful designers to reflect on how they operate
- *Observations and case studies*: retrospective or embedded analysis of processes used on particular projects in professional environments
- *Experimental studies*: running think-aloud experiments, often on artificial design projects, in a laboratory environment
- *Simulation*: capturing a design method an executable process and reflecting on the differences between results of simulated design and what is known of human design methods
- *Reflection and theorizing*: (non-empirical research methods) predicting something about design from an external theory (e.g. linking optimization in design to optimization in mathematics); applying general design theory in a specific design field (invoking the concept

of solution-focusing to explain the use of early stage sketching in architecture projects, perhaps); or the development of theory based on personal reflection, akin to interview methods, but applied to one's self

These methods are primarily derived from the scientific end of design studies (the science of design). However, as design is an interdisciplinary field, it is also possible to engage design on non-scientific terms, without the use of regularized methodologies. In fact, design studies can be engaged in design terms: as a body of knowledge to be synthesized so that this knowledge is of best use to the designers who would be informed by it. This approach, curiously called "designing designing" (in John Chris Jones' article in the very first issue of *Design Studies* [99]) or the "design of design" (in the sense of Fred Brooks [20]), is perhaps the best descriptor of the design research methods used in this dissertation. The design of design shares some goals with Fuller's design science (in the construction of a body of knowledge that is useful for designers in any domain), however it relaxes the requirement of rigor in exchange for instrumentalism: a well designed model of design is one that is productive, regardless of whether it was derived from observation of past design processes. Traditional design practices can be redesigned and replaced with new practices.

Looking at the design studies methods I apply in the context of game design, my research is indeed informed by interviews, particularly Nelson's interviews of designers in relation to their potential interactions with future design automation tools [156]. In my development of a set of design automation tools for the educational puzzle game *Refraction* (see Chapter 15), I offer results from being embedded in a design team working on a larger, ongoing project. Further, it is possible to interpret my use of computational caricature of automation in the design process as simulations; my intent is that these systems function as something other than neutral and accurate simulations (see Chapter 9). The bulk of my work, from the scientific perspective, can be seen as reflection and theorizing. However, it is better to understand the work as a series of prototypes in the design of design: partial mechanizations of game design created to elicit feedback on the true nature of game design.

4.3 Vocabulary of Design

Examining game design through the lens of design studies immediately highlights instantiations of a number of concepts that already have widely accepted names. The vocabulary of design is rich and ready to speak new insights about the nature of game design.

In 1973, Horst Rittel and Melvin Webber introduced the concept of "wicked" problems in the context of social policy planning [171]. They contrasted wicked problems, which have no definitive formulation and no ultimate test of solutions, with the "tame" problems in mathematics, playing Chess, and puzzle solving. While not all design problems are wicked problems in the exact sense Rittel and Webber defined, the specific property of "ill-definedness" (directly derived from wickedness) is widely considered to be an essential element of proper design problems. Game design, as a whole, is a wicked problem [136] in the full depth of Rittel and Webber's definition. Fortunately, many of the concrete problems that arise in particular game design projects (such as level design for a particular game) may simply be ill-defined.

Two primary strategies have arisen for addressing ill-definedness. The first is forcing definedness via a "primary generator" that limits the scope of the problem in a way that suggests solutions. This terminology is due to Jane Darke [52] as part of the very first original research article published in *Design Studies*. A primary generator often takes the form of an oversimplifying problem statement (e.g. "it seems like our goal here is to minimize fuel usage") for which a solution method is obvious (e.g. "let's rank our components by fuel usage and try to cut the worst offenders first"). Even when the solutions suggested by the primary generator are found unsatisfying (likely by pointing out a critical piece of the problem that was abstracted away), the terms set by the generator will often shape the vocabulary of future problem statements and potential solutions. The declaratively modeled design spaces described later in this dissertation (see Chapter 10) function as generators (whose purpose is first to address ill-definedness and secondarily to mechanically generate artifacts for final use).

The alternative (and complimentary) strategy is to proceed in the face of ill-definedness with a leading candidate solution in hopes of eliciting feedback that clarifies the problem and the mechanism by which the candidate solutions affect the goals of the design scenario (as they are currently understood). This strategy is a key motivation for the theory of creativity that I propose in Chapter 16. The generation of design artifacts and the gain of design knowledge are mutually dependent processes. Thus, the idea that knowledge should be gained to improve the quality of future artifacts is as valid as the idea that artifacts should be created to improve the quality of knowledge gained.[2]

Reflective practice, a concept introduced by Donald Schön [182], is

[2]If this sounds circular, that is because it is. Designers learn to make to learn to make to learn to—that either learning or making is the end goal is not nearly as interesting as the cyclic interaction between the two that constitutes the bulk of design work.

"the capacity to reflect on action so as to engage in a process of continuous learning" and "one of the defining characteristics of professional practice" (including, of course, the practice of design). Thus, the practice of design is never completely separable from the process of active learning—learning is as much a responsibility of the professional designer as is constructing the solutions they are nominally employed to create.

On "problem framing," Schön continues: "In order to formulate a design problem to be solved, the designer must frame a problematic design situation: set its boundaries, select particular things and relations for attention, and impose on the situation a coherence that guides subsequent moves" [183]. By enabling rapid development and refinement of design space models that function as Darke's primary generators, a machine's reasoning ability (usually thought to be of best use in solving problems) can be brought to bear on *framing* problems.

Making the distinction between solving the problem as currently understood and solving the design problem as a whole, Schön (with business theorist Chris Argyris [4]) distinguish between single-loop and double-loop learning. Although they originally spoke of organizations and institutions, in the context of design studies, these ideas are mapped onto designers and design teams. In single-loop learning, repeated attempts are made to solve the problem, accepting the problem as fixed (failures are blamed on the candidate solutions). In double loop learning, failure of candidate solutions may prompt a redefinition of the problem and the rejection of original goals. AI offers many methods for automating single-loop learning. Thus, although single-loop methods will fail to address problems *as* design problems, automation of single-loop learning may allow designers to more readily focus on the second loop, that of gaining knowledge that results in productive redefinitions of design problems. Towards one of my goals (demonstrating tools that respect design problems), I will consciously distinguish between the learning undertaken by design automation systems and that of the designers who employ them.

My final bit of Schön-derived vocabulary is the concept of "situational backtalk" (the key property which prototypes are designed to elicit), which follows from a model of design as a "reflective conversation with the situation" [182, p. 79]:

> In a good process of design, this conversation with the situation is reflective. In answer to the situation's back-talk, the designer reflects-in-action on the construction of the problem, the strategies of actions, or the models of the phenomena, which have been implicit in his moves.

Several of the computational systems presented in this dissertation directly aim at opening up wider channels for situational backtalk, but

they do so under the more familiar (and equivalent) terminology of *design feedback*.

In *The Sciences of the Artificial* [190], Herbert Simon (co-founder of the field of artificial intelligence as well as a major contributor to design research) conceives of an artifact as "a meeting point—an 'interface' in today's terms—between and 'inner' environment and an outer environment." "If the inner environment is appropriate to the outer environment, or vice versa," he writes, "the artifact will serve its intended purposes." Designing appropriate artifacts requires understanding the relevant mechanisms of the inner and outer environments that come into contact via the interface. Thus, formal models of design spaces (of appropriate artifacts) will likely commit to models of these mechanisms.

Simon's comments on design make reference to concepts prevalent in both AI and operations research: constraints, optimization, and means-ends analysis (planning). The Rational Model of design (misleadingly associated with and never actually advanced by Simon) defines the designer's job as optimizing a candidate artifact under known constraints, following a plan-driven process through discrete stages. As this model assumes the design problem is well-defined and well-understood, it was never adopted in design studies as a valid explanation of human design activity.

In place of performing optimization in a strict mathematical sense, Simon suggests that agents *satisfice* (conceptually, "satisfy" with "suffice" or "sacrifice") to find decisions that are "good enough" without continuing the search for better alternatives. Thus, even when an exact formulation of the value of an artifact is available, optimization is not necessarily an appropriate decision-making strategy. Focusing exclusively on satisfaction of constraints or optimization also misses chances for input from reflective practice. Satisficing, however, allows the conditions of "good enough" to depend on a designer's estimate of the effort required to find a significantly improved solution or knowledge of the relative importance of satisfying different constraints. The concept of satisficing is alive and actively studied, and it has been observed in the specific processes of many design fields [47, chap. 1].

In *Designerly Ways of Knowing* [47], Cross identifies design thinking with distinct modes of cognition that are often contrasted with science. Extending this contrast with the humanities, Cross reviews the "three cultures" view of human knowledge [47, p. 18]:

> The phenomenon of study in each culture is
> - in the sciences: the natural world
> - in the humanities: the human experience
> - in design: the artificial world [in Simon's sense]

The appropriate methods of each culture are
• in the sciences: controlled experiment, classification, analysis
• in the humanities: analogy, metaphor, evaluation
• in design: modelling, pattern-formation, synthesis

The values of each culture are
• in the sciences: objectivity, rationality, neutrality, and a concern for 'truth'
• in the humanities: subjectivity, imagination, commitment, and a concern for 'justice'
• in design: practicality, ingenuity, empathy, and a concern for 'appropriateness'

Cross summarizes the current understanding of design ability (at least as of 2006) in "resolving ill-defined problems, adopting solution-focused cognitive strategies, employing abductive or appositional thinking, and using non-verbal modelling media." Of these terms, non-verbal media is the only element I do not address further in this dissertation.[3] Cross' intent is to refer to how sketches on paper or 3D physical models (e.g. made of clay or cardboard) engage human abilities for non-verbal perception (visual, auditory, etc.). Doodles, sketches, and scrap-paper musings are a valuable part of many game designers' personal reflective practices[4] (capturing character concepts, level designs, user interface plans, etc.).

Abductive and appositional thinking (loosely, the ability to synthesize artifacts which are appropriate to a design scenario) are covered in depth in Chapter 8. This chapter untangles the complex relationship between abductive reasoning in logic (which can be readily automated) and the related sense of abduction that is tied to appositional reasoning in design.

4.3.1 Glossary

For later reference, here are concise definitions for some of the key terms emerging from design studies:

- *abductive/appositional thinking*: reasoning (logical or otherwise) to the design of an apt or appropriate artifact or explanation

[3] The adoption of solution-focused cognitive strategies, for example, is embedded in the process of building declarative design space models. After building a very rough design space model, a designer can iteratively care away specific pieces of the design space that contain easily recognizable flaws. In this way, a designer makes progress on capturing a space of interest without having to first (or potentially ever) articulate an overall explanation for their interest.

[4] `http://gamestorm.tumblr.com/`

- *appropriateness*: the property of being fit or well adapted for an intended purpose
- *artifact*: the (often human defined) interface/boundary between inner and outer environments
- *artificial world*: the domain of man-made artifacts (vs. the natural world or the human experience)
- *ill-definedness*: lacking a clear definition and implicitly any non-design means of accessing one
- *primary generator*: a problem definition which immediately suggests a space of solutions
- *problem framing/construction*: erecting a problem definition (in the sense of framing and construction of a house); primary generators frame a problem in a way that seems immediately solvable where as alternate framings might suggest the need to gather more information before proceeding
- *reflective practice*: learning while doing, particularly in a way that transforms the doing
- *satisficing*: accepting a solution that is "good enough" (despite the possibility that better solutions, even a well-defined optimum, could be eventually found)
- *single/double loop learning*: distinction between learning within the confines of a problem and learning about the nature of the problem
- *situational backtalk*: feedback gained from design practice or the interaction between candidate solution and environment
- *solution leading/focusing*: offering a solution to gain better understanding of a design problem
- *wicked problems*: problems with no definitive formulation or test of solutions (including the impossibility of ever having such a formulation or test)

4.4 The Automated Architect

Before I close this survey of design studies, I should review an important (and surprisingly early) attempt to study the way in which designers engage with alternative realizations of design automation tools. In 1967,

Cross reported on a series of experiments he called the "simulation of computer aided design" (CAD) [46]. These experiments were designed to explore the potential for machine assistance in architectural design without any particular limitations on the intelligence that might be required on the machine's part.[5]

In each experiment, a designer was tasked with a small architectural design project. Given a design brief (an informal statement of the project's goals and known constraints), the designer was asked to produce a concept sketch. The designer was given access to both conventional drawing tools as well as an "expert computer" that could be consulted via messages (in the form of text and sketches) delivered via closed-circuit television. A small team of building experts (with, for example, knowledge of building materials and constraints from the construction process) played the role of the "computer" from a nearby location. In order to explore the natural topics and protocols of conversation between designer and computer, no language or content restrictions were enforced.

In the majority of sessions, the designers reported that the computer accelerated their pace of work and reduced uncertainty in the final design (i.e. increased confidence). However, they also reported that the work was "hard" and "stressful," with the computer spotting issues with nearly every sketch they proposed. In scenarios where the role of designer and computer were reversed—it was the computer's job to produce the sketch and the designer's job to critique, amend, and refine—the designers reported no stress and even described the activity as fun.

In the forward scenario (the first), the computer was used to automate analysis, and the designer's creative synthesis was subject to criticisms from the distant machine (which was not aware of the design scenario beyond what was communicated to it via the designer). This scenario is roughly analogous to the present situation in formal methods for software engineering (e.g. model checking [6]) in which candidate designs of software systems are submitted for exhaustive analysis for known classes of bugs and compliance to a formal specification.

In the reverse scenario (where the human architect advises the automated architect) the computer was used for synthesis (subject to internally stored constraints), and the computer could proceed largely independently of corrective inputs from the designer. This is similar to the situation in Gillian Smith's Tanagra platformer level design tool [202] in which the tool was capable of synthesizing complete levels consistent with constraints expressed by a human designer, with a bias towards giving the computer initiative in design. Cross' reverse scenario is importantly dis-

[5] In today's terms, Cross was open to the idea that automated architecture might be an *AI-complete* problem. Nonetheless, he wanted to see how it would be played out in hopes that included readily automatable subproblems.

tinct from the setup of in interactive evolutionary computation (such as in *PicBreeder* [185] and *Electric Sheep* [59]) in which human input is only used as an evaluative feedback mechanism. Cross' reverse scenarios exercised a designer's ability to synthesize as well as analyze (submitting back amended drawings and newly articulated constraints).

This project to (in Cross' terms) "explore what CAD for architecture might be like" has an interesting historical context that serves to make it relevant to the present situation in game design. Just a few years after Sutherland's seminal SKETCHPAD [212] system for interactive structural drawing (the progenitor of modern CAD systems), this project sought to understand human and machine collaboration for the more abstract end of architecture, the part that is encoded in technical drawings only later in the design process. A mere decade after the founding of the field of AI, this project sought to understand the nature of communication with intelligent machines not bound by the limits of what was computable in its present day. In today's vocabulary, Cross' experiment seems to be an exploratory study in human-computer interaction designed around a "Wizard of Oz" experiment; however this kind of experiment would not be (re)invented in HCI until the 1980s [107].

The program of mechanizing game design entails exploration of what CAD for game design might be like. Although we lack the perspective to know which pre-existing system (if any) functions like SKETCHPAD in the context of game design, efforts in automation of individual facets of the design (such as those described in § 3.5) strongly demonstrate a possibility for wider automation that extends into the more abstract concerns of game design. The limitations of present-day computation can be partially sidestepped by building on representations that abstract away underlying algorithms (as part of the design space modeling practice developed in Chapter 10[6]). Finally, the artifacts-as-communication strategy, along with the focus on design spaces, provides a template for interaction between designer and machine that more closely matches Cross' reverse scenario and provides an alternative path to the analysis-heavy route currently being pursued in formal methods for software engineering.

[6]For example, during the writing of this dissertation, new back-end software system was released that allowed many of my previously created content generation systems to take advantage of multi-core machines without any modifications. Expressed as declarative definitions instead of custom algorithm designs, some of the engineering effort of others could simply be inherited.

Chapter 5

Computational Creativity

Like game design and design studies, computational creativity is a multidisciplinary affair that can be approached from several directions (philosophically, artistically, scientifically, designerly, and so on). The goals of computational creativity (CC) are multiple: to understand how machines can be creative, to understand human creativity in computational terms, and to develop computational models of artifact generation in various domains traditionally dominated by human creative effort. A mechanization of exploratory game design will indirectly speak to the first two aims, but it is best appreciated as an instance of the third.

The prehistory of computational creativity is entangled with that of artificial intelligence. In the same article that introduces the imitation game (now widely known as the Turing Test) and asks "Can a machine think?" [223], Alan Turing revives what is perhaps the first articulated opinion on computational creativity. In 1842, Ada Lovelace [131] states the following (which is widely interpreted as speaking to all machines) of Charles Babbage's Analytical Engine:

> The Analytical Engine has no pretensions to originate anything. It can do whatever we know how to order it to perform.

Her intent, it seems, is to suggest that if we ever saw a machine originate something (an idea, an artifact, etc.) we should immediately shift our credit or blame to the person who ordered it to do so. In the context of constructive artifact generators (where the space of output artifacts is defined by a strict, imperative recipe for construction without need for exploration or reflection), this perspective would seem appropriate. However, even outside of exotic architectures for generative systems, where the steps to be performed either depend on what is uncovered in search (e.g. in evolutionary systems) or are nowhere specified in the "order" (e.g.

in declarative, solver-based systems), we can quite easily create machines that (even in Turing's time) "take us by surprise."[1] The most exciting systems in computational creativity, I claim, are the ones that allow us to order the origination of something via steps that we either do not know how to perform or do not know how to articulate.

In the rest of this chapter, I review some of the vocabulary of computational creativity literature, sketch the ways in which game design (with which CC has only had sparse contact) offers new perspectives on CC, and outline the way this dissertation engages with creativity (computational and otherwise).

5.1 Vocabulary of Creativity

The vocabulary of computational creativity has no strict boundary with that of general creativity as philosophers, psychologists, sociologists, and others approach it. Thus, I focus my review on those terms most needed in my program of mechanizing exploratory game design.

5.1.1 "Creativity"

"Creativity" has no simple, universally accepted definition, and the same goes for "computational creativity." Computational creativity research proceeds without a unifying definition for creativity in the same way that artificial intelligence research proceeds without a unifying definition for intelligence. It is the regular cycle of exploration for attempts at defining creativity to prompt new perspectives on domain-specific instances of creativity. The techniques and viewpoints abstracted from those instances then inform future attempts at general theories. Progress is made by developing more broadly informed ideas at all scales and through contact with new and diverse domains.

5.1.2 Novelty, Value, and Surprise

Offered as necessary and sufficient conditions for creativity, three general concepts underlie Margaret Boden's [14] influential definition[2] of creativity: "Creativity is the ability to generate ideas or concepts that are novel, valuable, and surprising."

[1] Turing is paraphrasing Lovelace in this quote. He continues: *Machines take me by surprise with great frequency. This is largely because I do not do sufficient calculation to decide what to expect them to do, or rather because, although I do a calculation, I do it in a hurried, slipshod fashion, taking risks.* [223]

[2] This is Boden's working definition of creativity for the purposes of her book, not a final definition that attempts to put the question of "What is creativity?" to rest.

Novelty demands that the idea or concept (from now on, the artifact) is not simply a copy or a trivial modification of a previous artifact. An artifact can be mildly novel if it differs from others in an easily describable way (perhaps in a difference of a numerical parameter), or it can be wildly novel if it so different as to be incomparable to familiar artifacts (i.e. not representable in familiar vocabularies). In Boden's terms, artifacts that appear novel to an individual may be only *P-creative* (for psychological or personal creativity) whereas artifacts that are novel for an entire society may be *H-creative* (for historical creativity). Without reference to individuals or societies, Graeme Ritchie [170] judges the novelty of a creative system's output with respect to the contents of the *inspiring set*, "the set of all items used to guide the programmer in the construction of her program." The *Robot Scientist* (a room-sized robot that automates an impressive range of activity in functional genomics research) [111] and *HR* (a mathematical discovery system designed for use in number theory that has since been applied more broadly) [36] are two H-creative computational systems that yield results appreciably beyond their inspiring sets. For the purposes of this dissertation, I am most interested in producing systems that allow designers to navigate H-creative territory in game design.

Value demands that generated artifacts be good for something: they must have monetary value, aesthetic value, utilitarian fitness for a purpose (function), etc. Harold Cohen's *AARON* system [32] produced paintings that are widely exhibited in galleries and major museums. Scott Draves' *Electric Sheep* [59] is a well known animated visual art generator that functions as a distributed interactive genetic algorithm across hundreds of thousands of machines and people. Stepping away from high art to personalized consumer products, *thevibe*[3] is a recent system (and revenue-bearing business, producing monetary value) that turns waveforms from user-selected sound files into a 3D-printed, protective case for the user's mobile phone. In this dissertation, I intend to produce value for the game designer (who may value time saved in playtesting or content creation, insight derived from seeing demonstrations of flaws, or the assurance of a machine-checked second opinion on the absence of shortcuts in a puzzle).

Surprise is the requirement that novelty and value arise unexpectedly (whether informally or in a statistically rigorous sense of expectation). When hard work, perhaps in an exhaustive analysis of combinations, yields artifacts with novelty and value, the judgment of creativity is blocked by a lack of surprise. Surprise results when novelty is found in what is thought to be a small or well-understood space or when value is located in a large space thought to be mostly filled with uninterest-

[3] `http://www.shapeways.com/creator/thevibe`

ing junk. When exhaustive analysis is hidden away and automated (as with the design space models developed in Chapter 10), insight-producing surprise becomes more reachable.

5.1.3 Process vs. Product vs. Producer

In the early seventeenth century, Johannes Kepler discovered three novel, valuable, and surprisingly simple mathematical laws describing the motion of the planets around the Sun. History records this as a creative emergence, but are the laws creative? Was Kepler's analysis technique creative? Was Kepler creative? In 1981, Langley reported that the BACON system, a computational model of qualitative equation discovery, recreated one of Kepler's laws when given access to the same data set that Kepler used [120]. Is BACON creative?

No consensus has been reached on whether creativity lies in the process used, the end product, or in the (often human) producer. Nonetheless, particular models of creativity will be defined in terms of particular elements. For example, Ritchie's set-theoretic formulation of the potential creativity of software programs (processes) is defined strictly in terms of properties of the artifacts (products) it produces [170]. By contrast, the model of creativity that I advance in Chapter 16 speaks to the creativity of producers (by examination of their process).

The technical practice of developing design space models described in Chapter 10 focuses on automatically producing artifacts in a way that shields a designer from the underlying generative process. The intent is that the artifacts generated (game structures such as rule sets and level designs or gameplay traces) are novel and valuable, but I make no claims as to whether the hidden process implements a creative agent or not. In the terminology of double-loop learning, creativity in the inner loop (say, the creation of a level design subject to rigid constraints) is neither required nor important, so long as insight that drives the outer loop is produced. After all, my goal is to expand a human game designer's creativity.

5.1.4 Combinational, Exploratory, and Transformational Creativity

Boden [14] names three classes of creative activity on the basis of how these activities engage pre-existing conceptual spaces:[4] combinational,

[4]Boden's concept of a "conceptual space" is nowhere rigidly defined. The ambiguity in what counts as inside or outside of a conceptual space serves to soften and make subjective the boundaries between combinational, exploratory, and transformational creativity.

exploratory, and transformational creativity.

Combinational creativity involves the unexpected combination of two familiar ideas into a novel and valuable result. The essence of combinational creativity is that the result is reached in a single step. Humorous two-word phrases can be found with combinations of tokens from a magnetic poetry set, and poignant observations can be captured in a political cartoon by the juxtaposition of object and label. Combinational creativity can occur at many levels of detail. In game design examples, adding wings to a Koopa Troopa, in the *Mario* universe, yields the Koopa Paratroopa (a relatively small change), and replacing the fantasy theme in *Warcraft* with a science fiction theme results in *StarCraft* (a sweeping change). Small-scale combinations need to be embedded in a larger context to gain value and large-scale combinations need to be supported by coherent details at a smaller scale. Although the formation of combinations for *well-defined* conceptual spaces is relatively easy to automate, the challenge in judging novelty and value in candidate results for this form of creativity is in no way lessened.

In Boden's class of *exploratory* creativity, several steps must be taken within a conceptual space to yield a result. Boden's classic example involves exploring all of the places in a countryside while staying on well-marked roads. Many locations may be well away from heavily traveled routes, but any route taken can always be traced on a roadmap. In a more formal setting, exploratory creativity might involve exploring any combinations of a set of predefined elements (exploring the powerset) or any pattern of productions in a formal grammar. The novelty and value of any result is the product of several interacting choices that together define an idea or artifact. Linking back to design studies, exploratory creativity is best identified with single-loop learning: exploration in which the bounds of the problem are accepted as fixed. In game design, most low-level design tasks (e.g. assembling a level design from a library of pre-defined tiles) are within the purview of exploratory creativity.

It bears mentioning that the sense of *exploratory* game design that I seek to mechanize is distinct from Boden's sense of exploration within a given conceptual space. I am interested in helping designers explore in order to gain previously inaccessible knowledge whereas Boden's term would have them explore in order to catalog the contents of a previously delimited territory. The sense of exploratory game design I am most interested in best maps to Boden's final class.

Transformational creativity is an activity (easily identified with the outer loop of double-loop learning) that overturns previously accepted limitations on style and stated goals of exploration. In the driving scenario, transformational creativity might involve getting out of the car to follow a trail on foot or boarding a plane to leave the countryside alto-

gether. The results of transformational creativity are simply inexpressible in the language of previously established conceptual spaces. Transformational creativity is neither magical nor impossible, but it does require the ability to reason across different spaces and to forge the rules of a new space. In game design, transformational creativity can take the form of the introduction of new mechanics that are not already represented in an established game genre. New mechanics (composed of rules) allow completely new trajectories for gameplay and must be supported by new kinds of content. Transformational creativity is difficult to achieve, let alone sustain, because, with each transformation, large swaths of past experience that would drive informed decision-making are made obsolete. Sustaining transformational creativity requires a means to quickly master each new conceptual space encountered.

Transformational creativity is the most prized form of creativity; however, the human creative reach can be expanded by acceleration at any of these levels. The technical methods advanced in this dissertation primarily focus on using machines to carry out kinds of (perhaps P-creative) exploratory creativity where feedback from automated exploration will fuel designers to carry out (ideally H-creative) transformational creativity by their own processes. That is, from the perspective of Boden's three classes, the essential idea of this dissertation is to produce formal models of the otherwise informal conceptual spaces.

5.1.5 Closed Loops

Computational creativity overlaps with procedural content generation in the desire to create systems that autonomously generate artifacts for use in games. However, of the various architectures used in PCG systems, the ones that resonate with our understanding of human psychological processes have a preferred status in CC. Thus, while constructive (feed-forward or pipelined) processes may reveal interesting structures and requirements for particular game content domains, these software processes are never taken as a model for the psychological processes underlying general creativity.

The "generate and test" terminology, common in PCG, is also pervasive in CC (e.g. in Yu-Tung Liu's "dual generate-and-test" model of social creativity [128][5]). As a result, many of the systems demonstrated in a CC context often employ a generate-and-test architecture at the code level. In the context of scientific creativity (specifically predatory ecology), the

[5]Liu's model was advanced in a *Design Studies* article and derives from the ideas in Simon's *Sciences of the Artificial* [190], nicely linking computational creativity back to design studies.

LAGRAMGE[6] equational discovery system [61], employed a grammar-based generator to propose biologically plausible equations of species population over time and a model-fitting module to determine the error of the best numerical fit between the candidate equations and reference data. In an artistic context, the visual art subsystems of *Tableau Machine* [200], an interactive installation piece to which I made large design contributions, employed a generator based on shape grammars to amass a library of pictures with known structural motifs and a separate pixel-based analysis to judge emergent properties of the overall composition.

When referencing "generate and test" in relation to human creative processes, however, the idea of batch generation and filtering processes is usually not the intended target. Instead, in describing "the central feedback loop of creativity," Gary McGraw and Douglas Hofstadter [141] refer to "an iterative process of guesswork and evaluation." While this process does involve some generation and testing, the idea targeted has more to do with a closed loop of action and perception. This loop is better identified with Schön's reflective practice or the outer loop of double-loop learning than the pervasive generate-and-test software architecture. While the contingent generation feature of evolutionary generation systems does form a closed loop, this loop (like the inner loop of double-loop learning) works within a fixed problem definition. The ability to change the problem requires a certain amount of fluidity or "slippage" (in Hofstadter's terms [92]) not afforded by the genetic representations of most evolutionary systems.

Transformational or not, closed loops are a common motif in formal models of creativity. Loops may occur within single agents, as the case of Kathryn Merrick and Mary Lou Maher's curious sheep [142] simulation or *AM* (the automated mathematician), Doug Lenat's seminal discovery system [124]. Loops may also occur within societies of communicating agents, as in Rob Saunder's *Digital Clockwork Muse* [178] or David H. Feldman's *DIFI* model (delineating **i**nteractions between **d**omains of knowledge, **i**ndividual creators, and **f**ields of adopters) [70].

[6] LAGRA*m*GE is the awkwardly spelled successor to Dzeroski and Todorovski's earlier equational discovery system, LAGRANGE. This contrivance allowed GRAM as a substring in the newer system's name, highlighting its use of grammars in the proposal equations guided by a background theory. Seriously: `http://www-ai.ijs.si/~ljupco/ed/lagrange.html`. Following long-established AI traditions, many of my own systems are named according to shallow puns and set in smallcaps without regard for fitting an appropriate acronym.

5.2 Computational Creativity Meets Game Design

So far, CC has most heavily focused on creative domains with relatively self-contained artifacts like static pictures and symbolic equations for which a numerical valuation seems feasible if not fair (e.g. in the case of equations). While these artifacts can be reinterpreted in different contexts to yield different judgments of creativity, they are relatively easy to represent in a format that makes their most salient properties apparent. Game design, by comparison, is about crafting spaces of play; games are devices that afford some kinds of interaction but not others. The creativity of a game design has much more to do with what players can do with it than what novelty and value can be found in its representation as a collection of rules and content. Although game design touches on some traditional domains of CC (e.g. games often employ visual art and music), the core of game design involves a unique constellation of concerns that makes exploring CC in game design a worthwhile project in advancing our general understanding of creativity. Below, I discuss three issues that are brought into clear focus by an examination of game design.

Many forms of art involve artifacts that cannot be judged in a single instant. For example, music necessarily unfolds in time, making and breaking several expectations before the piece ends. Similarly, sculpture unfolds spatially, yielding different snapshot images from different viewpoints and continua in between. Gameplay has a unique branching-time sense of unfolding: a play experience is judged not only by how it happened to unfold in this instance, but also by how alternative choices (perhaps those to be attempted in replay) are considered to have fared. A game that reliably produces a positive emotional experience for its players may still represent an utter failure for a designer if the designer knows of systematically ignored possibilities for preferred alternative trajectories of play that branch off of those regularly observed. A soft sense of branching-time is present for other art forms (e.g. we can imagine what happens to characters in a fixed narrative if things had gone differently), however games make these alternatives into symbolically reified and physically realizable trajectories via replay.

Examined from another perspective, interactivity is a property that often goes underexplored in CC research. Games are not the only interactive art form nor are they the only form to involve the construction of interactive machines. Consider architecture: Le Corbusier is often quoted as saying "the house is a machine for living in" [122, p. 107]. The function of a building depends on its use by its inhabitants. A building can be used, misused, and reused for new purposes in the same way a game

can be played, cheated, and metagamed. Architecture is another domain that is dominated by indirect design: the goal is to produce an artifact with which others (who may themselves be creative agents with their own expressive goals) can assemble a desirable experience.

Play within rigid rules is not unheard of within the study of creativity—there is surely a continuum including both the Oulipian constrained writing exercises and the grammar-driven generators inside the poetry and narrative systems of CC. However, a computational account of the design of these rule systems, with an eye towards the play they afford, seems completely unexplored (outside of those efforts to formalize game design, such as those mentioned in § 3.5).

These game-anchored concerns do not make game design dominate creativity in other forms of art, but they do highlight explicit areas that future, universal explanations of creativity *must* address.

5.3 Mechanization and Computational Creativity

The discussion above has addressed only a fraction of the big ideas in computational creativity, but these are enough to articulate the goals of this dissertation in standard terms. By anchoring this dissertation in CC, I hope to motivate the particular research practices I adopt as well as motivate the evaluation I perform. A theme of each of the following items is that a program to mechanize exploratory game design (to turn it into a practice where machinery can replace some human labor normally required to prepare new territories for development) is different from a program of generally automating game design (replacing human creativity by machine creativity).

With respect to *novelty*, recall it is my goal to extend our creative reach in game design. Part of this is enabling the creation of new kinds of games that would not have existed otherwise. An evaluation of design assistance tools that shows the design of one new game (or any number) would be inconclusive, as we would expect human creativity to eventually reach any particular design eventually (perhaps by expensively delaying development of the critical play-time design automation system until it became absolutely required). Instead, what is important to examine is the extra burden incurred by designing games outside of familiar territory. Is innovation less risky than before mechanization? Can *transformations* to a conceptual space be evaluated more directly than before via some new instrument?

As exploratory game design explores unfamiliar territory in game design, the most *valuable* artifacts to generate are those that refine a de-

signer's understanding, making the new territory familiar. This sense of value is quite distinct from that used in designing games to evoke a particular aesthetic response in play. Seeking value for the designer, can we mechanically identify gaps between a game as-imagined, as-constructed, and as-played? Is the kind of backtalk elicited by evidence of these gaps productive for refinements of understanding and suggestive of follow-up exploration?

Examining an exploratory game designer's *closed-loop* creative process, there are clearly cycles of action and perception—textbook documentation of the game design typically describes an iterative process. However, textbook definitions do not speak to the low-level actions of game design: how to form a modification to an existing rule, how to choose the next move in a self-test while keeping a particular play style in mind, or how to place items and objectives in a level design to bring about a particular kind of challenge. A mechanization of game design involves machine-level descriptions of these critical but understudied subprocesses. Are there other subprocesses that go unnoticed without a mechanization of their surroundings? In mechanizing one process of exploratory game design, is another revealed to be a bottleneck in a designer's discovery process? Once identified, can this bottleneck be eliminated? Which loops in game design are readily mechanized with the technology available today, and which are blocked on significant improvements in artificial intelligence?

Even though my goals are aimed decidedly short of automating game design in general, it is instructive to examine the challenges general game design automation might face. Whether by bottom-up generalizations of procedural content generation or by top-down operationalization of best practices in game design, the broader automation of game design is on the agenda for many. Of a variety of systems designed with the aim of automating some form of creativity, Bruce Buchanan writes the following in his 2000 presidential address [25] for the then American Association for Artificial Intelligence (AAAI):

> (1) They do not accumulate experience, and, thus, cannot reason about it; (2) they work within fixed frameworks, including fixed assumptions, methods, and criteria for success; and (3) they lack the means to transfer concepts and methods from one program to another.

Responding to Buchanan's call for accumulation, reflection, and transfer, my goal of demonstrating tools that respect design problems exactly aims at preparing the robust toolset that future game design automation systems will require. Without needing to solve all of creativity first, I can still make systems that (1) systematically generate informative feedback

from which designers can learn and reason over; (2) devise representations for design automation systems that are focused on rapidly exploring an open-ended variety of working assumptions; and (3) develop domain-independent design practices backed by symbolic AI tools that *do* transfer from one domain to another. It is still the designer's responsibility to reflect-in-action; however, powerful tools that are well aligned with the goals of design will empower them to be more creative.

Chapter 6

Symbolic AI

The goals of artificial intelligence parallel those of computational creativity. AI seeks to understand how machines can be intelligent, understand human intelligence in computational terms, and demonstrate models of intelligent behavior in the contexts traditionally dominated by human intellect. With upwards of half a century of earnest academic and industrial research, there is now a wide array of freely available machinery ready to be reused in the service of developing new AI applications.

Symbolic AI is an approach to AI founded on symbolic logic, or, more broadly, the reflection of human cognitive processes into rules for the syntactic manipulation of symbols. Synonymous with classical AI, symbolic AI emerged during a time before many of the distinct areas of study in modern day computer science separated and stabilized. As a result, many ideas from early explorations with the symbolic AI approach are now to be found in the foundations of several fields that retain no obvious links to symbolic logic.

Simon, Newell, and McCarthy are three names closely associated with symbolic AI. Herbert Simon (1916–2001) (whose impact on the nascent field of design studies I have already reviewed) was consumed with the simulation and systemization of human individual and organizational decision-making. His terminology including "satisficing" and "bounded rationality" found broad adoption in economics, psychology and sociology. Simon's student, Allen Newell (1927–1992) (whose "knowledge level" concept plays an important role in Chapter 16), pioneered the program of implementing the symbol manipulation rules of symbolic logic as a software program (Newell and Simon 1956), making possible many of the mechanisms underlying databases (e.g. query planning) and programming languages (e.g. type checking). Developing a broader vocabulary of symbolic models of human intelligence, Newell's GOMS model (for

Goals, Operators, Methods, and Selection rules) effectively founded the field of human computer interaction (HCI) (Card, Moran, and Newell 1983). Finally, John McCarthy (1927–2011), designer of early Chess-playing programs and originator of the term "artificial intelligence," was concerned with how to express high-level ideas in a machine understandable form. Inspired by Alonzo Church's lambda calculus (a formal system in mathematical logic for describing computation), McCarthy's Lisp is perhaps the first declarative programming language, and it introduced both the foundational concepts of functional programming systems (e.g. garbage collection) and a macro system that promoted the development of domain-specific programming languages. McCarthy also campaigned for representation formalisms that are "elaboration tolerant." Elaboration tolerance is "the ability to accept changes to a person's or computer program's representation of facts about a subject without having to start all over" or do "surgery" on the formula [140]. Ideally, clarifications to a formal model can be accomplished with the addition of new facts rather than re-engineering previous definitions.

These individuals' contributions suggest that developing symbolic models of otherwise informal human behaviors, using machines to operationalize these models, and engineering declarative programming languages are powerful research methods that can have radical and rippling impact on fields seemingly unrelated to symbolic AI. The same methods are applied in this dissertation, where I hope to have a transformative effect on game design.

In the rest of this chapter, I review logic programming, summarize some canonical problems in symbolic AI and common elements of their mechanized solutions, and situate symbolic AI's most significant interaction with games in the past: general game playing. Finally, I briefly review Newell's *knowledge level*, a way of talking about humans and machines without making reference to their symbol-level system implementation details.

6.1 Logic Programming

Logic programming is a programming language paradigm (a peer to, for example, functional programming and object-oriented programming) that is based on first-order logic (also known as predicate calculus). As such, my review of the basic elements of logic programming also serves as a review of the relevant parts of formal logic needed to understand the technical content presented later in this dissertation.

Formal logic attempts to set up an equivalence between the truth of statements made in a natural language (such as English) and the mechan-

ical provability of statements in a formal notation. For consistency with the bulk of programming-oriented discussion presented later, I adopt a programmer-oriented notation in this section in place of the mathematics-derived notation often used in textbooks of formal logic.[1] Consider these simple statements:

> Socrates is a man.
> All men are mortal.
> Therefore, Socrates is mortal.

In a Prolog-like syntax (which is shared by a large number of logic programming languages), the first statement becomes what is called a *fact*: `man(socrates).` In this statement, `socrates` is a symbol, a syntactic token that we intend to reference a real-world object (Socrates, the Greek), and `man` is a predicate, a property that objects may or may not have (humanness, in this case). Without any other reference to `man` or `socrates`, `man(socrates)` has as much meaning as `prop47(obj3)`—it is true that some object has some property. Conventionally, numbers and general character strings also make suitable object identifiers, along with compound terms made from symbols with parameters (e.g. `nth_disciple('Socrates of Athens',1)` is a verbose replacement for `plato`). The name of a predicate, however, must always be a symbolic atom.

The second statement speaks about a property of many objects. It maps to a *rule*: `mortal(Obj) :- man(Obj).` (implicitly quantified over all objects that might take the place of `Obj`—note the use of a capital letter at the start of the variable name). The rule says that we can prove the `mortal` property of an object if we can first prove the same object has the `man` property. It should be not surprising that, if a logic program contains the fact from above along with this rule, the meaning of the program is the same as if we had simply written the additional fact `mortal(socrates).`

Rules enable proofs, and proofs allow us to speak of the truth of propositions (made from objects and predicates). When a rule involves the `not` operator, the sense of truth for the following proposition becomes inverted. The introduction of negation into a logic program allows for an interesting and very useful form of underdetermination. Consider this program providing rules for determining the truth of two propositions (effectively predicates without parameters) `p` and `q`:

```
p :- not q.
q :- not p.
```

[1] This is an intentional choice made to reduce the body of knowledge required by future users of logic programming. Just as Lisp can be concisely introduced in its own terms without invoking the lambda calculus, it is possible to build a workable foundation for logic programming without leaving the ASCII table (without $\forall$, $\exists$, $\neg$, $\wedge$, $\vee$, and $\rightarrow$).

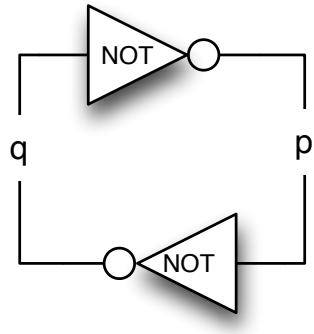

Figure 6.1: This Boolean circuit is *metastable*: it may nondeterministically settle into either one of two states (p is true and q is false or vice versa).

This program has two *stable models* (assignments of truth value for each proposition). In the first, `p` is true and `q` is not. In the second, `q` is true and `p` is not. This program is equivalent to the Boolean circuit in Figure 6.1 where logical negation becomes an inverter gate, propositions become named wires, and the process of logical inference becomes the process of value propagation along wires and through gates. In an extended analogy between logic programs and boolean circuits, a proposition is provable if it corresponds to a wire that can be determined to have a true value by propagating given values from other parts of the circuit. If the circuit is metastable (i.e. there are multiple states it might settle into), the circuit's stable states exactly correspond to the program's stable models.

There are three broad classes of logical reasoning: *deduction*, *induction*, and *abduction*. Each is associated with a type of logic programming system that mechanizes that form of reasoning.

6.1.1 Deductive Logic Programming

In *deductive* logic programming (the most common form), a programmer provides a body of facts and rules and asks whether a certain proposition, called the *goal*, is true. Starting from the fact and rule from the mortality scenario above, we can ask "*Is* Socrates mortal?" with the Prolog expression "`?- mortal(socrates).`" We can think of this as querying the truth value of a particular proposition, but the mechanism used is more general.

We can also ask "*Who* is mortal?" by including a variable in place of a specific object identifier: "`?- mortal(Obj).`" Trying this example in most Prolog engines yields an answer `yes` with a concrete result of `man(socrates)`. That is, given a general pattern as a goal, the Prolog engine seeks to find proofs of particular instantiations of that pattern. More solutions, if there are any, can be had by forcing the engine to backtrack and search for alternative proofs. The result, of course, is not always `yes`: "`?- mortal(plato).`" quickly yields `no` because we have not mentioned that Plato was a man nor have we expressed any other sufficient conditions for deriving mortality.

In more general terms, given rules for deriving effects from causes and facts asserting the presence of particular causes, deductive logic programming systems mechanize the derivation of the implied effects. Given rules describing the mechanics of a game and facts describing a sequence of player inputs, mechanized deduction can determine if the player's input achieves some goal in the game world.

6.1.2 Inductive Logic Programming

Where deductive logic programming is about deducing the truth of a goal pattern, *inductive* logic programming is about inducing the structure of a new rule from data. Specifically, inductive logic programming engines (such as Progol [153]) have the job of constructing a rule so that a set of required deductions can be made once that rule is added to a background program. Inductive logic programming (ILP) implements a kind of machine learning that shades into statistical-relational learning (SRL) as probabilities are introduced [108].

In the mortality scenario, suppose we are not given the rule for deriving mortality and must induce it from examples. In the training set used as input, *instances* (sets of facts that describe different scenarios) are paired with *labels* (facts involving the target predicate to be learned). Here is the program representing our training set:

```
man(socrates).
mortal(socrates).
man(plato).
mortal(plato).
-mortal(zeus). % zeus is not mortal
```

Asking our ILP system to induce a rule for the *mortal* predicate, the system considers simple rules before considering more complex rules. The rule "`mortal(X).`"—that every object is mortal—correctly classifies Socrates and Plato, but incorrectly predicts the label for Zeus. Adding a single term (whose structure is taken from the example data), the program tries the rule "`mortal(X) :- man(X).`" As this rule has a tiny cost (one clause with one body term) and perfect predictive accuracy, the system outputs this as the solution to the induction problem.

Realistic ILP applications involve learning much more complex concepts, often inventing several predicates simultaneously [54]. Further, many ILP systems can induce rules that trade off predictive accuracy in exchange for simpler structure of the output rules. Where appropriate, additional knowledge can be provided that restricts the space of rule structures to those that are plausible in the application domain (often greatly speeding up search as well) [152].

Returning to general terms, given a set of facts describing observed causes and effects, inductive logic programming mechanizes the proposal of general rules that would explain the observed effects from the recorded causes. Applied in game design, given a complete dump of a game's state over time along with facts describing the player's input, we might conjecture that the game causes the player to tend towards play by simple (but unobservable) rules. Applying ILP, we can mechanize the proposal of data-consistent models of player decision-making (often simply called "player models" [193]). In an inductive player modeling application, providing an appropriate inductive bias over the space of predictive rules to be considered is a difficult design problem (but an approachable one).

6.1.3 Abductive Logic Programming

When the mechanism linking causes to effects is known (specified by rules), and specific effects are observed, *abductive* logic programming (ALP) mechanizes the proposal of specific unobserved causes that, if assumed, would explain the observations.

Knowing that all men are mortal, and learning that someone named Xenophon is also mortal, we might reason that this is because Xenophon is a man. If we know of another means of becoming mortal, perhaps being a god who sufficiently angers other gods to the point of having immortality revoked, then an alternative explanation is also possible. However this second explanation (that Xenophon is a fallen god) seems less likely to be the case.

In a murder mystery scenario, abductive reasoning is the mode of thinking that allows us to derive the means, motive, and opportunity for a murder from the evidence provided. The "deduction" associated with the famed fictional detective Sherlock Holmes is, in fact, abduction. When you have eliminated the impossible, whatever remains, however improbable, *may* be the truth.

Eliminating the impossible is done by the mechanism of *integrity constraints* in ALP. One integrity constraint might tell us that a person can only be in one place at a time (or, written in terms of a violation: it is impossible for someone to be in two different places at the same time). In this way, if we gather evidence placing someone at one location at the

time of a murder, we know not to abduce the explanation that they were also at the scene of the crime at the very same time.

Like inductive reasoning, abductive reasoning is *unsound*: it can produce results that are not true. Of those results that are consistent with observations, Occam's razor tells us that the simplest explanation should be considered the most likely. As such, ALP systems (such as PROLOGICA [167]) make sure to only invent explanatory facts that have a causal relationship to the observations. Further, many systems expose a weighting system that allows a numerical cost to be associated with potential explanations. Optimizing this cost metric yields preferred explanations.

When the structure of potential predictive rules can be captured with a simple set of facts, it is trivial to reuse ALP systems to implement ILP systems. All that is needed is to pose the question "what is the structure, if interpreted as a predictive rule, that would correctly classify all of the training examples (and not violate any integrity constraints)?" (where the weighting system is used to capture the complexity cost of the predictive rule and the cost of mispredictions). Further, when the space of explanatory facts and the cost metric is well defined, it is possible to reuse deductive logic programming systems to implement ALP systems with a goal like "`?- optimal_explanation(Explanation,Cost).`" This reduction to deductive logic programming is utilized in nearly all ILP systems. For example, ALEPH [207] is a framework for producing heavily customized ILP systems on top of Prolog.

6.1.4 Prolog

Although Prolog is not the only deductive logic programming language (Curry [89] is an interesting Haskell-like alternative to Prolog), it is, by far, the most widely known. As such, it is unfortunately common for statements made about Prolog to be interpreted as if they were true of all logic programming languages (deductive or otherwise).

Prolog engines (such as SWI-Prolog [228] or tuProlog [56]) have, at their core, search algorithms. These algorithms, often based on SLDNF,[2] implement a depth-first search process through the space of possible proofs. Starting with the goal term, Prolog engines chain backwards through rule definitions to reach facts that support them. When a goal term cannot be matched with any fact or rule that might be used to prove it, the search process backtracks. Because of the depth-first nature of the search process and the likely presence of recursion in the rules provided by the programmer, it is quite easy to get Prolog engines trapped in infi-

[2]SLDNF stands for Selective Linear Definite clause resolution with Negation as Failure [3], but don't let that scare you away. In effect, it means your code is run top-to-bottom, left-to-right, chasing down unfamiliar terms as they are encountered.

nite loops. Even when the rules of a program appear to have the correct logical reading, whether the program diverges or not greatly depends on the order of rules in a program and the order of terms within a rule.

The possibility for program divergence, while undesirable from a logical standpoint, is a symptom of a useful property of Prolog: it is a Turing-complete programming language. Judging whether an arbitrary Prolog program will diverge is undecideable (this is just the Halting Problem in poor disguise). In practice, Prolog programmers memorize a collection of safe patterns that can be applied to write (hopefully) convergent programs [210].

6.1.5 Constraint Logic Programming

Though any imaginable program can be expressed in vanilla Prolog (by virtue of its Turing-completeness), constraint logic programming extends Prolog with specialized constraint solving algorithms (such as the simplex algorithm [51] for solving linear programming problems). Constraint logic programming (CLP) allows programmers to express the subset of their problem that involves, for example, numerical constraints with a more natural notation and gain performance benefits from the use of problem-specific solvers within the overall depth-first proof construction search.

The CLP(Q,R) library [93], allows the Prolog programmer to write natural-looking constraints over expressions of real-valued numerical variables. The algebraic expression "`A + 5 < cos(X)`" is treated as a single proposition from the logical perspective (the less-than relation either holds or it does not). During search, this constraint term is posted to or dropped from the constraint solver every time the proposition is assumed true or the choice is backtracked. The constraint solver can block proof construction if a set of mutually inconsistent constraints is ever posted to it. Finally, before the solution is displayed, the solver can be asked to generate concrete numbers that can be plugged in for the `A` and `X` variables that witness the satisfaction of the constraint.

It is possible to think of constraint solving in abductive terms: explaining what values should be assigned to these variables so that all of the constraint expressions yield true and the objective function is maximized (the desired observation). However, thinking of variables-to-be-assigned is actually a far more general perspective that can equally well describe the non-numerical parts of CLP.

6.1.6 Answer Set Programming

Answer set programming (ASP) is a programming language paradigm[3] that overlaps each of the categories mentioned above. Although answer set programs can be read as first-order logic statements and are usually expressed in a syntax (called AnsProlog [7]) that bears a strong resemblance to Prolog, ASP is based on fundamentally different foundations than the Prolog-derived traditions of logic programming. Where Prolog is *proof-theoretic* in nature—it is oriented around building *proofs* of goals (often via backwards-chaining)—ASP is decidedly *model-theoretic* in its foundation—it is concerned with enumerating the stable *models* of a logic program. When applied to developing descriptions of design spaces (as in Chapter 10), having the meaning of programs tied to models (which are directly interpretable as artifact descriptions) instead of proofs (which might be interpreted only as ordered justification of why an implicitly specified artifact is appropriate) provides a better mental model for programmers to adopt.

ASP systems, called answer set solvers, do not search over the programmer-provided rules directly. The program is first syntactically expanded (i.e. compiled) into a set of (mostly) Boolean formulae in a process called *instantiation* or *grounding*. Each proposition in the grounded problem becomes a Boolean *variable*: something to be assigned true or false in a particular model. Rules become *constraints* over these variables, forcing the conclusion of a rule to be true if each of its premises is true. The problem emerging from the grounding phase resembles a SAT (Boolean satisfiability) problem. The problem of enumerating the program's answer sets (satisfying assignments to the Boolean variables) is solved by a high-performance combinatorial search algorithm for finite, but large search spaces. The nature of the underlying search algorithm, as in constraint logic programming, is hidden from the programmer and can be swapped out without altering the semantics of user-crafted answer set programs.

As a result of its alternative foundations, ASP is not subject to many of Prolog's noted failings. The semantics for conforming search algorithms ensure that the search phase of ASP always terminates in finite time and that, if there is a solution (an answer set), it will be found before termination. Further, the results of an answer set program are insensitive to the ordering of rules in a program or the ordering of terms within a

[3]Whether "answer set programming" is a programming language, a programming paradigm, or even "programming" at all is a topic of scholarly debate [26]. For me, however, the situation is simple: AnsProlog is a programming language (at least at the same level of fuzziness as Prolog); ASP is a paradigm; and, yes, it is "programming" (a craft practice for making machines do stuff). In my estimation, disagreement on any of these points only functions to preserve the status quo where ASP is a relatively obscure academic curiosity, forever in the shadow of crufty old Prolog.

rule. Although the grounding phase of modern ASP systems is Turing-complete (a relatively recent change [76]), the grounding phase of ASP is similarly insensitive to definition ordering.

The key strategy of ASP is to target problems in a specific band of computational complexity, namely *NP-hard combinatorial search and optimization problems*, and make this family of programs easy to specify (at the cost of Turing-completeness). This focus, applied both to performance and usability concerns, on difficult search and optimization problems not only makes ASP an attractive choice for developing design automation tools (as the rest of this dissertation argues), but also for providing a simpler alternative and high-performance foundation for next-generation ILP systems [43].

6.2 Standard Problems

Combinatorial search and optimization problems arise frequently in the creation of design assistance tools and play-time design automation systems. However, these problems are always tied to specific design issues in specific games. Thus, it is not immediately obvious what, of the large body of past work in combinatorial search and optimization (which extends far beyond logic programming), is relevant to learn from or reuse.

Solving a domain specific problem (without inventing an entirely new algorithm) usually involves identifying which standard problem it most closely resembles. Thus, knowing a little about the variety of problems for which we already have high-performance solutions can go a long way towards accelerating development of solutions for the problems that arise in the mechanization of exploratory game design.

I provide a non-exhaustive sampling of some common problems. Scheduling involves the assignment of tasks (with durations) to time slots and resources so that the overall cost or delay of a task set is minimized and that no resources are overused. Planning involves the invention of a sequence of actions whose effect on a system is to satisfy some goal. Theorem-proving seeks a sequence of applications of inference rules that form a proof of a goal from a set of premises. Constraint satisfaction involves selecting an assignment of values to variables so that all or as many as possible of a set of constraint expressions is satisfied. Integer programming is a variation on linear programming in the case of integer variables. Finally, Boolean satisfiability checking asks, given a set of formulae over Boolean variables representing rules and propositions, whether there is an assignment of truth values to propositional variables that satisfies every formula.

I hesitate to list more common problems, as there is little diversity

of interest in this space of problems. One of the foundational results of theoretical computer science is that most variations of the above problems are mutually reducible [104]—an algorithm for one can solve all the rest at the cost of an encoding that may (only polynomially) inflate the size of the problem. This phenomenon is not just a theoretical curiosity; it is the mechanism by which pragmatic designer-programmers can gain access to search and optimization power tools without committing to a career in algorithm design.

The *search* variants of these problems (the ones that answer whether there is a solution and, if so, what is one such solution) generally have a complexity in NP-hard (the class that includes SAT) while the *optimization* variants (that seek the smallest, biggest, or otherwise best satisfying solution) have a complexity in FNP-hard (the class that includes MAX-SAT). Many algorithms exist to solve the different problems because algorithms designed for particular problems can take advantage of knowledge of the common structures that commonly arise in that problem. Thus, while one system, such as SATplan [106], might solve planning problems by translating them into SAT problems, a faster system, such as FF [91], might use an algorithm specifically designed to solve planning problems.

The different problems above involve overlapping features such as propositions, fluents, assignments, constraints and goals. Reducing a domain specific problem to one of the standard problems requires expressing the concerns of the domain with the structural features supported by that problem. Many standard problems involve propositions: logical statements that may or may not be true. Via a grounding process (also called propositionalization), first-order predicates over the logical state of a domain scenario can be readily shoehorned into a propositional framework. Fluents are propositions whose value varies in time. While planning problems natively involve fluents, their presence can be simulated in non-temporal problem formulations by reifying their value over time as a predicate that takes a time-point parameter. Assignments are mappings between domains and ranges, and they are the native structures of constraint satisfaction problems. If a domain involves assignment of a variable or value type unsupported by a particular problem, a auxiliary predicate (taking a variable-name parameter and a value parameter and stating that the variable is assigned that value) can be introduced to model the assignment. Different problems involve different kinds of constraints (relative ordering of tasks in a schedule, co-occurrence of actions in a plan, forbidding or requiring the truth of certain propositions in satisfiability checking, etc.), so picking the problem which most naturally allows the constraints of a domain to be expressed is important for saving human modeling effort (potentially more important than the run-time performance gained by making a speed-focused choice). Finally,

goals are target formulae to prove, explain, or answer (as a query). Goals and constraints are largely interchangeable: one can constrain that a goal is achieved or set as a goal that a set of constraints be satisfied.

None of the standard problems will capture exactly the concepts of interest in particular game design automation problems. However, the degree to which game-specific problems involve standard features like fluents (perhaps the position of the player character over time) and constraints (that there exists a solution to a generated puzzle), the available software tools for standard problems can be recycled to save the effort of developing and maintaining entirely new systems.

6.3 Standard Solutions

Without going into the details of any one planning or constraint-solving algorithm, there are a number of features shared by many advanced algorithms used in combinatorial search and optimization. Without reusable infrastructure, play-time design automation projects (for richly constrained problems) would either be encumbered by the need to understand and apply these features or suffer the run-time performance penalty of not applying these highly beneficial techniques that are known for their wide applicability in the combinatorial search community. Even though knowledge of the following features is not required to build design automation systems with the techniques of Chapter 10, it is instructive to know how much complexity is hidden under the hood of modern combinatorial search infrastructure.

6.3.1 Backtracking and Completeness

The template used for many combinatorial search algorithms is the following: make a choice (e.g. assign a variable, select an action, mark a proposition as true or false, etc.), simplify the resulting problem, and then invoke the same algorithm on the simplified problem. If the recursive call fails, make a different choice. If there are no more choices to try, then backtrack on the most recent choice (causing choices at a higher level to be reconsidered).

This search fundamentally works variable-by-variable, and, upon backtracking, entire regions of the search space of potential solutions are discarded as a whole. This is distinct from a generate-and-test setup where candidate solutions are completely assembled (perhaps making an assignment for every variable) before the constraints or goal conditions of the problem are checked. Even though not every combinatorial search algorithm uses backtracking, backtracking is an important reference strategy

to contrast with the more common generate-and-test and contingent[4] generation methods in PCG.

An important property of many backtracking algorithms is completeness—no feasible solutions are overlooked, and thus, they will eventually enumerate all solutions if allowed. Backtracking and completeness are independent concepts: not all backtracking algorithms are complete (e.g. depth-first search on a graph with cycles will diverge, missing potential solutions), and many complete search algorithms do not backtrack (e.g. breadth-first search). Completeness is an important property for the artifacts-as-communication strategy. It is informative when an algorithm can report, with certainty, that no artifacts exist with the requested properties (e.g. to ensure that a puzzle has no shortcut solutions).

6.3.2 Heuristics

Instead of always making the first available choice in a search problem, most high-performance algorithms will ask a heuristic function to estimate the value of making a particular choice based on the structure of the remaining problem (e.g. the number of constraints that involve the variable to be considered next). Heuristics attempt to prioritize search along promising directions. *Admissible* heuristics will change the order in which choice will be explored, but they will not prune away any options that would eventually need to be explored.

Different heuristics can cause a fixed algorithm to perform dramatically better or worse (in terms of running time) depending on which structures are present in the problem instance to which they are applied. Many heuristics encode algorithm-specific search advice, and provide no potential design insight (aside from the indirect impact of faster prototype iteration cycles). Thus, I claim design automation tools should shield designers from the need to develop, encode, or choose between heuristics while exploring new design territory. Although it is often tempting to encode problem-specific search advice as heuristics (locking the search algorithm into a particular domain), tools should provide hooks that allow a designer to specify this advice with minimal coupling to the underlying search algorithm.

[4] In contingent generation systems, the artifacts emerging from the generation subsystem depend on the set of artifacts that have previously passed through the testing subsystem (e.g. how mutation and crossover in genetic algorithms operates on the population that has passed a selection process based on fitness scores). Backtracking search, despite making choices that strongly depend on the outcome of previous choices, does not fit into this category because it builds solutions piece-by-piece and does not employ any subsystems that directly propose complete artifacts (i.e. they have no clear "generator" to distinguish from a "tester" even without regard for the level of contingency).

6.3.3 Consistency and Propagation

Instead of directly iterating through all potential assignments for a variable (in the heuristic-advised order), many algorithms also explicitly track the set of values each variable may take given the choices made so far. When one variable is assigned, it often results in a reduction of the set of consistent values other variables may take. Thus, propagating constraints amongst variables can significantly reduce the search space, and such reduction is potentially available each time a choice is made by the search algorithm. Constraint propagation effectively provides an upper bound on the space of feasible solutions to a problem. As a result, it can provide early feedback that allows a search algorithm to abandon large branches of a search space.

Constraint propagation (also called local consistency) can report that a problem is infeasible (when a set of locally inconsistent constraints is detected or a variable can be shown to have an empty domain), but it cannot generally prove that a problem is *feasible*. There are some problems for which there are no locally inconsistent constraints, yet the there is still no satisfying assignment of the variables to be found.[5] As a result, constraint propagation is an *incomplete* decision procedure for constraint problems, but it is immensely useful nonetheless.

Constraint propagation can be interleaved with human gameplay to create new game mechanics. After each move in a puzzle game, the space of consistent remaining moves can be updated. This space could be visualized for the player as an optional hint, or it could be used as the basis for a trigger that notices when the player wanders into infeasible territory (where they can only make progress by undoing past moves). Because constraint propagation is weaker than complete search, it often yields answers quickly, making this technique potentially suitable for online, reactive analysis of live gameplay.

6.3.4 Constraint Learning

Another critical feature of many modern search algorithms that aims to allow early-escape from infeasible territory is constraint learning. When the search algorithm encounters a conflict (dead end), instead of simply backtracking, it analyzes the conflict in an attempt to extract the simplest

[5]Consider this set of constraints on integers $a, b, c \in \mathbb{Z}$: $a < b$, $b < c$, and $c < a$ (a rock–paper–scissors setup). The problem is arc-consistent (i.e. considering each pair of variables in isolation, for every value one variable may take, there are values in the other variable's domain for which the constraint may be satisfied), yet the problem is infeasible. This trivial problem is not path-consistent (a path-consistency checker, considering triples of variables, could prove it infeasible), but there are counterexamples to the completeness of path-consistency algorithms as well.

underlying cause. When only a small subset of the previous choices are responsible for violating one of the problem's constraints, those choices can be transformed into a new constraint that further prunes the search space the algorithm must consider. Because this constraint is implied by previously known constraints, it will only cause pruning of choices that would have eventually failed anyway; constraint learning does not change the set of solutions that the algorithm will eventually output, but it does change the speed at which they are uncovered.

This learning-during-search process is an excellent formalized metaphor for the learning in the inner loop of double-loop learning. The learner gains knowledge of new, previously unexpressed constraints that change how future search will proceed, but at no time is the overall problem refined in a way that changes what counts as a solution.

6.3.5 Randomness and Restarts

Exacerbated by the inclusion of many of the above advanced techniques for speeding up search, a common problem with many complete combinatorial search algorithms is extremely unpredictable running times. Depending on the cleverness of heuristics, constraint propagation, and the conflict analysis underlying constraint learning, some problem instances may be completely solved after a few choices while slight variations on the problem instance will trick the algorithm into plodding through nearly all possible combinations of choices (a space that is exponential in the number of assignable variables in the problem).

The heavy-tailed cost distributions associated with deterministic algorithms contribute to their seeming brittle and unreliable (despite theoretical guarantees of termination). The strategy of introducing randomness and restarts allows an algorithm to trade small increases in best-case running time for massive decreases in the mean and variance of running times [85].

The bulk of wasteful search (that explores branches of the search space containing no solutions) is the result of particularly unfortunate assignments being selected very early in the search process. To avoid spending time in this unpromising territory, many modern algorithms include logic that will restart the search (either from the root or from local landmarks) after a certain number of conflicts (called the cutoff value) are encountered. However, in a deterministic algorithm without constraint learning or other side effects of exploration, simply restarting the search would be pointless, as the algorithm would proceed to get bogged down in exactly the same area. Accordingly, the general trick is to encourage restarts to explore alternative choices. Often, an algorithm implements this by occasionally (with some probability) ignoring the advice of the heuristic and

making a random choice instead. This probability parameter trades between trust in the domain-independent (but structure-aware) heuristic as an indicator of where to explore next and trust in the prevalence of conflicts as a domain-dependent (but structure-agnostic) signal of solution likelihood.

If implemented correctly, the introduction of restarts does not affect the completeness property of an algorithm. The simplest way of retaining completeness after adding restarts is to automatically expand the cutoff value after each restart (usually by a constant multiplicative factor, leading to a geometric progression of cutoff values) so that eventually the entire space of the finite-by-definition problem can be exhaustively searched without restarting. More advanced and asymptotically optimal restart policies (including those that continuously update beliefs about the running time of the ongoing search process [105]) are also possible.

In Chapter 15, I describe how a geometric restart policy was backported into the implementations of previous game content generators (originally based on depth-first search), yielding solutions to previously intractable problems and a mean and variance of running times small enough to be feasibly measured for comparison to other methods. This change, applying a small piece of advanced search algorithm design wisdom, required alteration of a problem-specific algorithm. In the ASP-based content generators that I created to replace the original generators, a wide array of choices for different search heuristics, constraint learning policies, and restart policies could be explored by simply passing different configuration arguments to the domain-independent answer set solver, leaving the domain-specific problem definition untouched.

6.4 General Game Playing

Historically, the deepest point of contact between symbolic AI and games has been the development of programs that play strategic games using AI techniques. Emerging from early explorations with AI Chess playing, general game playing (GGP) is the symbolic AI problem of playing arbitrary games competitively. Instead of applying human intelligence to the development of new algorithms for each new game to be played, GGP agents must be designed to take the rules of an unfamiliar game as input. Thus, these systems demonstrate that strategic play for arbitrary games is readily mechanizable. The link between games as artifacts and the spaces of play they afford (at least in mathematical terms) has a broad and formal basis in symbolic AI.

6.4.1 History

In 1958, Newell, Shaw and Simon [157] surveyed the very early stages of computer game playing. Although early work focused specifically on Chess as an application area where human players provided an appropriate reference opponent, the foundational principles were always formulated with a broad space of board games in mind. From John von Neumann's 1928 introduction of the minimax principle as a general theory of board games, through Claude Shannon's 1949 instantiation of minimax for reasoning in Chess as a plan for how an algorithm could one day be designed, to Alan Turing's 1950 hand-simulation of a Chess-playing algorithm that did not need to evaluate all possible continuations of a board state, these early efforts were intended to demonstrate that game playing (an activity involving intelligence and even creativity) could be meaningfully captured in symbolic terms.

Later, with the development of alpha-beta pruning, quiescence search, transposition tables, endgame tablebases, and other advanced and often Chess-specific software and hardware design techniques, machines were eventually made that could occasionally beat the best human game players (with the 1996 and 1997 games between IBM's *Deep Blue* machine and reigning world champion Gary Kasparov being a landmark). Human players in the game of Go, however, still regularly outsmart the best of Chess-inspired Go-playing machinery. As a result, much game playing research has shifted focus from Chess to Go, yielding the development of new methods involving Monte-Carlo tree search that trade worst-case guarantees for improvements in average-case playing performance [213].

Just as Go-playing programs are informed by the best of the techniques derived from Chess, GGP agents (such as FluxPlayer [180] and Centurio [146]) recycle and generalize results from Go-focused developments. Following the general trend in development, it seems that the tricks developed for particular games eventually find their way into general-purpose tools. As a result, work that simply intends to use GGP technology, perhaps in the service of design assistance tools of play-time design automation, need not be distracted by tracking the latest developments in algorithm design. The space of games for which, given just a symbolic description, standard tools can quickly calculate mathematically perfect (or at least acceptably competitive) play will only increase over time.

6.4.2 Rule Representations

Describing the rules of arbitrary games to a black-box GGP agent is a technical challenge in itself. Different game playing programs accept different rule representation languages with different levels of expressivity.

The more a game representation language is restricted, the more assumptions the designer of a GGP agent can make about the space of play (yielding improvements for run-time and strategic performance). At the same time, loosening restrictions on the representation language allows for one GGP system to be applied to a much wider space of games, making reusable infrastructure available for design automation projects that cannot afford the development of original, game-specific strategic play systems.

An example that starts from the heavily restricted end of the spectrum is the Zillions of Games (ZoG) commercial GGP system.[6] The system ships with encodings of several hundred familiar competitive multiplayer games such as Chess, Go, and Tic-tac-toe as well as single player games and puzzles like the 15-Puzzle and Towers of Hanoi. ZRF, the language capturing the rules of these games, includes constructs for describing the order of turns, properties and descriptions of game pieces, the initial board configuration, and winning conditions as geometric relations over board state. The language is designed for human authoring, and it includes the ability to reference art assets (such as images to represent pieces) that will be used when playing games against human opponents and to include natural language notes (giving an overview of the game's common strategies, for example). Because all games in ZRF necessarily involve cyclic turn-taking over games with completely observable state (i.e. they are perfect information games), a Chess-inspired (depth-limited alpha-beta minimax with other common enhancements) algorithm is able to function as a opponent for any game expressible in ZRF.

Whereas ZoG is a consumer product—an opponent-in-a-box for a variety of familiar games that human players already have traditions of playing—the AI research community explores the other end of the spectrum: they intend to produce competent agents for any game, regardless of how much attention human players have given that game or games similar to it in the past. The goal is to advance the state of the art in strategic search, rather than to create opponents for human players.

The common representation used in machine-vs.-machine GGP competitions is GDL (the Game Description Language) [82]. GDL makes no Chess-related assumptions—it does not presume the existence of a game board, pieces, capture rules, or strict alternation between player moves. In GDL, games are specified by abstract logical rules (akin to Prolog rules) for determining which statements are true initially, what actions are allowed by each player in a particular game state, and how to compute the properties of the next game state after player actions are selected [130]. GDL expressions can be used to describe board games with turn-taking

[6] `http://www.zillions-of-games.com/`

by modeling the board and the state of whose turn it is with logical predicates, but it is also possible to model other strategic encounters, such as card games, that are not defined in terms of the movement of pieces on a board. The latest variants of GDL can express games with simultaneous actions, imperfect information, and spontaneous probabilistic events [215].

Programs that play GDL games are, in effect, logic programming systems with hard-coded queries (e.g. which legal action has the highest estimated value according to this estimation scheme?). GDL constructs are not so much games as they are state–transition system descriptions: the rules are a compressed representation of the space of legal board states and the move-labeled transitions between them. In terms of Nelson's factoring of the concerns of game design, GDL speaks only to abstract mechanics. General game playing is highly relevant to game design automation; however, it covers just one facet of design interest. In Chapter 12, I describe my machine playtesting tool, LUDOCORE, a kind of GGP system for single-player games specified in a language with the broad expressivity of formal logic similar to GDL and an intent to be authored by human game designers similar to ZRF. Later, in Chapter 13, I describe BIPED, a system that allows human play of LUDOCORE games with a reactive graphical interface that parallels ZoG's player interface (something that does not exist for the GDL ecosystem).

6.4.3 General Game Playing vs. Action Planning

GGP and classical action planning both involve selecting actions that bring about some goal condition in an environment governed by logical rules. However, the problems are distinct, with GGP being significantly more difficult than action planning. In most variations on action planning problems, performing a particular action in a particular state results in a unique and easily derivable successor state. As a result, classical planning algorithms need only decide what to do in the states of the world that they intentionally bring about. On the other hand, in multi-player games or games that include an unpredictable natural environment, input from the other players or nature is needed to deduce which state arises next. Therefore, in game playing (or non-deterministic planning generally), an algorithm must consider what to do in any state that its potentially adversarial environment can force it to enter.

This relationship is the same as the one between the complexity classes NP and AP that describe the problems that can be solved in Polynomial time by Nondeterministic Turing machines and Alternating Turing machines respectively. Intuitively, problems in NP can require guessing a single solution (e.g. a sequence of actions) that can be quickly verified to

achieve some goal. Problems in AP (equivalent to PSPACE) can require guessing a general solution policy (e.g. an action to perform in response to any opponent action) such that a key property of the solution policy (perhaps that a win for the first player is forced) is verifiable. For a game with n types of moves played for a maximum k steps (a constant), this solution policy could take $O(n^k)$ space to store in an directly verifiable form. This is why AP is equivalent to PSPACE and why perfect play for generalized versions of many games (often played on n-by-n boards) is PSPACE-complete. For generalized games without a constant depth bound (e.g. n-by-n Chess [71] or Go [172]), perfect play is EXPTIME-complete (equivalent to APSPACE).

Unlikely as it sounds, this is *good news* for the rapid development of game design automation tools. The problem of simulated play for all known games breaks down into three major categories, for each of which there is a concerted and ongoing effort to develop domain-independent and high-performance combinatorial search and optimization infrastructure: NP (for deterministic, single-player games), AP/PSPACE (for non-determinstic or adversarial games with constant depth), and EXPTIME/APSPACE (for adversarial games with non-constant depth). For problems in NP, SAT-like systems (including most answer set solvers) are a natural choice for rapidly developing solutions. For problems in AP, QBF (for unrestricted quantified Boolean formulae, also called QSAT) solvers such as Quaffle [232] or sKizzo [8] play the same role that a SAT solver like MiniSAT [206] plays for problems in NP. Interestingly, advances in QBF solving have determined that treating problems in AP not as quantified formula to be satisfied but as two-player *games* to be won by a specific player yields dramatically more concise encodings and better solver performance [2]. For games reaching all the way to EXPTIME, there are no general-purpose SAT-like tools. However, the very same game playing systems that enter GGP competitions can be reused to gain approximate solutions without inventing any new algorithms.

The obsession with perfect strategic play in these formulations of the game playing problem may seem misguided in light of a human game designer's interest in producing games for play by human players (who lack the mental resources to realize perfect play for most games). However, when a designer is allowed to include extra restrictions (in the form of additional rules) on the strategies chosen by simulated players, systems that mechanize perfect play become immediately useful for mechanizing playtesting against hypothetical players with any biases and handicaps the designer cares to encode. Imagining that a certain kind of player will always or never take advantage of particular actions when the opportunity arises, solvers for perfect play directly permit asking "what is the best/worst thing that could happen to a player who plays according to

this restriction?" This is exactly the idea of the *restricted play* framework proposed by Jaffe et al. in the context of building a game balancing tool [98].

Assuming that fundamental improvements in algorithms for NP, PSPACE, and EXPTIME problems will come from experts who are dedicated to solving these problems in the abstract, formalized game playing, in all of its variants, is as *solved* as it is ever going to be. Developing game design automation systems via reductions to standard problems for which domain-independent solvers exist is not only a way to avoid project-specific search algorithm design, it is a way to future-proof[7] the design and implementation of these systems.

6.5 Super-symbolic AI: The Knowledge Level

Almost all of the key results in artificial intelligence are tied to specific symbolic systems (such as formal logic and frame systems) or specific sub-symbolic architectures (such as evolutionary algorithms and artificial neural networks). Nearly all of the broad goals of AI engage with human intelligence in some way (either as an object of study or a reference point for comparison); however, the human mind is neither particularly fluent with formal logic nor is the human brain made from neurons that bear any deep resemblance to those artificial neurons historically of interest in AI. To speak about the intelligence of humans and machines in unbiased terms, a radical alternative mode of discussion is needed that does not presume or prefer particular details at the symbolic and sub-symbolic levels.

In the very first presidential address to the then American Association for Artificial Intelligence (AAAI), Allen Newell set out to define a level of description and discussion for intelligent systems that was independent of their symbolic and sub-symbolic architectures [158]. Two key concepts are co-defined at Newell's *knowledge level*: *knowledge* and *rationality*. At this level of description, an intelligent *agent* (human, machine or otherwise) is something with a physical body that is capable of *actions*, a body of *knowledge* to which more knowledge can be added, and a set of *goals*. These actions, bodies of knowledge, and goals need not be found in any

[7] Occasionally new facets of combinatorial search algorithms are invented (such as constraint learning or random restarts) that have sweeping effects on the design of algorithms for all problems within a complexity class. In these cases, even if one is an expert in, say, action planning, one may be blindsided by dramatic improvements uncovered by SAT experts. Not committing to algorithmic details in the design of search-intensive applications provides insurance against these unexpected exponential and super-exponential speedups at the cost of a usually polynomial-time encoding process.

particular data structures or organs of the agent—they are details we ascribe to the agent through observation of its overall function in an environment on the basis of our own knowledge (i.e. it hinges on adopting an *intentional stance* [55]).

Defining a "principle of rationality," Newell writes, "If an agent has knowledge that one of its actions will lead to one of its goals, then it will select that action." Agents will only take actions that are physically possible and have been selected, but further (optional) extended principles of rationality may be needed to narrow down the set of selected actions to just one if such predictive power is desired. "Knowledge," Newell writes, is "whatever can be ascribed to an agent so that its behavior can be computed according to the principle of rationality." The co-definition of knowledge and rationality is not accidental—it is a precise definition of a workable description of an agent at the knowledge level. That is, if an agent behaves rationally according to the actions, goals, and knowledge you attribute to it, you have successfully described it at the knowledge level. The knowledge level cannot, however, tell you whether an agent is objectively rational or not. Further, the sense of rationality defined at the knowledge level is largely independent of utility-maximizing economic rationality (although this is an example of what form an extended principle of rationality might take).

Given a knowledge level description of an intelligent agent along with a known symbol level description, we can say that the knowledge level *rationalizes* (or puts knowledge-informed *intent* behind) an agent's behavior while the symbol level *mechanizes* (or puts *subpersonal* mechanisms behind) that behavior. The primary function of knowledge level modeling is to reflect on the apparent function (i.e. knowledge + goals + actions) of intelligent systems with the aim of producing new systems that rationally pursue a similar function more effectively. As a whole, this dissertation follows a program very much inspired by knowledge level thinking. It proposes *rationale* for the behavior of exploratory game designers (suggesting they are after design knowledge and experience that will expand their ability to confidently design new games) at the same time that it proposes new *mechanisms* of game design (offering systems based on symbolic logic which show how key parts of exploratory game design can be carried out without the magic of personhood or the requirement of recognizable intelligence or creativity). Chapter 16 offers a broadly-informed knowledge-level description of game design that, in light of the formal systems developed in the chapters leading up to it, makes bold new predictions about the nature of human creativity in design processes. It culminates in a new, computationally realizable model of creativity defined expressly at the knowledge level: *rational curiosity*.

Chapter 7

Synthesis of Goals

In this brief interlude, I synthesize the goals of my dissertation as concerns at the intersection of the fields in the previous chapters and link them to the five cross-cutting strategies mentioned in the introduction. Figure 7.1 provides a high-level visual overview of the major interdisciplinary links and situates my goals against this interdisciplinary background.

Between game design, design studies, computational creativity, and symbolic AI, each is an application, an instance, a motivator, or a new perspective for the others. While it is already uncommon for any pair of these disciplines to come into contact as deeply as they do here, my work hinges on bringing all four into simultaneous coherence. In the chapters to come, I will freely mix the vocabularies of each of these in describing a new technical process for game design, the declarative modeling of design spaces, and an array of systems exemplifying this process.

The goal of expanding human creative reach in game design is specifically tied to the concerns of games (vs. general design). To achieve this goal, I use symbolic AI to give designers new modes of prototyping and playtesting as well as offer acceleration for game content design and evaluation tasks. Directing these technical interventions where they are appropriate requires a description of game design in terms of (computational) creativity, particularly one that highlights bottlenecks in the creative process that can be relieved. The strategies of exploiting a machine's ability to automate logical reasoning and the strategy of developing computational caricatures are most effective towards this goal, where the way in which machines *can* and *should* assist game designers remains wide open for exploration.

The goal of supporting deep, play-time design automation is not one that requires creativity, but it does require the automation of key design thinking processes. In the same way that symbolic AI can be used dur-

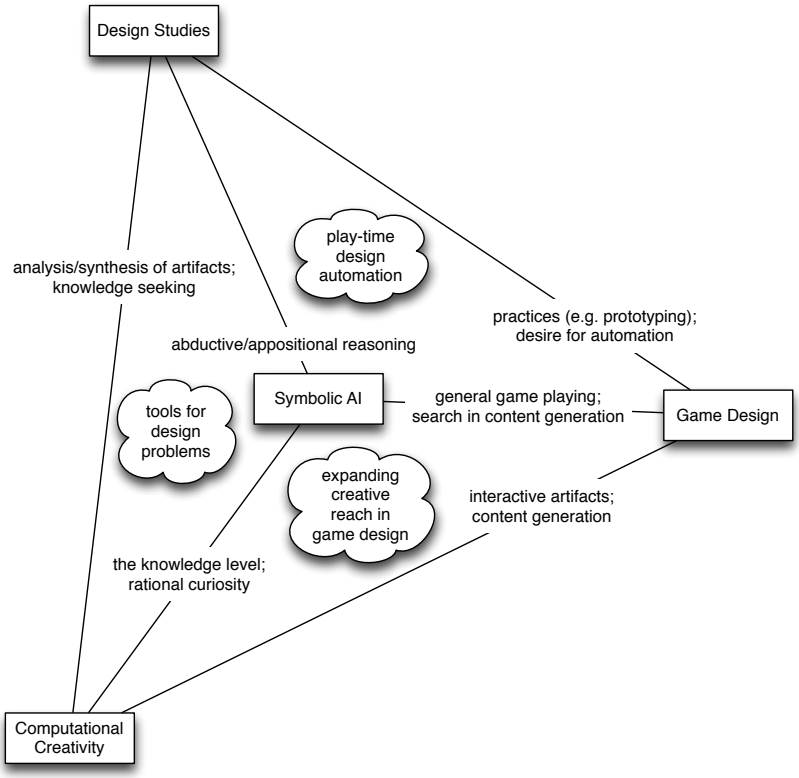

Figure 7.1: Non-exhaustive visual survey of the key interactions between each of the disciplines that this dissertation engages. The three goals of the dissertation are synthesized as concerns at the intersection of triads of these disciplines, with each making heavy contact with symbolic AI.

ing the design process to support the designer, it can be used in play to support new and particularly player-responsive game mechanics and play experiences. This goal is best served by the strategies of focusing on design spaces over individual artifacts and targeting procedurally literate designer-programmers who are capable of authoring these spaces for future games.

Finally, the goal of demonstrating tools that respect design problems is a natural outgrowth of the perspective adopted in this dissertation when the specific concerns of game design are factored out. The inherent properties of design problems as identified in design studies and the demands of creativity placed by my own model of creativity and that of others suggests the need for a new generation of software tools that specifically targets the outer loop of double-loop learning: applying the available infrastructure of symbolic AI to help designers in problem formulation and transformation. The artifacts-as-communication strategy is most effective towards this goal, where, by speaking in detailed artifacts, the bandwidth for situational backtalk in exploratory design is greatly enhanced over specification-oriented feedback mechanisms common in design verification (e.g. in the formal methods of software engineering).

Chapter 8

Mechanizing Appositional Reasoning

In the field of design studies, the term "appositional reasoning" denotes a mode of cognition that is also labelled "abductive reasoning." However, instead of seeking an *explanation* that is consistent with situational observations (the logician's sense of abductive reasoning, readily mechanized with symbolic AI tools), a designer's appositional reasoning seeks configurations of artifacts that are *appropriate* given the designer's best understanding of an ill-defined design scenario. In this chapter, I map the vocabulary of human design practices into the domain of AI knowledge representation, explicating notions such as design space, artifact, form and function. This mapping suggests new requirements on knowledge representation for design spaces based on a *non-explanatory* reading of abduction. In light of these newly discovered requirements, I motivate the translation of the concerns of appositional reasoning into answer set programming (ASP), a programming paradigm supporting abductive logic, and articulate concrete demands for a future generation of symbolic AI tools that would better enable the mechanization of appositional reasoning.

8.1 Introduction

Design automation accelerates human productivity and creativity, but it often proceeds in narrow, domain-specific directions that neither draw from nor intentionally contribute to our general understanding of human design thinking. Work in design studies has explicated ways of thinking that are unique to design and not subsumed by either the sciences or

the humanities [47, preface]. Developing a computational account of the key ideas in design thinking will unlock new opportunities for design automation based on domain-agnostic AI techniques and shed light on the idiomatic use of existing automation techniques.

Recall (from Chapter 4) veteran design researcher Nigel Cross' identification of design ability as "resolving ill-defined problems, adopting solution-focused cognitive strategies, employing abductive or appositional thinking, and using non-verbal modelling media" [47, chap. 1]. Though each of these terms refers to deep and complex topics in design studies, the mention of "abductive" thinking should offer some familiarity to the AI researcher, but "appositional" less so. Abduction has been called the "logic of design" [134], in a formal motivation for the leap from required functions to the potential form of a designed artifact. Why abduction (the unsound reasoning from observed effects to explanatory causes which underlies diagnosis) has anything to do with design, however, is better explained by examining the alternate terminology, that of appositional reasoning.

In the rest of this chapter, I unpack the concept of appositional reasoning with the aim of uncovering the requirements it places on the knowledge representation for systems that would automate this mode of cognition. My central strategy is to detach design from the explanation-focused intent of logical abduction and to refocus it on the synthesis of *appropriate* design solutions, the core concept of appositional reasoning (as defined in the next section). This move reveals new uses for AI-supported abductive logic programming in creating systems that automatically generate artifacts along with demonstrations of their appropriateness. By mapping the idea of a design space (from which artifacts are created through appositional reasoning) onto the knowledge representation affordances of answer set programming (ASP), I can realize design automation systems for a variety of domains while maintaining a strong connection with the best understandings of human design practices.

8.2 Appositional and Abductive Reasoning

Appositional reasoning, though variously defined [47, chap. 2], is intended to refer to *the synthesis of artifact designs that are apt, appropriate, or fitting to a design scenario.*[1] Well-designed artifacts (whether they be building plans, circuit layouts, or mechanical parts) are intricately mated to their outer environment and intended use (in that their inner envi-

[1]Keep in mind that there is more to design than just synthesizing artifacts. To pick just one other element from Cross' list, *resolving ill-defined problems* is another essential process in design.

ronment conforms to expected styles, resource constraints, or durability concerns).

However critical the fit between artifact and scenario, the definition of the conditions of this fit (perhaps a "fitness function" or "complete specification") is rarely available for realistic design problems. Recall the endemic "ill-definedness" [216] that is incrementally overcome by iteratively proposing artifacts that are fit to the designer's best understanding of the design problem so far. Given a concrete candidate solution, it is possible to ask specific questions about what a design scenario requires even when stakeholders are not able to articulate their needs in general terms. Throughout this process, the requirements of the design scenario (the details defining the design problem) are themselves subject to the same iterative refinement that is applied to candidate solutions.

The strategy of tackling ill-definedness by leading with candidate solutions (one of the designer's key abilities as Cross noted above) further emphasizes a designer's need to be able to reason from the current hypothesized understanding of the design scenario to the design of a fitting artifact—to apply appositional reasoning. Appositional reasoning is required not just to produce a final artifact for delivery, but also to even learn what the problem is!

To a first approximation, "abductive reasoning" and "appositional reasoning" are used interchangeably in the literature of design. Interpreting this practice with an AI reading of abductive reasoning (detailed shortly) would suggest that the heart of appositional reasoning is to be found in the process of forming explanations. This is inconsistent, however, with the accepted understanding of human design processes. Abductive reasoning, originally due to Peirce [164], has become associated with "inference to the best explanation." The following scenario captures this mode of inference [100]:

> D is a collection of data (facts, observations, givens),
> H explains D (would, if true explain D),
> No other hypothesis explains D as well as H does.
> Therefore, H is probably correct.

In design, however, the notion of observations finds no direct referent, and there is no universal equivalent of Occam's razor (often used in judging the quality of explanation) that generalizes across styles and genres of design.

Echoing Roozenburg [173], I associate the primary pattern of reasoning in design with a *non-explanatory* sense of abduction, one that he calls "innovative abduction." Meanwhile, Magnani's "manipulative abduction" [133], a mode of scientific reasoning that foregrounds construc-

tion of artifacts in place of explanations, resembles the solution-focusing strategy familiar from design thinking.

To forge a link between the appositional reasoning of design and the well understood abductive reasoning supported by domain-agnostic AI tools, I need to further unpack the idea of appositional reasoning by examining the construct to which this reasoning is applied.

8.3 Design Spaces

In the context of this dissertation, I define a *design space* to be the set of artifacts (or candidate solutions) that would be appropriate for a design scenario, given a designer's experience thus far. To conjure a fitting artifact, a designer need only sample any element from this set. When the set is defined extensionally (perhaps as a small collection of pre-approved artifacts), this task is trivial. In the more interesting case, when the set is defined intensionally[2] (through a definition of the necessary and sufficient conditions for appropriateness), the task of producing an artifact becomes intimately tied to the nature of the design space's formal representation.

The conditions for inclusion in a design space are exactly the conditions of appropriateness. Unpacking appropriateness, we meet the concepts of form and function. *Form* is comprised of all of the details necessary to realize an artifact (its shape, size, material, internal composition of subsystems, etc.). Meanwhile, *function* is the capacity of the artifact to be used in a larger environment (including interaction with external agents). This vocabulary could be expanded, as in Gero's seminal schema for design prototypes [83]; however, this definition of form and function are sufficient to articulate a general computational account of appositional reasoning.

Though design is often idealized as the derivation of form from function, realistically, the relationship between form and function and between the two of these and appropriateness is more complex. A designer's experience or access to external resources can place strong constraints on the appropriate form of an artifact, regardless of its intended function. Likewise, a designer's intent to embed a signature style or to emulate the style of another designer is an additional source of constraints on form. Meanwhile, the sources of constraints on appropriate function are widely varied as well, from simply pleasing a client to supporting an intended use (such as dispensing cash) even in the face of an adversarial user or en-

[2] "Intentional" means to do something on purpose, with intention. "Intensional" precisely means being defined by necessary and sufficient conditions—it's a math thing. The definition $\mathbf{X} = \{1, 2, 3\}$ is an extensional definition where $\mathbf{X} = \{1 \leq i \leq 3 | i \in \mathbb{N}\}$ is an intensional definition.

vironment. Requirements placed on an artifact's function may translate not just to constraints on form, but also to constraints on the environments to which the artifact may be deployed (e.g. re-scoping the target audience).

Sometimes what makes an artifact appropriate is precisely its dysfunction, or, more precisely, its ability to disrupt the function of another artifact. When trying to probe the robustness of an existing system, designers may seek out artifacts that, in interaction with that system, cause a failure or other undesirable emergence—playing the devil's architect. For example, when designing a system of tax laws, appositional reasoning is at play in inventing plausible business plans that could exploit loopholes in the policy. The function that these likely undesirable artifacts serve is to argue against the working design of a larger system. The existence of this type of dysfunctional artifact, even if only conceptual, is an important form of situational backtalk that drives larger processes in design thinking.

Appositional reasoning, more deeply, is the construction of artifacts which come from a designer's working design space. The design space is a construct defined by the interleaving concerns of form and function, whose constraints are driven by both the design scenario (external constraints articulated and interpreted thus far) and the designer's own intent and curiosity.

8.4 Requirements for Knowledge Representations

At this point, it is clear that a computational account of appositional reasoning hinges on the ability to formally represent design spaces: collections of evolving constraints on appropriateness which are to be authored in part by external stakeholders and in part by computational systems themselves. In terms of AI knowledge representation (KR), we need a substrate that is capable of expressing constraints over a space of static artifacts (sufficient for judging appropriateness of form). Further, we require support for expressing constraints on the dynamic interaction between an artifact and the relevant facets of its environment (sufficient to judge appropriateness of function). Finally, we clearly need to tie this representation to efficient tools for sampling artifacts from design spaces. I offer the following requirements on a KR scheme for design spaces that satisfyingly supports appositional reasoning:

- **Constructivity**
  - Expresses complete configuration of artifacts

  - Supports automatic artifact generation

- **Constraint**
  - Expresses form constraints (both local and global)
  - Expresses function constraints (existential and universal)

- **Evolution**
  - Supports iterative development (elaboration tolerance)
  - Supports automatic learning and/or repair of constraints

- **Analysis**
  - Supports justification of appropriateness
  - Supports diagnosis of externally provided artifacts

8.4.1 Constructivity

The requirements of constructivity capture my intent that the represented design spaces are not merely descriptive or analytic; these design spaces should directly afford automated synthesis of usable artifacts. I say the configuration of an artifact is complete if that configuration includes enough detail to realize the artifact without further design thinking. The configuration may be, for example, simply the input into a lower-level process for which the generation problem is well-defined (i.e. not a design problem).

Imperative procedures (such as the constructive content generators in PCG) certainly offer constructivity; however our goal is to transparently define a space in terms of appropriateness, not to simply generate artifacts. By modeling the design space declaratively, a designer gains immediate access to several efficient generation procedures and also future-proofs their system against radical improvements in generative procedure design.

8.4.2 Constraint

Appropriateness (the defining concept of design spaces) manifests itself as constraints on the configuration of artifacts and their potential behavior. While many existing representations of design spaces only speak to local constraints on form, such as the top-down composition rules in the shape grammars emerging from architectural design [27], a desirable KR scheme will naturally support both local and global constraints on an appropriate artifact's form. In architectural grammars, local constraints, which

restrict the interaction between a component and its parent or immediate children, might ensure that a wall is augmented with either a window or door, but not both. The requirement that all west-facing walls have windows (perhaps to capture the light of sunset) cannot be expressed (without major reorganization of a grammar) as a local constraint in a system that allows room patterns to be rotated—the west-facing property of a particular wall is the product of the choice of rotations for all shapes that hierarchically enclose that wall. Similarly, the constraint that there be between two and five exterior doors on a building is another statement about a global property of a design that is only detectable from the analysis of the artifact as a whole.

With respect to function, the scheme should distinguish between those constraints that an appropriate artifact need only be capable of satisfying (e.g. that a puzzle have *at least one solution*, an *existence* constraint) and those that it must satisfy in all cases (e.g. that *every* valid solution to a puzzle involves a particular strategy of interest, a *universal* constraint). In electronic design automation, "high-level synthesis" is primarily concerned with these universal constraints on the function of appropriate artifacts, specifically the design of hardware with behavior that always matches the behavior of a reference specification in software [44].

8.4.3 Evolution

Because design spaces are used in an iterative process in which requirements are constantly shifting, design space representations should afford incremental knowledge engineering whereby new knowledge can be incorporated into a KR with minimal reengineering of existing definitions, the property McCarthy calls "elaboration tolerance" [140]. A much more formal description of elaboration tolerance in knowledge representation schemes is given in Aarati Parmar's dissertation [162].

Towards a computational account of the larger design process that modifies design spaces in response to experience with candidate solutions, design space representations should be amenable to machine learning and automated repair (or belief revision) techniques. That is, even though accelerating the inner loop of double-loop learning (by automatically constructing artifacts from a formally defined design space) is a means to help human designers engage in learning in the outer loop, KR schemes that afford direct mechanization of learning in the outer loop are clearly more desirable.

8.4.4 Analysis

Finally, as design spaces encode design knowledge, I desire that this knowledge representation support usages other than just direct synthesis. A suitably represented design space should afford not just the generation of artifacts, but also the generation of justifications for why that artifact is apt, appropriate, or fit to the current design scenario. That is, it should permit the reconstruction of design rationale with respect to the expressed conditions of appropriateness. Additionally, given an externally produced artifact in the same configuration scheme as those described by the design space, it should be possible to diagnose why that artifact would or would not have been generated from the current design space. Such discussions of the relation of a specific artifact to its degree of appropriateness are critical to learning in the outer loop. Even though appositional reasoning is primarily about the synthesis of artifacts, for the cost of encoding a formal sense of appropriateness, designers should expect something more than just synthesis in return.

While there are likely to be many KR systems that satisfy these requirements, what is important here is the link between the requirements and the context of design. Modeling design spaces in a satisfying KR should allow them to more fluently explore the space of alternative conceptions of what is appropriate than they could otherwise.

8.5 Means of Mechanization

Seeking to mechanize the synthesis of appropriate artifacts, one could browse through the available high-performance combinatorial search and optimization tools offered by symbolic AI (such as solvers for the standard problems reviewed in § 6.2). Although, at some level, nearly all of the available tools could be used to implement artifact generators, the problem specification language for different tools will make the task of capturing a design space easier or harder.

8.5.1 SAT Solvers

Boolean satisfiability (SAT) solvers mechanize the right kind of reasoning: given a set of design choices (represented by Boolean variables) and set of constraints (represented as formulae over these variables), a SAT solver can determine if the constraints are satisfiable and offer an example set of choices (an assignment of the variables) that witnesses this satisfiability. Assuming the other requirements on knowledge representation were satisfied, Boolean formulae would satisfy the constructivity requirements.

Regarding the constraint requirements, the formalism for SAT is expressive enough to encode any existential property (in propositional logic) that one could imagine. Adaptation of the algorithms underlying SAT solvers can be employed to handle universal properties as well [166]. Informally, two SAT solvers can be paired in a way that one solver (called the guesser) searches for a satisfying assignment of only the existentially quantified variables. Given a satisfying partial assignment, the second solver (the checker) searches for assignments of the universally quantified variables that would falsify the first assignment (proving, by existence, that the universal properties did not hold for that assignment). In this way, the second solver works as an oracle for the first solver. This is why the complexity class of problems involving both existential and universal constraints with exactly one alternation (i.e. 2QBF problems) is called NP^{NP}—the set of problems that are in NP when given access to an oracle for NP problems.

Even though SAT solvers can be nested to arbitrary depth to handle constraints of ever increasing complexity, the approach of modeling design spaces with the Boolean formulae that SAT solvers understand falls apart quickly, even in the face of only existential constraints. In this formalism, a constraint can say that "*variable66* is false or *variable102* is true or *variable5013* is true", but this is far from a natural encoding of appropriateness for any practical problem. In this formalism, it would be the designer's job to build all of these individual constraints from much higher-level ideas that exist only in their thoughts. In an island map generation example, stating that all regions adjacent to the edge of an n-by-n grid map must be covered in water (a simple enough constraint on form) must be encoded by $4n$ identically structured constraints over different variables. That is, the constraint is representable, but only very tediously.

In terms of my requirements on knowledge representation schemes, direct SAT encodings completely fail on the evolution requirements. It would be the designer's responsibility to reduce the artifact synthesis problem all the way to expressions over Boolean propositions.[3] This representation is far from elaboration tolerant—refining the form of one constraint means altering all of its propositional instantiations. This same result would apply for attempts to, say, capture design spaces as planning or scheduling problems.

[3]The use of SMT solvers, which allow propositions to be built from richer variable types than just the Booleans (e.g. real numbers or bit vectors), does not escape this failing either. Part of what is needed is a *first-order* representation language that allows a single rule to talk about entire classes of objects at once (using logical quantification). The proposition "`object_22_width < object_33_width`" is a statement about variables of a numeric type (whose values might come from infinite domains), but it still only talks about two specific objects: `object_22` and `object_33`.

8.5.2 Abductive Logic Programming

Revisiting abduction from another angle, abductive logic programming (ALP, previously reviewed in Chapter 6) is an extension of traditional (deductive) logic programming that integrates abductive reasoning with affordances for knowledge assimilation and default reasoning [103]. From the knowledge representation perspective, ALP importantly allows specifying abduction problems with statements in first-order logic: direct statements about properties of objects or the relations between objects (a marked improvement over SAT's abstract Boolean variables).

Where deductive logic programming is usually concerned with whether a complex query can be proven from given facts, ALP is concerned with producing (abducing) those new sets of facts that, upon deductive analysis, satisfy a query while not violating any integrity constraints. Because ALP is simply a symbol-manipulation formalism (as opposed to a model of explanatory reasoning), one can equally well use it to implement the constrained construction required of formal design space representations. The space of artifact configurations and potential interactions with an environment are captured with the set of abducibles, properties relating to form and function are captured with deductive rules, and the filtering of artifacts by appropriateness is specified via integrity constraints.

Unfortunately, most ALP systems have been based on Prolog. Thus, the search processes they employ are not complete for many problems nor do they exploit many of the now well-known advances in combinatorial search technology (such as those features reviewed in § 6.3). Being careful to avoid divergence or finding ways to encode the constraints of appropriateness in a way that yields acceptable performance often works against the requirements of evolution. Is there something more elaboration tolerant and better performing than ALP that provides a similar modeling language on top of first-order logic? Given the ability to express appropriateness via integrity constraints directly, can the vestigial appendage of an explicit goal (a feature of explanatory abduction) be removed?

8.5.3 Answer Set Programming

Providing exactly the step up from ALP that I request, answer set programming is founded on model-theoretic semantics in place of the operational or proof-theoretic semantics of traditional logic programming [7]. The notion of a set of supported models provides a cleaner mapping for the set of appropriate artifacts than does the order-sensitive, proof-oriented notions behind Prolog-based ALP systems such as PrologICA [167].

For an overview of the design and application of a state-of-the-art and

freely available answer set solving system, see "Potassco: The Potsdam Answer Set Solving Collection" [78]. For the purposes of this chapter, however, it is sufficient to understand that these tools support language constructs that allow the specification of a set of abducible facts (via choice rules), deductive definitions, and integrity constraints. For a more thorough treatment of using these representational features to model design spaces, see Chapter 10.

Beyond the ability to simply construct artifacts, unmodified answer set solvers have been used to implement concept learning [175], knowledge repair [75], and diagnostic reasoning over ASP representations [18]. Thus, modulo quantitative concerns such as run-time performance and soft concerns such as degree of usability, ASP satisfies every one of my requirements for a KR substrate in the service of automated, appositional reasoning.

8.5.4 Future Demands

ASP is not a panacea, however, as there are still many concepts that a designer may want to express in a design space model that are tedious to encode (though not impossible). Specifically, one should demand future instantiations of ASP (or alternative systems that could compete with it) to provide better support for non-Boolean properties and more natural encodings of problems with universal constraints.

The properties one can speak of in basic first-order logic are Boolean valued; that is, objects either do or do not have a named property. To name the "west-facing" property of a wall, one must use the proposition `direction(wall56,west)`—translated: "it is true that the *direction* relation holds between the object *wall56* and the `west` object." If there are only four direction values of interest, this is only a minor syntactic inconvenience. However, if directions are measured as an integer number of degrees (or worse, a real number), the number of instantiations of the `direction` predicate produced during propositional grounding in most ASP systems quickly grows unwieldy. To regain acceptable performance for this kind of problem (and recover from syntactic inconvenience), future systems should support a wider array of property types. The experimental hybrid ASP system Clingcon [80] (used in a few of the example design automation systems demonstrated later in this dissertation) augments basic ASP with support for large integer ranges and allows expressions like "`\$abs( direction(wall56) \$- west) \$$<$ 10`" for saying "`wall56`'s `direction` is within `10` degrees of the value of the constant `west`". Following the SMT (satisfiability modulo theories) community [23], it seems reasonable to expect native support for properties with programmer-friendly types such as bitvectors and arrays in the future.

Currently, the lack of these features in stable tools requires one to develop helper utilities that expand all combinations of the missing features before preparing the final knowledge representation. For example, bitvectors can currently be modeled in vanilla ASP systems only by producing a table of facts encoding the effects of various bitvector operations. Here is a fact recording one of 64 cases that an unrolled model of 3-bit vectors would require: `bitvector_and(0b110,0b010,0b010).`

Although many answer set solving systems are capable of expressing universal constraints (those that do are called *disjunctive* solvers), the manner in which these constraints are expressed is nowhere near as obvious as is expressing existential constraints. Universal constraints must currently be expressed through a rather counterintuitive technique called "saturation" [62]. This is unfortunate because universal constraints naturally arise in the description of appropriateness for interactive artifacts (certainly including game content and game rules). A much more natural encoding treats the universal constraint as a game: the guesser proposes an artifact, and a checker representing the environment proposes a response. Thomas Eiter and Axel Polleres proposed a general strategy for modeling elevated-complexity problems in this "guess and check" format [63], and others have improved upon this approach and generalized it to complex optimization problems (e.g. subset and Pareto optimization) [77]. However, these techniques could (and I claim should) be built into future variants of AnsProlog, the representation language for ASP systems.

To handle higher complexity models (those using universal constraints) with the tools readily available today, one answer set solver can be used as the the tester that filters that output of another solver used as a generator. To generate puzzles with no shortcutting solutions, for example, one solver can propose puzzles that have at least one difficult solution and another solver can be used to search for simpler solutions to the puzzle. This is exactly the same technique as stacking or nesting SAT solvers mentioned above, but at least this time a more friendly modeling language is available at each scale.

With today's available technology, particularly that of ASP, there are ample resources for developing natural-enough encodings for many design spaces for which synthesis and analysis can be completely automated.

8.6 Conclusion

Appositional reasoning is a fundamental piece of design thinking, stable across all domains of design. By unpacking the designer's appositional reasoning as a *non-explanatory* abductive process that aims to produce artifacts that are *appropriate* for a partially understood design scenario,

I have uncovered new constraints on satisfactory, domain-independent knowledge representations for design spaces. Linking these requirements to the affordances of abductive logic programming, I have focused attention on the clean representational substrate provided by answer set programming (ASP). Although ASP is not necessarily the only satisfying way to represent design spaces, I hope that future developments make encoding interesting conditions of appropriateness in this formalism easier and more fluid.

Chapter 9

Computational Caricature

Textbooks of game design advise designers to prototype and playtest as they iterate towards a final product. However, this high-level advice takes advantage of the human ability to fill in the gaps; the specific, concrete, and often project-specific prototyping and playtesting methods depend on undocumented reflective practices of designers. A mechanization of game design requires more detail about the myriad interconnected subprocesses of game design, but, as game design is a wicked problem, no such formal description can (ever[1]) be had. My strategy for overcoming this is to adopt a design-of-design approach, synthesizing formalized fragments of game design as it *could be* as opposed to how it objectively *is* at present.

I propose the creation of *computational caricatures* as a design research practice that aims to advance understanding of game design processes and to develop reusable technology for design automation. Computational caricatures capture and exaggerate statements about the game design process in the form of computational systems (i.e. software and hardware). In comparison with empirical interviews of designers, arguments from established theory, and the creation of neutral simulations of the design process, computational caricatures provide more direct access to inquiry and insight about design. Further, they tangibly demonstrate

[1]The implication here is that mechanizing game design it in itself a wicked problem. Imagine we had reliable automation for some large part of the design process for certain kinds of games. The relative cost and abundance of the kind of games producible with this automation would alter the landscape of what is appreciable as an interesting and non-trivial game, highlighting different kinds of games as the critical target of automation. This situation where partial solutions give rise to the next (and often more difficult) evolution of the problem is one of the hallmarks of wickedness. Nevertheless, it could be fun.

architectures and subsystems for a new generation of human-assisting design support systems and adaptive games that embed aspects of automated design in their runtime processes. In this chapter, I frame the idea of computational caricature, review several existing design automation prototypes through the lens of caricature, and call for more design research to be done following this practice.

9.1 Introduction

Design research generally intends to understand and advance the process of design or to transform the space of artifacts that might result from that process. In many cases, this takes the form of carrying out design projects with larger questions in mind. In this chapter, I describe a design research practice that targets game design, specifically addressing the questions asked by the first Workshop on Artificial Intelligence in the Game Design Process [67]:

> How can retrieval, inference, knowledge representation, learning, and search loosen the bottlenecks in the game design process? How can AI provide assistance to game designers and/or share the creative responsibilities in design?

In the design of game content artifacts (such as music, level maps, and story fragments), many automated systems draw heavily on AI search techniques [220]. In my game content generation work, I have shown how to use automated inference tools to generate game content from design space captured in a concise knowledge representation [195] (expanded in Chapter 10). Earlier, I employed learning and retrieval in a generative art installation (outside of games) that adapted to its audience to stay interesting over multiple months of interaction [200].

That the full spectrum of AI has already come into contact with automated content generation should be no surprise. The design of artifacts by machines has been a central topic of discussion in computational creativity [34], a field that uses theory and system building to advance understanding of both machine and human creativity. Recently, I proposed a connection between several computational creativity and AI topics (artificial curiosity, automated scientific discovery, and knowledge-level modeling) and the full context of creative game design, making predictions about the game design process as carried out by human designers and future machines [196].

I believe that following the program of computational creativity in the context of game design will continue to advance our understanding of the design process and unlock the building blocks of a new generation of

human-assisting design automation tools and currently-unreachable game systems that embed aspects of the design process in their runtime systems. Thus, the key question to ask is whether a machine can design, and, if so, how? Noted design studies researcher Nigel Cross echoes this sentiment [47, chap. 3]:

> Asking 'Can a machine design?' is an appropriate research strategy, not simply for trying to replace human design by machine design, but for better understanding the cognitive processes of human design activity.

This question can be tentatively answered by many means. Taking an empirical approach, Nelson and Mateas [156] interviewed expert game designers and, through playing the role of the interface between designers and automated reasoning tools, they drew several conclusions about the potential roles for AI in assisting game designers (including accelerating exploratory prototyping, performing early stage verification of designs, and supporting design-level regression testing). In contrast, my own development of a knowledge-level theory of creativity in game design (expanded in Chapter 16) draws weight from established theory in computational creativity, AI, and game design. Distinct from purely-empirical or purely-theoretical approaches, I believe the most convincing answers to the question of whether and how a machine can design games will take the form of computational systems, inspired by theory and constrained by the practicalities of what is realizable with today's computational resources.

I claim that building *computational caricatures* of the game design process (with all of the subjectivity and bias that the term implies) will provide direct access to insight regarding the role of AI in the game design process.

9.2 Computational Caricatures

The practice of building computational caricatures of the game design process is tied strongly to the sense of caricature familiar from visual art, and it holds special promise for game design (where aspects of automated design are increasingly being embedded into generative and adaptive games).

9.2.1 Visual Caricature

In visual art, caricature refers to a style of portraiture (creating images of a person with the intent to capture likeness, personality, and mood) in which distortion is used to make the subject more easily identifiable.

Caricatures attempt to be a *better* representation of a subject than an accurate depiction (such as a photograph or photo-realistic painting) would be. Through exaggeration and oversimplification, an artist makes a statement about what is most salient about that subject. Different artists may decide that different aspects of a subject are the most salient or, agreeing on saliency, they may decide to present the same aspects in different ways. As a vehicle for the artist's claims, these loaded portraits can express some ideas more quickly than the equivalent verbal description.

What makes a caricature notable is not just the choice of aspects of the subject that are emphasized but also any of the other choices that go into it. One artist may decide to borrow stylistic elements such as line weight or shading techniques from another when creating future caricatures. In actuality, a caricature captures two kinds of beliefs: "this is what is interesting about the subject," and (implicitly), "this is the most obvious way for it to be recognizably represented." In the remainder of the chapter I will be discussing a kind of abstract caricature that is related to visual caricature through analogy.

In this analogy, the subject of a caricature (conventionally a human face) is replaced by a cultural process. In the example systems I describe later, the subject of each caricature is a proposed process for automating some aspects of game design through the use of AI. The medium of the caricature (conventionally ink and airbrush on white paper) is replaced with computational media: software and hardware. The representation strategies used to carry out oversimplification and exaggeration (e.g. enlarging eyes, noses, chins, and ears while eliminating wrinkles and other blemishes) are translated into choices in a computational system's implementation: ignoring certain inputs, focusing on specific subproblems, or placing very strict requirements on potential users of the system.

However popular, the presence of large, airbrushed chins is not the essence of caricature. Instead, it is the general use of oversimplification and exaggeration techniques to make the subtleties of a subject more identifiable. As such, in the subsequent discussion of computational caricature, it is not my goal to understand the different ways in which cultural processes can be distorted in order to transform them into software. Instead, I want to highlight the use of a caricaturist's techniques in a computational medium for the purposes of making deeply technical claims more identifiable.

9.2.2 Procedural Portraits and Computational Caricature

In the general context of using AI in cultural production (which clearly includes the use of AI in the game design process), Mateas [138] proposed

the idea of building "procedural portraits": representations of human cultural processes in the form of realized computational systems.

Portraits of processes are easily recognizable as simulations. However, simulations greatly vary in the degree to which the simulator agrees with the simulated. ACT-R, a system-as-theory cognitive architecture, has been used to make quantitative predictions about human behavior [88]. Such simulations are an attempt at accurate modeling (equivalent to photo-realism in portraiture). By contrast, Weizenbaum's *Eliza*, a simulation of Rogerian psychotherapy in the form of a chatterbot, is best described (in Weizenbaum's own words) as a "parody" [227]. *Eliza* captures the "nondirectional" nature of Rogerian therapy in a near-stateless program that simply asks the user about the most recent input. Exaggeration and oversimplification are used liberally and openly in a charged simulation like *Eliza* whereas they would be rationalized or explained away in a neutral simulation like ACT-R.

By contrast to general portraiture, caricature allows us to be taken seriously in going after nuggets of truth (or at least proposed truth) without having gotten all of the surrounding details right. One of the values of caricature is rapidity of recognition. The kind of flash-communication afforded by caricatures makes them well suited as conversation-starters. In the context of system building, procedural portraits can make complicated and deeply technical arguments accessible, tangible even. The translation of a (too) familiar human practice into cold, machine crunching can make it unfamiliar enough that we gain the new perspective required to put some old questions to rest and ask more important ones.

In the same way that visual caricature encoded two kinds of beliefs (statements about the subject and statements about representational strategies), caricature in procedural portraits conveys two messages: "this is what is interesting about the human cultural process," and (implicitly), "this is the most obvious way to implement it on a machine." That is, we can often read more messages from a caricature than the caricaturist intentionally embeds. When a particular system architecture provides wider affordances for interpreting our message than expected, it is said to have "architectural surplus" [139]. Identifying sources of architectural surplus not only improves our ability to communicate through the building of systems, it paves the way for new kinds of systems that use AI to manipulate and create human-appreciable meaning automatically. This process becomes much more concrete when I zoom into the context of game design.

9.3 Computational Caricature of the Game Design Process

Computational caricatures are procedural portraits created with the values of caricature in mind (enhancing recognizability through exaggeration and oversimplification). As such, every computational caricature of the game design process will embed one or more exaggerated and often controversial statements about game design (alternatively thought of as propositions, claims, or hypotheses). Likewise, every computational caricature will make wildly simplifying assumptions in the process of reducing their perspective on design into an arrangement of code that is executable on a machine. The choice of which concerns to abstract away tells us about the intersection of what the caricaturist believes is salient and what is realistically feasible given the implementation techniques known to them. That a particular computational caricature of game design does not address a well-known aspect of human game design practice (perhaps learning through observing human playtesters) does not immediately imply a statement of unimportance. Instead, it might imply that the caricaturist knows of no promising architecture for realizing that practice in their caricature in a way that illustrates the intended claims.

The creation of a computational caricature (whether by a game designer, AI scientist, or design studies researcher) will always invite questioning beyond whether the statement the caricature seems to push is simply valid or not. They invite questioning into how accurately the system's knowledge representation models a designer's beliefs: into how the chosen algorithms (perhaps a specific kind of search) captures the designer's working processes; into how the machine's apparent values overlap with or diverge from those in traditional human practice (such as whether the value of a game flows from its internal structure, from empirical properties only observable in playtests, or some interesting balance of these); and so on.

Tanagra[2] [202], a demonstration of mixed-initiative design for platformer levels (described in detail later), invites us to explore the statement "humans and machines should produce game content cooperatively, asynchronously editing a shared design" (my words). Playing with this system suggests follow-up questions: When an infeasible design arises, should it always be the human's responsibility to resolve this? How do we know when a level design is finished, and can machines have an opinion on the subject? Should two humans cooperate to design levels in this way? The machine seems to be faster at verifying basic playability properties than

[2]In this chapter, the names of computational caricatures are set in italics as would be works of art and artifice.

the human; what other asymmetries are there and how should they be exploited in future design assistance tools?

A visual caricaturist's preference for certain line weights and shading techniques translate into a computational caricaturist's preference for code-level implementation details. Every computational caricature sets, reinforces, or breaks precedents in implementation techniques. Every system (whether the caricaturist is conscious of it or not) makes the implicit suggestion that future systems should use a similar problem formulation, programming language, or software library to realize a given concern.

Even the details of a caricature that are invisible without deep inspection become potential foundations for future systems and theories. In a review of content generation systems based on answer set programming that involved a code-level analysis [195], I found that every system assembled fragments of logic programs on the fly in a process of "dynamic program construction." Dynamic program construction, an architectural motif that appeared to its first users as an implementation detail, has emerged as a standard practice that future declarative, solver-based content generators should likely employ. In an interpretation of logic program fragments as encodings of a designer's beliefs about a design space, dynamic program construction implies a computational model of designers that heavily recycle domain knowledge between the analysis and synthesis phases of design.

Taken together, the building of computational caricatures is a design research practice that, while advancing personal goals (such as sharing opinionated claims about game design and the technology that should power future design automation systems), is centered on rapidly communicating deeply technical statements about game design and the role AI can play in its process. By building and sharing these systems, the caricaturist simultaneously accelerates the discourse around game design and produces tangible products of value along the way: systems that engage in a human cultural process. The reader should not be convinced by my claims alone, however. The best demonstrations of the value of the practice of computational caricature are the caricatures themselves.

9.4 Exemplars

In this section I review three (of many) examples of computational caricatures of the game design process found in the wild. None of these systems were designed explicitly as computational caricatures; nevertheless, it is possible to pick out statements about design that each system seems to be pushing and note where unintended details have also lead to revelations about game design. While many of the systems are the product of

joint work, I attempt to identify the caricaturist behind each system and speculate on their individual motivations. A sketch of how each of these examples works as a computational caricature can be seen in Table 9.1.

9.4.1 A Designer in a Box

Cameron Browne's *Ludi* is a board game designer in a box, or in Browne's words "a system for playing, measuring and synthesizing games" [21]. In its game synthesizing subsystem, *Ludi* uses a genetic algorithm to search the space of games expressible in the same language understood by its (also search-based) game playing subsystem. *Ludi* judges the value of potential games based on properties of typical playthroughs (records of simulated play between modeled players). These game properties are encodings of intuitive concepts like "completion," "duration," and "uncertainty" in a mathematical formulation.

Ludi seems to answer our question of whether a machine can design with a resounding "yes" and continues with the claim that "machines are human-competitive designers" (again, my words). *Yavalath* is a grid-based strategy game designed by *Ludi* (incidentally, named by *Ludi* as well) that is commercially available. On the popular site BoardGameGeek, *Yavalath* ranks[3] just above the family puzzle card game *Set*.

In reaching directly for the mechanical construction of a human appreciable game, *Ludi* is blissfully unaware of the potential feedback from human playtesters (it does not employ any), the design patterns used by any of the games outside of its very specific niche genre, and the need to redefine one's own representation system to eventually express new kinds of artifacts. These are not shortcomings of the system; instead they are the simplifications that made possible transforming *Ludi* from a thought experiment into a live system. The physical existence of *Ludi* and its product, *Yavalath*, materially changes the conversation around machine design: machines demonstrably *can* design real, valuable games, and what remains now is the question of *how* machines should design (or *why* machines design, how we should assign credit/blame between machine and programmer, and the multitude of related open questions).

Though the particular properties selected for use in *Ludi*'s game evaluation routine are presumably not part of the caricaturist's intended claim, *Ludi* demonstrates that there are mathematical properties that we should look for in two-player, strategic board games that do not immediately reduce to classical game theory. This is an unexpected result for human game design that also suggests a role for machines in the design of future

[3] `http://boardgamegeek.com/boardgame/33767/yavalath`

board games (as verifiers and automated explorers of localized design spaces).

For machine design, *Yavalath*'s terse construction in *Ludi*'s game description language raises interesting questions about how much the designer of a representation language affects the products of the systems that use it. Did Browne, who is independently an experienced board game designer, do the hard work behind inventing *Yavalath* by pre-selecting a representation that was rich with interesting game designs? Similar issues were raised by the reinvention of fragments of set theory by the mathematical discovery system AM [123], resulting in a discussion that has shaped the discourse around automated discovery for several decades.

9.4.2 Cooperating with the Machine

Gillian Smith's *Tanagra* is "a prototype mixed-initiative design tool for 2D platformer level design, in which a human and computer work together to produce a level." Upon starting, *Tanagra* presents the user (assumed to be a level designer) with a blank canvas and the basic ability to paint platforms into the game world. A large button labeled "Start Generator" is also present, and, when pressed, results in the near instantaneous filling of the canvas with familiar platforming elements (platforms with gaps, enemies and stompers), all placed to conform to the system's internal rhythm-inspired design theory and the limits placed on the player's avatar by the associated game's mechanics. With functionality in place for both completely unassisted human level design and completely automated level design, *Tanagra* invites us to reflect on the give-and-take between the two designers at work. In a typical demonstration, the human operator will draw only a few sparse platforms, place high level requirements on large gaps in the partial design to be filled, and perform minor aesthetic clean-up activities before declaring the level completed (leaving nearly all of the low level platform sizing, placement, splitting and recombining to the machine).

Just a few minutes of observing *Tanagra* interacting with a human user (level designer or otherwise) begins to raise the questions mentioned previously in § 9.3 (e.g. should we communicate only through design edits?). The idea of mixed-initiative design for geometric artifacts where the human operator expresses high-level constraints and leaves the machine to adjust the details is hardly new, perhaps originating in Sutherland's SKETCHPAD system [212]. Nonetheless, the existence of *Tanagra*, operating in the domain of platformer level design (much more familiar to game designers and game researchers than SKETCHPAD's mechanical design domain) transforms the abstract discussion about the distant potential of a mixed-initiative setup in future game design tools into a more concrete

discussion. We can now ask very specific, technical questions inspired by *Tanagra*'s implementation: Why do no commercial level design tools have any support for expressing design constraints (such as reachability of some location by the player's avatar), even without the ability to automatically satisfy them?

Where *Tanagra* zooms ahead of the industry standard platformer level design tools in support for intelligent assistance (perhaps providing more support than is needed as an exaggeration to ease one's recognition of the system's aims), it sharply oversimplifies other aspects usually required of platformer level design tools: exporting files for use in an external game (*Tanagra* lacks a "save" button), importing custom tilesets, and placing additional level details such as background art, flying enemies, optional paths, scripted and triggered events, etc. As a computational caricature, these distortions of platformer level design are welcomed in exchange for a tangible, interactive demonstration of a potential future for level design tools and a validation of the software architecture that made it possible.

On the implementation side, *Tanagra* is the composition of a reactive planner and a numerical constraint solver (an unprecedented choice in level design tools, to say the least) [202]. The caricaturist suggests that reactive planning is a useful top-level architecture for managing the mixed-initiative interaction and orchestrating the high-level search processes involved in geometry generation. However, she does not attempt to characterize level design as a constraint solving process (nor is the integrated constraint solver used, architecturally, as anything more than a supporting software library). However, that constraint solving (through answer set programming) played a key role in several content generation systems [195] suggests that the use of constraint solvers is actually a source of architectural surplus—it allows us to read the system as possessing a body of declarative design knowledge in addition to its procedural knowledge. That this surplus can be attributed generally to constraint solvers (which take a declarative specification of a problem's design concerns and produce a satisfying assignment of numerical and structural properties) and not specifically to answer set solvers is a direct result of *Tanagra*'s seemingly incidental use of a numerical constraint solver. Given the leverage provided by constraint solvers, we are inspired to think of alternate formalizations of game design that bring the creation and satisfaction of constraints to the foreground.

9.4.3 Imagining Gameplay

My own *Ludocore* (detailed in Chapter 12) is a "logical game engine" for producing queryable, logical models of the core rule systems of a game [199]. Many game engines intend to ease implementation of com-

plex videogames by abstracting away the details of 3D rendering, asynchronous resource loading, and other technical challenges. By contrast (and through extreme exaggeration), *Ludocore* is intended to ease implementing games that lack not only graphics and sound but also any form of live player input. Instead, *Ludocore* abstracts a videogame down to the central rules that govern the primary game state and how that state is affected over time by game events.

Using *Ludocore*, a designer can rapidly encode and iterate on the design of the most formal aspects of their game. In exchange for hyperformalization, *Ludocore* promises the ability to imagine gameplay for these as-yet incomplete games. The system's "gameplay trace inference" affordance allows a designer to ask for symbolic gameplay traces, from the vast space of potential low-level action sequences possible in a game, that satisfy arbitrary logical constraints. In this system, questions such as "is the game winnable?", "how would someone get from here to there without using this special item?", and even "how should I connect the various regions in my level design such that the player cannot escape without encountering all of my content?" all fit into a common representation.

Through exaggeration and oversimplification, *Ludocore* manages to realize an interactive prototype of a system that can imagine play for arbitrary games. This capability, of course, is conditioned on the "designer" being an experienced logic programmer, the rules of the game being primarily symbolic as opposed to numerical, the requested properties of gameplay being expressible in a subset of first-order logic, and having to wait unreasonable lengths of time for the results of certain kinds of queries. It is on top of this admittedly shaky foundation that *Ludocore* makes its statements about AI in the game design process: "Deeply modeling videogames requires capturing not just the game's rules, but (prepare yourself for a mouthful) also the configuration of the game's world, a body of assumptions about the kinds of players who might play the game, yet another body of assumptions about the situation in play that currently interests the designer, and, on top of that, the ability to reason through all of this to generate concrete gameplay traces which tell the designer something that was out of range of their human inferential ability." Simplified:[4] "deep computational modeling of gameplay is hard, but possible."

Ludocore's method of inference (using an answer set solver) is based on a system of exhaustive search. While this procedure for reasoning is alien to us (it is a poor model of how designers actually imagine gameplay traces), it does lead to example gameplay traces which we would not likely think of ourselves. Relatedly, *Ludocore*'s hyper-declarative programming

[4]These are my words as retrospective interpreter of my own system, years after its development.

language (which recycles Prolog syntax) is highly unfamiliar to most game designers (who are more likely to be familiar with an imperative programming paradigm, if any). Nonetheless, this logical representation seems to be particularly well suited for use by machines, and it has seen reuse in other design automation prototypes [155].

Where *Ludocore* is the most extreme instance of computational caricature that I review here (with the least generally accessible results), such extreme distortion allows for demonstrating interactions that would be entirely unreasonable in a neutral simulation of design processes. The more promising facets of an extreme caricature can be recycled into and evaluated in the context of more tame caricatures. For example, the "structural query" feature of *Ludocore* (which produces static world configurations as a result instead of traces of dynamic gameplay) was isolated and extracted as the less complicated practice of simply using answer set solvers to power game content generators (mentioned above). Ludocore's knowledge representation (but not inference techniques) were recycled in the *Biped* game prototyping tool (another computational caricature, expanded in Chapter 13) which added graphics, sound, and (most importantly) live player interaction with early-stage game prototypes [198].

9.4.4 Summary

As a caricature, each of the example systems above attempts to make some technical claim about AI in the game design process easy to recognize. In doing this, they make many simplifying assumptions, some of which are oversimplifications of the design process that should be overlooked while others are promising abstractions to be reused in future computational systems (either in subsequent caricatures or future design automation and game systems). Table 9.1 summarizes each of the above systems' status as a caricature with a list of claims, oversimplifications, and abstractions.

9.5 Conclusion

Towards the goal of better understanding the role of AI in the game design process (both for what this tells us about human game design and future design assisted *by* and embedded *in* machines), I have described how creating computational caricatures accelerates discourse and uncovers promising implementation techniques that exhibit architectural surplus. These caricatures (rich with subjective bias, exaggeration, and oversimplification) provide more direct inquiry into the questions of if and how machines can design than would more neutral simulations of the design process. Further, they appear to be more effective at exploring

Caricature	Claim	Oversimplifications	Abstractions
Ludi	*Machines can automatically design games that humans genuinely appreciate.*	• Ruleset invention boils down to sampling from a given, genre-specific grammar. • Games can be evaluated without human player interaction.	• Use simulated play to evaluate candidate levels. • Quickly reject potential designs with easily detectable flaws.
Tanagra	*Humans and machines should collaborate by asynchronously modifying a shared design.*	• Produced levels need never be extracted from the tool nor imported into a separate game. • Two designers can sufficiently communicate through design edits alone.	• Use constraint solvers to quickly resolve low-level design problems. • Enforce limits imposed by the game's mechanics during the level design process.
Ludocore	*Beyond worlds and rule systems, designers accumulate assumptions about players and play.*	• Designers think like SAT solvers, and they read/write logic programs fluently. • Arbitrary game rules are naturally expressed in declarative logic.	• Use pre-existing solvers to automatically generate content and imagine gameplay traces. • Quickly capture the high-level, symbolic mechanics of a game in a small amount of declarative code.

Table 9.1: A non-exhaustive summary of the example systems' claims about and oversimplifications/abstractions of the role of AI in the game design process. The claims are intended to be quickly recognized in the design of the system. Oversimplifications should be disregarded with respect to the claim (though similar simplifications may enable future caricatures). Finally, abstractions provide implicit advice for how to structure related systems as well as provide fodder for more closely interrogating the claim.

deeply technical ideas about the design process than do purely empirical or purely theoretical approaches.

I hope to inspire the creation of many more computational caricatures of the game design process, and I welcome exaggeration and oversimplification in the service of transforming distant ideas into computational systems.[5] I intend that this design research practice produces valuable results for AI, game design, and design studies generally.

[5]The strategy of simplifying a problem so that it can be realized as a computational system is not unique to computational caricature, of course. The computational prototypes built in everyday game design practice apply the same strategy. Whether one's computational prototype is also a computational caricature is a matter of audience interpretation—does the audience find a broader message in the system or, as it was perhaps intended, does it mainly serve to answer a project-specific question?

Chapter 10

Answer Set Programming for Design Automation

The goal of this chapter is to introduce a general framework for building design automation tools with answer set programming. In particular, it develops a practice around modeling design spaces as AnsProlog programs.

Recall from § 4.3 that one of the ways to describe design is as a *double-loop learning* process. The inner loop is concerned with finding solutions to a given, well-defined problem and learning how to iteratively improve solutions to better satisfy the constraints in that problem. The outer loop is concerned with learning that results in a transformation of the working problem definition, perhaps abandoning old constraints or establishing new criteria for preference amongst solutions. Design automation could address either or both of these loops. However, in this chapter, I focus on producing reliable automation for the inner loop: synthesizing artifacts that are appropriate with respect to formally defined conditions for appropriateness.

Although I intend this chapter to be immediately useful in mechanizing exploratory game design, the technical methods developed here are not intrinsically tied to game design at all. Meanwhile, the target audience for this practice is videogame designer-programmers. That is, I intend to offer something to people with a programming skillset suitable to implement parts of videogames and a design sense that permits reflection on what is really desired for a larger game design in response to experiences with example artifacts. Further, I require a willingness to learn a new programming language (not an unreasonable burden, as Chapter 17 demonstrates) and an openness for situational backtalk to

come from a machine's analytic reasoning instead of only live experience with hand-crafted designs.

In the first section of this chapter, I introduce ASP in a context through which it is rarely seen in AI circles, and I give a high-level overview of the practice of modeling design spaces with ASP. In later sections, I provide a series of basic AnsProlog programming tutorials, reviews of pre-existing design automation systems built on ASP, and conclude with a series of in-depth domain modeling tutorials that highlight how reflective practice concretely interacts with incrementally developing design space models.

10.1 ASP in Context

In AI research circles, it is common to describe answer set programming as a knowledge representation formalism. For example, this is the stance taken in Chitta Baral's book *Knowledge Representation, Reasoning and Declarative Problem Solving* [7], which defines the basic structures of AnsProlog. A recent discussion amongst ASP researchers concluded that "ASP is not a programming paradigm in a strict sense, but a formal specification language instead" [26]. While I agree that ASP involves formal specifications (this is precisely how answer set programs function as design space models), I strongly disagree with the first conclusion on the basis of the personal experience shared in this dissertation. I believe that treating ASP as a programming paradigm is essential for realizing broad and practical applications for this formalism that would never be explored otherwise. For example, Martin Brain, Owen Cliffe, and Marina De Vos' "A Pragmatic Programmer's Guide to Answer Set Programming" [17] establishes a body of knowledge associated with the craft practice of solving real-world problems with ASP.

Clarified, answer set programming is a programming *paradigm* (a peer to, say, lazy functional programming and message-passing, object-oriented programming), and AnsProlog is a programming *language* (distinct from, but analogous to Prolog, Java, and Python). What sets AnsProlog apart from most widely known and broadly applied programming languages (particularly its heavyweight cousin Prolog) is that it is a "little language."

10.1.1 AnsProlog as a Little Language

Jon Bentley [9] introduced the concept of little languages by starting with the concept of computer languages as something that "enables a textual description of an object to be processed by a computer program" and pro-

vided several examples of languages with a marked littleness. As Bentley's littleness is often (but not necessarily always) a result of carefully scoping the domain of use for a language, the concepts of little languages and domain-specific languages (DSLs) have since widely overlapped. Some of the common properties of little languages are the following:

- They are used as generated output of some programs and input to other programs; they function as a formalized interchange format.
- They are thought of as describing an object. For example, JSON (JavaScript Object Notation) is a little language for declaratively specifying nested list-and-record objects that finds use far beyond JavaScript.
- They are specific to a particular problem or problem domain. For example, GNU Make is a little language specifically for describing conditional build processes for software projects [208].
- They are often limited in theoretical expressivity. Regexes (regular expressions) and SQL (structured query languages) are little languages that intend to give up Turing-completeness (at least without unrealistic abuse of obscure features) in exchange for either very concisely specifying a problem or solving a problem via very efficient means.
- Finally, they may have a smaller or simpler array of language constructs or fundamental concepts that a programmer must learn. Although the little language DC [147], for expressing brief numerical computations to be carried out with arbitrary precision numbers, includes many of the same mathematical operators that might be found in Java, it offers no way of defining variables, opening files, or defining and importing modules of reusable code.

Examining each of these in turn, AnsProlog is often used as both an output and input format. While this chapter focuses on how to create AnsProlog programs by hand, each of the pre-existing design automation systems reviewed in section below involve assembling situation-specific programs on-the-fly from pre-authored program fragments.

When used as design space models, AnsProlog programs indeed describe an object: the space itself. Regexes and SQL programs function as descriptions of spaces in the same way. Programs in these little languages speak only about the properties of a potential result (a string match or a query result set) without saying how that result should be found in a separately defined corpus. In a interesting twist, AnsProlog programs define

a space of artifacts that should be enumerated without the presence of a pre-existing corpus over which to search.

Although answer set programming is targeted at complex combinatorial search an optimization problems, the structure of these problems is relatively unrestricted and the domains in which these problems may arise is quite widespread. In this sense, AnsProlog is a little language, but it is not strongly domain-specific. Similarly, while regexes are specific to the problem of sequence matching, they can be used in almost any area of application programming. Languages like GNU Make, however, are awkward to use for anything outside their intended target domain.

AnsProlog's traditional point of comparison (particularly in an AI context) is Prolog. The ability for answer set solvers to use the advanced combinatorial search algorithms that they do is a natural outgrowth of explicitly targeting ASP at problems in the complexity classes NP and NP^{NP} (problems for which more structure can be assumed than in arbitrary, Turing-complete languages). To express a broader array of programs, traditional Prolog includes a broad array of imperative (so called "extralogical") language features that complicate designing general, high-performance search algorithms for the language.

Finally, AnsProlog, at least as Baral defines it [7], has a very small number of language features and fundamental concepts. Although this makes the core features of AnsProlog very easy to understand, it makes the language too little to naturally express many realistic problems of interest. Accordingly, highly pragmatic answer set solving tools such as recent versions of the Potassco tools [76] add an array of optional features designed to make practical programming easier. Among these features are built-in arithmetic (otherwise not defined in pure, symbolic logic), complex aggregates (which allow speaking about how many facts in a collection are true in an answer set without encoding the counting logic with only logical *and*s and *or*s), optimization criteria (an extralogical preference between answer sets), and the integration of the imperative programming language Lua [97] (for performing arbitrary transformations on symbolic terms during the grounding phase of answer set solving). These advanced features, particularly the Lua integration, provide important *escape hatches*[1] that allow the programmer to temporarily break out of the formal logic paradigm without having to give up the benefits of staying in that paradigm (namely free, high-performance search algorithms for every domain of application).

[1] `http://c2.com/cgi/wiki?EscapeHatch`

10.1.2 Perspectives

From an AI perspective, an AnsProlog program is a formal specification of a space of answer sets (the set of stable models consistent with the program is read as a series of assertions in symbolic logic). From a design perspective, it is natural to look at a designer-specified AnsProlog program as a model of their design space of interest. Pairing this program with a specific answer set solver yields a fully-automated artifact generator that, from the perspective of computational creativity, is an example of a machine carrying out artifact generation activity in an area traditionally dominated by human effort. From the pragmatic programmer's perspective, AnsProlog programs are definitions in a little language that should be written by domain-specific programs and consumed by answer set solvers in a larger process that yields artifact descriptions for use in the problem domain. Each of these perspectives is an equally valid description, and all are relevant when building design automation with ASP.

10.2 Design Spaces in ASP

The key idea in this section, as setup in Chapter 8, is to encode models of design spaces with answer set programming. In the following subsections, I setup a general model–solve–interpret–refine cycle, and describe the high-level strategy for representing artifacts and spaces of artifacts with ASP. I follow this up with a discussion of modeled vs. unmodeled properties and an in-depth tour of each of the processes described in Figure 10.1.

A design space is an abstract and likely informal concept that exists only hazily-formed in a designer's mind. A designer is often capable of generating artifacts directly from this space through manual construction effort. Design automation seeks to replace this manual effort for many reasons. First, it may simply be too costly to involve a human designer in the creation of every artifact required (this is the cost reduction sought in many procedural content generation applications). Alternatively, a machine replacement for the designer may be explicitly required (as in play-time design automation for games where design automation becomes part of the formal mechanics of a game). Finally, a designer may simply not trust themselves to consider all of the design problem's constraints at the level of detail and reliability required by the domain. Thus, the dotted arrow in Figure 10.1, the *intent* to generate artifacts from a design space, represents the process I wish to realize via automation.

Following the arrow down from design spaces, a designer now has the option of modeling their design space as an AnsProlog program (using ASP as AI researchers intend it: as a formal specification language). Once encoded, invoking an answer set solver on this program will (after some delay for computation) yield answer sets. During the solving process, modern answer set solvers will carry out conflict analysis on any dead-ends in the search space that they encounter. This leads to the inner-loop learning of new constraints within the solver that are intended to speed up solving without altering the space of valid solutions. In a more interesting outcome, the designer may decide that, upon finally seeing all of their known conditions for appropriateness laid out before them in a formal representation, that there is a simpler, underlying structure that gives rise to many of these special-case requirements. Re-encoding the design space after this realization often results in a change to the set of artifacts the solver should produce.

With answer sets in hand, the process of interpretation, which may be carried out either by the designer or an automated system, constructs a domain artifact using the descriptions borne by an answer set. These artifacts are guaranteed to be appropriate with respect to the formal specification created previously, so a designer may simply choose to deliver

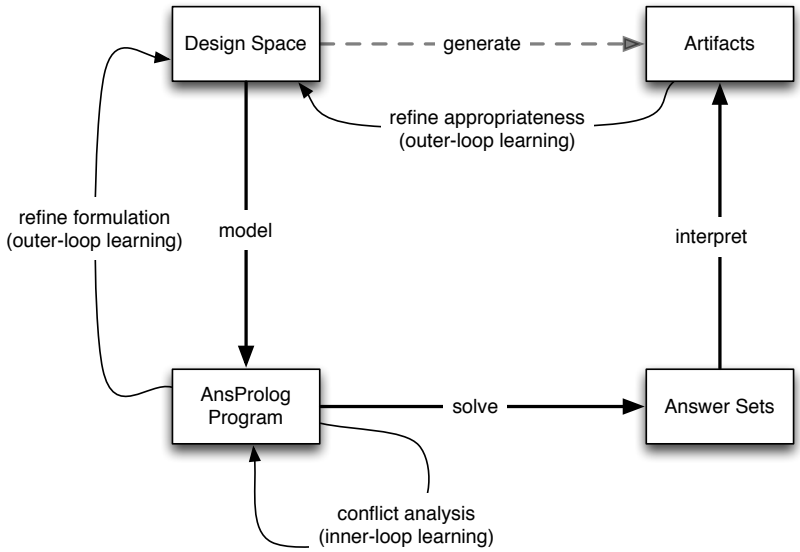

Figure 10.1: Modeling a design space as an AnsProlog program. The *intent* to generate artifacts from a designer's hazily defined design space is carried out by modeling the design space (producing a tentative formal specification of appropriateness), applying an answer set solver to produce answer sets, and then interpreting the logical facts in those answer sets as descriptions of an artifact to be used in the desired domain. Observing artifacts other than those that the designer would think to construct manually provides new opportunities for learning in the outer loop of the designer's double-loop learning. Likewise, spotting regularities in special cases encoded in the model can prompt a complete reformulation of how appropriateness is defined. This diagram is itself a refinement to the model–solve–interpret flow described in the practice documented by Brain et al. [17].

the artifacts to whomever requested them (not shown in the diagram).

In a much more interesting outcome, a designer may examine the resulting artifacts and take them as evidence that the working specification of appropriateness should be refined (perhaps to acknowledge an obviously inappropriate flaw). As this refinement leads to a redefinition of the working problem definition, it is an example of outer-loop learning in design. Another kind of outer-loop learning occurs when, examining their best attempt at formally encoding a notion of appropriateness, a designer realizes a fundamentally new formulation of appropriateness could express the working conditions in a different way (particularly, one that is more amenable to the kinds of refinements a designer wants to make).

Overall, this is a cyclic network of processes. From the designer's perspective, the cycles that relate to their working design space are the most important. As a result, whether the answer set solver really does carry out conflict analysis (or, assuming it does, whether it learns anything useful) is largely unimportant. This processes is included in the network primarily as a way to highlight the outer loops.

10.2.1 Representing Artifacts

Working backwards from artifacts in an effort to expose what needs to be modeled, I start with a high-level strategy for representing artifacts themselves. In order for an artifact to be constructed in the interpretation process, an answer set must describe that artifact in sufficient detail that interpretation is a well-defined problem (otherwise it becomes a design problem of its own).

The natural way to describe an artifact in AnsProlog is to generate facts that assert that the artifact has specific properties. For example, to be able to construct a physical wooden maze as an artifact, an answer set should describe where to place the start and finish cells of the maze and where walls or other obstacles should be placed. An answer set need not describe every physical property of the artifact (e.g. which type of wood to carve it from) if this information can be reasonably interpreted from the context of the problem. In addition to the minimal set of facts required to construct an artifact, we should ascribe additional properties to artifacts that allow us to describe that artifact's appropriateness. We want to include facts that describe the maze's shortest-path solution or record, for every location in the maze, which nearby maze is closest to the start cell. Some of these logical facts should be present in all valid answer sets (e.g. a fact attesting to the maze's solvability) or present in none of them (e.g. that there existed a shortest-path solution that was too short for the maze to be considered interesting).

Using the same strategy as traditional Prolog, these facts will be ex-

pressed with predicates in first-order logic—statements about the relation between symbolically named objects. For example, `start(5,6)` literally reads that the `start` relation holds between the `5` and `6` objects, but it takes little interpretation to also read it as a statement that the starting cell of a maze is located at cell `(5,6)` on a two-dimensional grid.

10.2.2 Representing Spaces

Representing a space of appropriate artifacts requires a means to abstract over which artifact we are talking about. There are three classes of rules in AnsProlog that allow space-level discussion of artifacts described by logical facts. Each of these classes is expanded with concrete examples in the next section.

At a high level, choice rules describe which sets of facts about an artifact *may* be assumed without justification. For example, in maze generation, one choice rule might assert that the solver is free to assume any cell may be the starting cell of a maze.

Deductive rules describe what facts *must* be derived in the presence of others. For maze generation, deductive rules describe the set of cells a player can reach given a particular arrangement of walls and starting cell.

Finally, integrity constraints describe the conditions under which a candidate artifact should *not* be considered appropriate. To avoid generation of mazes with uninterestingly long straight-line hallways, an integrity constraint can be specified that *forbids* candidates for which the long-hallway pattern can be detected (deduced). To ensure that generated mazes are indeed solvable, an integrity constraint might say that any candidate maze in which reachability of the cell with the finish property cannot be deduced should be forbidden (applying a kind of double-negation).

In this way, a designer-programmer, writing a handful of rules that shape the space of answer sets, can model a design space of artifacts whose conditions of appropriateness are expressible in logical rules.

10.2.3 Modeled and Unmodeled Properties

Integrity constraints provide very expressive, direct control over the properties of generated artifacts. In fact, they offer the convenience of a post-filtering process on a generator output without committing to a strict generate-and-test architecture. However, this control is only afforded over those properties of artifacts that a designer manages to model. In general, a design space model will only speak to a subset of the concerns that a designer knows to be critical for artifact appropriateness.

Although future answer set programming systems may include more expressive modeling languages and more exotic search algorithms, certain properties, it seems, are never likely to be expressed as part of an answer set program. The results of running an arbitrary user program or the elicitation of human feedback on candidate artifacts (an otherwise common ability for, say, basic and interactive genetic algorithms) are two such examples that would be left unmodeled in any ASP-encoded design space model.

Consider this example from the generative visual art domain: The output of the Neuro Evolutionary Art (NEvAr) system [132], is often interesting when its otherwise very abstract compositions resemble a human or animal face. NEvAr uses a neural network trained on audience feedback to capture a local, fuzzy sense of interestingness that can permit the system to generate many face-like images without an explicit logical model of how images come to resemble faces. This neural network evaluation (rich with floating-point arithmetic) is a natural fit for a generate-and-test architecture, but it is impractical with the largely symbolic ASP framework.

Where NEvAr gets away without committing to a complete model of visual aesthetics, a similar strategy can be used in ASP. This strategy involves replacing a broad but inexpressible concern with a collection of narrow, special cases that help a generator to avoid generating artifacts with easily describable flaws. In the maze generation domain, while "difficulty" seems impossible to express as a simple logical formula, one can still add rules that prune away potential mazes that are insufficiently difficult for obvious reasons: the solution is much too short, the solution does not involve changing direction often, one cyclic pattern of choices (e.g. left-left-right-repeat) accounts for nearly all of the required choices, there are no other paths which get anywhere close to the solution (potential "garden paths" that a player might enjoy avoiding), and so on.

Following such a strategy, while difficulty is still an unmodeled property, many of the design implications of considering difficulty to be a part of a maze's appropriateness are addressed via other modeled properties.

10.2.4 Modeling, Solving, Interpretation, and Refinement

Modeling, solving, and interpretation would seem to form a pipeline that directly implements generation of artifacts. However, these processes, in concert with refinement, will usually be carried out many times in the course of developing even relatively simple design space models.

Modeling

In a first-pass through the modeling process, the most important thing for a designer to invent is a schema for describing artifacts as answer sets. If a simple Boolean flag can describe some feature of an artifact, that flag can be represented by a simple proposition (a fact from a predicate without parameters). Using richer, multi-parameter predicates allows expressing most data-like artifact components. For example, sets can be described by facts asserting whether an object is in the set or not; sequences can be represented with assertions about which object follows another in the sequence; tables can be represented with assertions about which object occupies the cell at a particular row and column; and graphs are naturally represented by sets of node labels and tables of edge labels. Code-like components of an artifact can be readily built from data-like representations. For example, a reactive event handling system might be modeled as a table that maps event identifiers to sequences of actions, where actions are symbolic identifiers that have special, interpreted meanings in the final execution environment.

This artifact description schema may be refined later, but some preliminary schema must be established to begin the other processes shown in Figure 10.1. To sketch a representation schema without yet using any AnsProlog rules, a common strategy I use is to simply describe one example artifact with a collection of facts. This fact-only program is a minimally functioning design space model: it will yield an artifact description (just one) when an answer set solver is run on the program. I will return to how a designer goes about modeling a design space beyond the first pass when I discuss the refinement process later.

Solving

Once a minimally functioning design space model is defined, a designer's direct interaction with the answer set solver is minimal. For easy problems (those involving a small number of logical objects and admitting a large number of solutions), a designer may never need to tweak the solving process.

If a new facet of appropriateness is modeled that presents the solver with a much more difficult search problem, the designer can, without any modification of their design space model, try out different solver configurations (usually by altering simple command line flags, even without expert knowledge of what exactly these flags control). These configurations can radically reshape the solver's internal search process. Altering the use of constraint propagation and learning, randomness, restarts, choice of search heuristic (from a predefined set), and the use of various kinds of pre-processing phases can have a dramatic effect on the time it takes the

solver to find solutions and the incidental style of the first valid solutions encountered. Except for clearly marked options for early termination, none of these algorithm tweaks alters the completeness of the underlying search process (its ability to find and enumerate all solutions before terminating in finite time) or the space of valid answer sets that might be emitted.

Interpretation

Starting from the hand-crafted answer set from the trivial design space model (the one we created to sketch out the representation schema), a designer needs to decide on a process for interpreting this set of facts as an artifact. In the very early stages of exploratory design, this interpretation may simply take the form of reading the facts and imagining the artifact. In almost all cases, however, there is a much more natural visual representation of an artifact that affords instant recognition of the artifact's structure without tedious decoding of logical statements. As such, a designer should take some time to find or produce a bit of visualization infrastructure that will accelerate (or even completely automate) interpretation in future cycles. In the rest of this chapter, I use a utility called *Lonsdaleite*[2] (based on GraphViz [66]) to develop visualization logic in the same iterative refinement cycle as the associated design space model.

For artifacts that are not primarily visual, e.g. an interactive artifact like a generated mini-game ruleset (as explored in Chapter 14), the interpretation process involves building the infrastructure that can load answer sets into a more complex system (e.g. a game engine that will interpret the ruleset). As this can be a serious engineering challenge by itself (to the point of bogging down the iterative design space modeling process), complete automation of the interpretation process is not always a designer's goal. So long as it is easy to try out individual answer sets in the target environment (perhaps through a small amount of copy/paste and manual editing), it is reasonable to leave some human effort in the interpretation process. After all, during iterative development, the designer should be present to make the observations that trigger refinement of the design space model.

Refinement

Refinement is the process of taking concrete issues raised by preliminary artifacts and transforming these into new or reformulated constraints on what makes an artifact appropriate. Because of the inherent ill-definedness (and occasional wickedness) of design problems, the process of

[2] `https://github.com/rndmcnlly/Lonsdaleite`

refinement will not necessarily converge on an objectively defensible definition of appropriateness for artifacts. Instead, refinement will guide a designer to producing a design space model that blends the appropriateness-derived constraints and preferences that the designer knows about with what is naturally expressible in AnsProlog, efficiently solvable by answer set solvers, and reasonable to interpret without too much engineering effort. ASP is not a panacea for modeling design spaces once and for all, but it does provide some powerful opportunities for offloading some of cognitive burden that arises in appositional reasoning.

Upon interpreting the one answer set from the initial, trivial design space model, the first natural refinement move is to replace one of the hand-entered facts with a basic choice rule. With no other constraints, the artifact described by solutions to this crudely refined design space model are quite likely to demonstrate some obvious design flaw that should be forbidden in the future. This prompts an elaboration: the inclusion of an integrity constraint. Proceeding in this way, modeling new generative possibilities by rewriting hand-entered facts with choice rules and carving away wild combinations with the addition of pattern-matching deductive rules and artifact-pruning integrity constraints, the designer carries out a kind of sculpting process on the design space.

Expanding the sculpting metaphor, it is important to note that a designer's sculpting of a design space involves both additive modeling (adding material to a base) and subtractive carving (removing material from a base). That is, with ASP, designers manipulate design space models as if they were clay (equally amenable to additions and subtractions, but unwieldy at large scales) as opposed to the modeling-focused medium of wire-and-plaster sculpture or the carving-focused medium of stone sculpture.

In a generate-and-test architecture, a designer must conceive of their design space in a form that involves performing all of the additive moves first (building up a broad but not wastefully large base of material) before performing all of the subtractive moves (pruning away material without recourse to make slight additive repairs). With the constructs of AnsProlog, a designer may declare that one choice rule is conditioned on the result of a deductive rule (perhaps the same deductive rule that feeds into an integrity constraint). This expresses the idea of interleaved additive and subtractive processes that could not be partitioned into two discrete phases while also not committing to a particular ordering of operations as a directly constructive process might.

The result of many incremental refinements is often not the pristine design space model a designer seeks. Instead, and often in my experience, the result is a complicated assemblage of overlapping concerns and special case patches. Even the most elaboration tolerant representation scheme

can only go so far because refining elaborations trend toward growth. This result is not a failure, though, because this distended design space model functions as a detailed record of newly discovered constraints on the appropriateness of artifacts.

From here, the refinement process will most likely involve a complete rewrite of a design space model, perhaps representing artifacts in different kinds of terms that make expressing the space of special case concerns much simpler (in terms of lines of code, solver running time, and the conceptual complexity of the sheer number of named concepts involved). These occasional restarts are hardly exceptional; in fact, they are to be expected both as part of sculpture and as part of a designer's reflective practice. When enough anomalies are found with respect to a designer's current understanding of appropriateness, eventually a kind of revolution occurs (exactly a *paradigm shift* in Kuhn's terms [117]) that compacts the new knowledge into a more coherent framework. The cost of these revolutions is decreased when the body of knowledge being reformed has already had algorithm-design concerns stripped out, as is the case when building design space models with ASP.

10.3 Programming Tutorials

In this section, I walk through three self-contained example programs: a classic introductory problem involving only propositional logic, a solution to the graph coloring problem emphasizing the process of grounding, and a program for constructing Golomb rulers that introduces optimization over solutions and hints at the promise of emerging results in ASP research. Even if the reader is familiar with, say, graph coloring, the discussion I provide along with each example should be enlightening for what it reveals about how to think and write in AnsProlog, issue commands with Clingo, and visualize outputs in a lightweight manner. These examples are intended to provide a general literacy with ASP before introducing the practice of using it to model design spaces.

10.3.1 Hello Soggy World

The classic introductory example for AnsProlog involves reasoning about a soggy patch of grass. The grass may have been wet by the natural effects of rain, the artificial influence of a sprinkler, or it may turn out not to be wet after all.

To begin, we will invent a few propositions (predicates that take no arguments) to represent the different statements that might be true of the world: *rain* means "it rained;" *sprinkler* means "the sprinkler was on;" *wet* means "the grass is wet;" and *dry* means "the grass is dry." These four propositions give rise to a space of sixteen ($2^4 = 16$) possible world descriptions (potential answer sets), including many suspicious combinations such as simultaneous wetness and dryness or dryness despite the presence of either of the wetting processes. Table 10.1 depicts the space of worlds expressible with these propositions as a truth table. As rules are introduced, the space of valid answer sets is whittled down to just two possibilities.

To describe the effects of rain and sprinklers on grass, we need some rules. The first is a choice rule (with an empty body), asserting that any number of items from the aggregate may be true:

```
{ rain, sprinkler }.
```

Offering this single line of code to an answer set solver and asking for all of the answer sets yields four solutions: nothing; just `rain`; just `sprinkler`; or both `rain` and `sprinkler`. From a generate-and-test perspective, this line has implemented our generator. We never hear of `wet` because we have not yet provided the rules for deducing it. These definitions provide the means to analyze the grass in each of our generated worlds:

```
wet :- rain.
wet :- sprinkler.
```

`rain`	F	F	F	F	F	F	F	F	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**
`sprinkler`	F	F	F	F	**T**	**T**	**T**	**T**	F	F	F	F	**T**	**T**	**T**	**T**
`wet`	F	F	**T**	**T**	F	F	**T**	**T**	F	F	**T**	**T**	F	F	**T**	**T**
`dry`	F	**T**	F	**T**	F	**T**	F	**T**	F	**T**	F	**T**	F	**T**	F	**T**
Program 1	✓				✓		✓		✓		✓		✓		✓	
Program 2		✓					✓				✓				✓	
Program 3		✓					✓				✓		-	-	-	-
Program 4	-	-			-	-	✓		-	-	✓		-	-	-	-

Table 10.1: Answer sets in the soggy grass problem. The top half of this table shows the inclusion of the four propositions in each of the 16 potential world descriptions for the soggy grass problem. The bottom half marks whether or not each potential world is a valid answer set after the addition of extra rules. Program 1 involves the choice rule alone. Program 2 shows the result of adding the three deductive rules. Program 3 shows the results after adding the sprinkler-shutoff logic, and Program 4 shows the result after requiring the observation of wetness. Dashes indicate answer sets that were explicitly forbidden by an integrity constraint.

```
dry :- not wet.
```

Adding these rules does not change the number of valid worlds from above (four), but it does provide each with additional detail. The answer sets are now: `dry`; `rain` and `wet`; `sprinkler` and `wet`; and `wet` with both `rain` and `sprinkler`.

Suppose we know something else about the scenario: an automatic shutoff system prevents the sprinkler from wastefully running while it is raining. The following rule seems to express this idea, but it is not grammatically valid:

```
not sprinkler :- rain.
```

The problem is that `not` is not allowed in the head of a rule. The intended effect of this invalid rule is to forbid worlds where `sprinkler` and `rain` are simultaneously true. Thus, an integrity constraint is the appropriate choice. Adding the following constraint narrows the combined program down to only three possible worlds:

```
:- sprinkler, rain.
```

Having encoded all of our relevant knowledge about the patch of grass, we can finally ask an interesting question: how did the grass get wet? To ask this, we want to prune away all worlds that are inconsistent with our observation. In general, if `goal` is a proposition that is true when our interest is satisfied, the integrity constraint "`:- not goal.`" accomplishes the required pruning. This is like saying "throw out the world if it isn't one that interests me." Thus, adding the following integrity constraint

yields only those possible worlds that provide answers to our query:

```
:- not wet.
```

There are now only two possibilities: it is wet because it rained, or it is wet because the sprinkler was on (but not both). With only four propositions and a handful of rules, it is not hard for a human to mentally sort through the sixteen combinations (particularly when the rules at play are familiar from everyday experience). In some of the systems employing ASP for design automation that I discuss later, the grounded answer set program involves hundreds of thousands of propositions (with the number of potential combinations being exponential in that number) and nearly as many grounded rules encoding the mechanics of a by-definition unfamiliar design problem. So, while the query-posing setup given in this introductory example might seem too simple to be useful, the same techniques scale to queries well beyond a human's capability to provide equivalent answers.

10.3.2 Graph Coloring, Complexity, and Visualization

To get an idea of how complex problems can be captured in very small answer set programs, let us now work through a solution to the classic graph-coloring problem. In this problem, it is our job to assign every node in a graph one of a small set of colors so that no pair of nodes connected by an edge is assigned the same color. Deciding whether a graph can be colored with a given number of colors is exactly the decision-problem version of the chromatic number problem from Karp's famous list of NP-complete problems [104].

See Figure 10.2 for an example graph that we will attempt to color. To represent this graph and the available colors, we should use a small collection of facts:

```
node(a).
node(b).
node(c).
node(d).
edge(a,b).
edge(b,c).
edge(a,c).
edge(c,d).
color(red).
color(blue).
color(green).
```

In the parlance of guessing, deducing, and forbidding, we will use a choice rule (with cardinality bounds) to guess a unique color assignment and an integrity constraint to forbid conflicting assignments. The following two lines are sufficient to solve the graph coloring problem for any

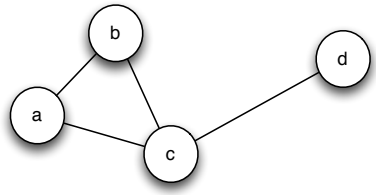

Figure 10.2: This small graph is the example input to our graph-coloring program.

graph and set of colors:

```
1 { assign(N,C) : color(C) } 1 :- node(N).
:- edge(N1,N2), assign(N1,C), assign(N2,C).
```

This brief program is, in essence, a reduction from the graph coloring problem to the problem of determining whether an answer set program has an answer set. Thus, it should be intuitively clear why answer set solving is itself an NP-hard problem. Whether our two-line solution yields a system with attractive performance or not depends on whether the algorithms and heuristics used by the answer set solver (which are readily swappable without disturbing the encoding) can exploit the structure of our chosen problem encoding (which can be altered without immediate concern for the solver).

Supposing I have put the collection of facts into a file called `instance.lp` and the two rules in a file called `encoding.lp`, I can ask Clingo to generate a coloring of this graph by invoking this command in my terminal:

```
$ clingo instance.lp encoding.lp
```

That command results in the output below:

```
Answer: 1
node(a) node(d) node(c) node(b) edge(c,d) edge(a,c)
edge(b,c) edge(a,b) color(red) color(green) color(blue)
assign(b,green) assign(c,blue) assign(d,red) assign(a,red)
SATISFIABLE

Models       : 1+
Time         : 0.000
  Prepare    : 0.000
  Prepro.    : 0.000
  Solving    : 0.000
```

From this, we can tell several things. Most obviously, Clingo has determined that the problem was satisfiable, that the graph is not inherently uncolorable. The note that there are "`1+`" models means that while only one solution was generated, there still remains a possibility of finding more (indeed, there are 12 solutions to this problem). If the graph had

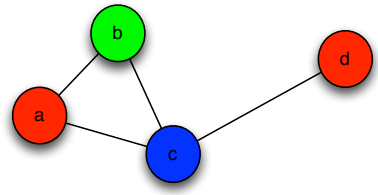

Figure 10.3: The first solution identified for coloring the graph from Figure 10.2 with three colors.

contained just two more edges (linking `d` to `a` and `b`) then Clingo would have reported it unsatisfiable after a small amount of search.

Next, looking at the text of the answer set itself, we can spot terms that describe the color assignment: "`assign(b,green assign(c,blue) assign(d,red) assign(a,red)`". Note that the answer set contains more than just the guessed assignments; it contains every statement that is true in that world (including the facts we simply stated in the problem instance). Visualizing this solution's color assignment on the original graph yields Figure 10.3.

Finally, while the preparation (grounding), preprocessing (startup), and solving (search) times for this particular instance are all too small to measure, monitoring these times is useful for diagnosing slowdowns that would delay insightful feedback when using ASP in exploratory design for larger problems.

Factoring a concrete problem into an instance description (composed only of ground facts) and an instance-agnostic problem encoding (composed mostly of rules with variables) is a common development strategy. During iterative development, a very small testing instance (for which solutions can be manually verified) can exercise the workings of the same rules that will apply to larger instances. The adaptation of the general rules to a particular instance happens in the process of instantiation. To see how the two rules in our problem encoding are expanded in light of our small test graph, we can ask Clingo to dump the grounded program as text with this command:

```
$ clingo instance.lp encoding.lp --text
```

This results in the output shown in Figure 10.4 where copies of a rule are created for all value combinations for the variables mentioned in the body of the rule and aggregates (expressions in braces or brackets) have been expanded with all value combinations for the variables quantified by the `:` operator. As a rule of thumb, the more variables a rule involves and the larger the domains of each of those variables, the more instan-

```
node(a). node(d). node(c). node(b).
edge(c,d). edge(a,c). edge(b,c). edge(a,b).
color(red). color(green). color(blue).
1#count{assign(b,blue),assign(b,green),assign(b,red)}1.
1#count{assign(c,blue),assign(c,green),assign(c,red)}1.
1#count{assign(d,blue),assign(d,green),assign(d,red)}1.
1#count{assign(a,blue),assign(a,green),assign(a,red)}1.
:-assign(a,red),assign(b,red).
:-assign(a,green),assign(b,green).
:-assign(a,blue),assign(b,blue).
:-assign(b,red),assign(c,red).
:-assign(b,green),assign(c,green).
:-assign(b,blue),assign(c,blue).
:-assign(a,red),assign(c,red).
:-assign(a,green),assign(c,green).
:-assign(a,blue),assign(c,blue).
:-assign(c,red),assign(d,red).
:-assign(c,green),assign(d,green).
:-assign(c,blue),assign(d,blue).
```

Figure 10.4: This variable-free program is the result of instantiating the graph-coloring program for the small example in Figure 10.2.

tiations that rule will have (and the longer grounding will take). This expansion is simultaneously the mechanism that allows a very small answer set program to represent a very large grounded problem and the mechanism that most often leads to grounding performance problems for inexperienced programmers.

Had we tried to solve the graph coloring problem using a SAT solver (which would have equally shielded us from writing high-performance combinatorial search code), it would have been our responsibility to create a low-level problem formulation at least as complex as the output in Figure 10.4. Encoding the set cardinality constraints, such as those expressed with the bounds in the choice rule for assigning colors, as Boolean formulae is itself a subtle problem when bounds involve values other than zero and one. Naturally, the solver's low-level input would need to be constructed programmatically. Instead of maintaining a custom problem-building program, ASP offers us the ability to define our high-level problem declaratively (here, in two lines) and to leave the repetitive process of assembling a collection of (mostly) Boolean formulae to the grounder. It bears mentioning that the majority of ASP systems are distributed with the grounder and solver as separate programs and that Clingo's unification of these tools is a rare, if modest, convenience.

In the grounded program text, note how references to the *node* predicate have been elided in the instantiation of the choice rule. Because the grounder can determine that the *node* predicate is always true for every node (as we directly stated as much with facts), the grounder can safely drop those terms from the body of any rule that mentions them

without altering the space of valid answer sets. The semi-naïve evaluation algorithm [76] that powers this optimization is also capable of producing exact answers to natural[3] encodings of the HORNSAT problem (akin to SAT without the use of negation) or the circuit value problem (akin to Boolean circuit satisfiability with all inputs known)—both are P-complete problems. Thus, for some problems, particularly those that involve simply analyzing a given artifact instead of synthesizing a new artifact, it is sometimes an attractive strategy to address the problem with the grounder alone, treating the grounder as a general-purpose Datalog engine.

As another exercise in working with rules involving logical variables, let us now think about how to visualize the output of our graph-coloring tool. Knowing that there are terms like "`assign(a,red)`" in Clingo's output, we could write a custom parser that looked for these terms (along with those describing nodes and edges) and use them to build an input file for Graphviz,[4] an open source graph layout and rendering tool. This could be done in about a hundred lines of Python. This is not an unreasonable amount of effort, but it seems quite out of balance with respect to the slender two lines of AnsProlog that implement the coloring logic. Consider this program (in the file `viz.lp`) that captures the key visualization design choices in a few more rules:

```
graphviz_graph_type(graph).
graphviz_node(N) :- node(N).
graphviz_edge(N1,N2) :- edge(N1,N2).
graphviz_node_attr(N,fillcolor,C) :- assign(N,C).
graphviz_global_node_attr(style,filled).
graphviz_global_node_attr(shape,circle).
```

My visualization tool Lonsdaleite[5] will render graph diagrams for any problem that includes these `graphviz_*` terms in its answer sets. To recycle this tool for other ASP projects, we need only maintain the small collection of rules that deduce a graph description from the problem-specific elements. For our graph-coloring program, this command will use Lonsdaleite to display a version of the diagram in Figure 10.2 in my web browser:

```
$ clingo instance.lp encoding.lp viz.lp | lonsdaleite -cub
```

Because our visualization rules were all either facts or Horn clauses, we

[3]Through some *unnatural* encodings, problems well beyond HORNSAT can be decided exactly. This is demonstrated by the encoding of a Turing machine that can be found in the Potassco overview article [78] without the use of negation, choice rules, or any other feature that would leave a choice for the solver.

[4]`http://www.graphviz.org/`

[5]Lonsdaleite, also called hexagonal diamond, is an allotrope of carbon. With all of the familiar carbon crystal names like "graphite" and "graphene" already used by *several* other graph visualization tools, I was obliged to pick a less common name (that even I misspell every time) in order to get the graph pun.

can use the same rules to create visualizations with the grounder alone. In this example command, imagine `example_coloring.lp` contains several `node`, `edge`, and `assign` facts generated by a previous run of our coloring program:

```
$ clingo example_coloring.lp viz.lp --text | lonsdaleite -ub
```

During the rapid iteration of exploratory design space modeling, I have found it convenient to freely mix the details of my test instances, problem encoding, and visualization logic in a single file where all can be updated at the same time. I often delay factoring out the reusable pieces until I have a better understanding of a problem domain and begin looking to integrate my ASP-based solution with a larger system.

10.3.3 Golomb Rulers, Optimization, and Numerical Constraints

Let us test our working knowledge of answer set programming with a slightly more complex problem: constructing optimal Golomb rulers. A Golomb ruler is a ruler with marks at certain integer-spaced positions such that the distances between every pair of marks is distinct. Figure 10.5 shows a Golomb ruler of order 4 (the number of marks) and length 6 (the length of the longest span it measures). For a thorough treatment of the often misunderstood complexity of problems relating to Golomb ruler construction and a surprisingly diverse list of real-world applications of Golomb rulers, see "On the Complexity of Constructing Golomb Rulers" [143].

The Golomb ruler construction problem is naturally parameterized by two integers: order and length. Instead of specifying length directly, we will instead specify the total number of markable positions on the ruler to allow for rulers of potentially shorter length (for real-world applications, shorter often means cheaper and thus better). To specify numerical parameters, we use the following AnsProlog syntax:

```
#const order = 9.
#const positions = 45.
```

These statements are not rules per se. Instead, they instruct the grounder to perform a symbolic substitution in the rules defined later (akin to the *#DEFINE* directive in the C preprocessor).

Now, using these constants, we assert that there is a set of positions, and that a precise number (equal to the `order` constant) of them are to be marked. Note the use of the ".." syntax to concisely specify a large family of facts (`pos(0)`, `pos(1)`, ...`pos(45)`) at once. Asserting `marked(0)` anchors our generated rulers at the origin. We do not assert `marked(positions)` because this would force our ruler to span the entire set of given positions

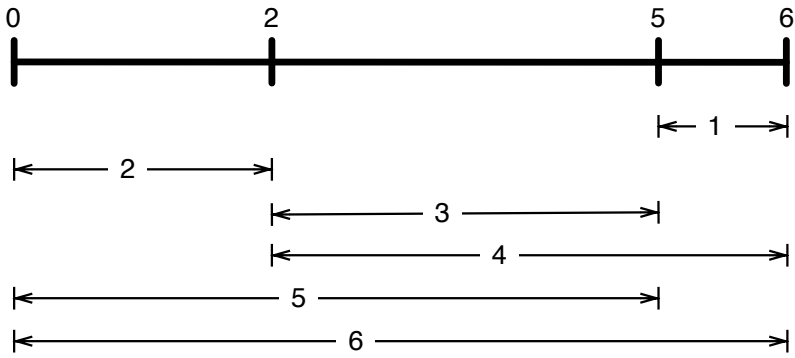

Figure 10.5: Example Golomb ruler of order 4 (the number of marks) and length 6 (the largest distance between marks). This is an *optimal* Golomb ruler because there are no shorter rulers of the same order.

(we would like to allow shorter rulers of the same order).

```
pos(0..positions).
order { marked(P) : pos(P) } order.
marked(0).
```

If we were to invoke Clingo on the program as presented so far, we would find several different ways to scatter the desired number of marks across the given space of positions. However, many of these solutions would fail to have the key property of having all distances distinct. To enforce this, we must deduce which distances can be measured by the ruler (and where they are measured from) and forbid rulers for which any distance is duplicated. As expected, this entails the use of a deductive definition and an integrity constraint. In the code below, "`marked(P1;P2)`" is a convenient contraction of "`marked(P1), marked(P2)`" and "`measure(P2-P1,P1)`" conveys the idea that "the distance `P2-P1` is measurable on the ruler starting at position `P1`." The integrity constraint states that a ruler is invalid (i.e. not a Golomb ruler) if a distance can be measured from two (or more) positions on the ruler.

```
measure(P2-P1,P1) :- marked(P1;P2), P1 < P2.
:- pos(Dist), 2 { measure(Dist,P) }.
```

The seven lines of code so far implement a functioning program for constructing Golomb rulers. To apply this program to the search for an 11-mark ruler, we can override our constant definitions on the command line while also instructing the solver to use an alternate[6] heuristic (one

[6]VSIDS does not perform remarkably better than the default BerkMin heuristic—

derived from the Chaff SAT solver [148]):

```
$ clingo golomb.lp -c order=11 -c positions=72 --heu=vsids
```

For a given number of marks, there is a minimum length of Golomb rulers of that order. While these optimal lengths have been tabulated up to order 26 (the result of an ongoing massive distributed computation effort[7]), we can easily verify some of these optimal lengths with a small addition to our answer set program. These next two statements deduce the set of positions covered by a ruler (the size of this set is the length of the ruler) and instruct the solver to minimize the size of that set. Specifically, the `covers` predicate is true for a position `P1` if there is is a `marked` position `P2` that is greater than it.

```
covers(P1) :- pos(P1), marked(P2), P1 < P2.
#minimize { covers(P) }.
```

The minimize statement is not a logical rule. Instead of adding detail to or pruning away potential answer sets, it provides a metric by which the answer sets can be ordered. When asked to compute more than one solution, the solver uses the metric value of the previous solution as a constraint on the value of the next solution. In this way, successive solutions are required to improve in quality until the problem is made unsatisfiable (proving optimality of the previous solution) following a branch-and-bound strategy.

In this specific problem, minimizing the size of a set was sufficient to express our optimization criteria. However, a more general syntax for optimization allows the use of weights.[8] In the following statement (equivalent to the *minimize* statement above), I describe the metric to be optimized as a weighted sum over the set of covered positions (with all weights set to unity):

```
#minimize [ covers(P)=1 ].
```

Supposing we did not know the optimal length for rulers of order 10, we might guess 72 (the optimal length for order 11 rulers) as an upper bound and ask Clingo to search for the optimal value with this command ("`0`" means "compute as many answer sets as possible"):

```
$ clingo golomb.lp 0 -c order=10 -c positions=72
```

Printing out solutions as they are found, the metric progresses 71,

this simply demonstrates the command line syntax for altering the choice of heuristic.

[7] `http://www.distributed.net/OGR`

[8] In addition to weights, prioritized (lexicographic) optimization is also possible. By including terms from multiple predicates in the same statement (including negative literals) a wide array of optimization criteria are expressible. For the complete syntax of optimization statements, see the Potassco guide [78]. For more complex optimization criteria (such as Pareto efficiency) made possible through advanced metaprogramming, see "Complex Optimization in Answer Set Programming" [77].

69, 67, 66, 61, 60, and then 55 before confirming there are no shorter Golomb rulers. Indeed, 55 is the known optimal value for rulers of order 10. This search, taking 32 seconds on a single core of a 2006-era server processor, involved a problem with 2,772 propositional variables related by 10,439 constraints. The size of this problem is mostly due to the *measure* predicate which grounds to $O(p^2)$ cases for a problem with p possible positions.

Because the Golomb ruler problem is explicitly focused on numerical variables and the distinctness of their differences, we should consider exploring the use of the experimental hybrid constraint answer set solver called Clingcon [80] that uses the high-performance GECODE library [184] internally for constraints over integer variables. The following Clingcon program sets up an integer variable for each mark, enforces that marks are assigned in increasing order, and asserts that the difference between every pair of marks is distinct using a global constraint. Finally, it instructs the numerical constraint solver to treat the value of the last mark as the metric for optimization. For now, it is not important to fully understand the Clingcon-specific syntax involving *$*, as the hybrid solver is used very rarely in the rest of my work.

```
#const order = 9.
#const positions = 45.

$domain(0..positions).
mark(1..order).
mark(1) $== 0.
mark(M-1) $< mark(M) :- mark(M), M > 1.

$distinct { mark(M2) $- mark(M1) :mark(M1):mark(M2):M1<M2 }.

$minimize { mark(order) }.
```

Because this program's grounding does not directly depend on the number of potential positions on the ruler (only the number of marks), we are free to be much more lenient with our guessed upper bound. Here is the command for determining the optimal length of an order 10 ruler using numeric constraints:

```
$ clingcon golomb_numerical.lp 0 --csp-num-as=0 \
       -c order=10 -c positions=10000
```

This variation yields the desired solution in less than a quarter of the time of the previous variation (on the same hardware). While it is likely that using GECODE directly from a custom C++ program would provide some speedup, the ability to mix the Boolean constraints of traditional ASP with numeric constraints (demonstrated later in this chapter) while staying within a very concise modeling language is what makes Clingcon remarkable.

10.4 Existing Design Automation Examples

In the worked examples above, I ignored the complexity of integrating ASP-based artifact generators into a deployable system. In the remainder of this section, I review three instances of ASP-based artifact generators employed in the context of larger systems that operate in unique domains, rich with widely varying and realistic constraints on the form and function of appropriate artifacts.

10.4.1 Diorama

DIORAMA[9] is an open source, comprehensive map generator for the real-time strategy game *Warzone 2100* (Pumpkin Studios 1999) that generates natural-looking, detail-decorated terrain maps with desirable gameplay properties. Internally, DIORAMA uses ASP to solve two gameplay-critical subproblems in map generation. Externally, the system collects a set of map design requirements from the user with standard user interface widgets (drop-down menus, numerical spinners, and checkboxes) and allows them to repeatedly sample different maps satisfying their constraints. An example of DIORAMA's user interface and a schematic diagram of a generated map are shown in Figure 10.6. Saving a map generated with this tool injects it into the game's map library where it can (without additional user intervention) be interpreted in context: as a 3D world populated with the buildings and doodads relevant to gameplay (where outputs look like the one shown in Figure 10.7). After this, the user may generate additional maps or use a traditional map editor to refine the generated map to their taste.

The cliff structure of the terrain in a *Warzone* map has a strong impact on gameplay by blocking land-vehicle passage between tiles with sufficiently different height values. In the first phase of the map generation process, amongst other details described later, a coarse grid of cells is assigned numerical height values with terms like `cellLevel(X,Y,Height)`. From `cellLevel` facts, passage between tiles can be derived using deductive logical rules. Additional user-settable constraints enforce the presence of interesting geographical features: undulating plains, smooth seabeds, raised or sunken player bases, and a prescribed number of unreachable mountaintops. Logical rules for describing undulation and other modeled properties, of course, are also included in the system's internal AnsProlog program.

The problem of placing player base locations and oil wells (the drivers of the economy in *Warzone*) is tightly coupled with the terrain generation problem; cliffs provide a natural, indestructible defense against direct,

[9] DIORAMA documentation and source: `http://warzone2100.org.uk/`

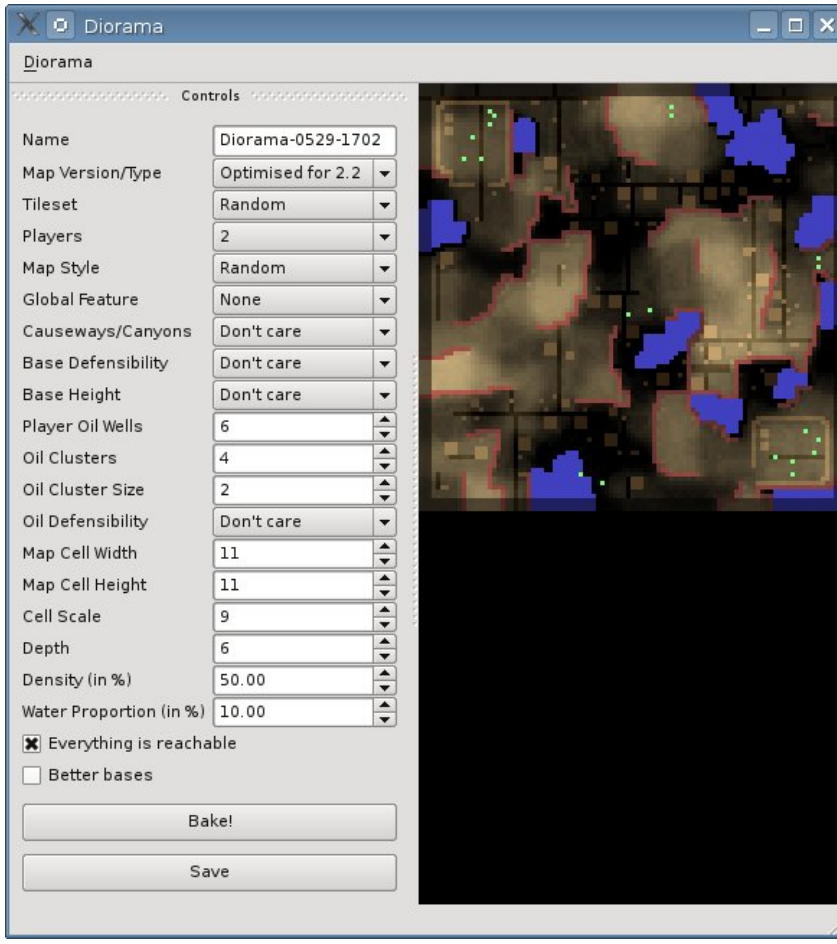

Figure 10.6: Screenshot of DIORAMA's user interface. The "Bake!" button can be pressed to repeatedly sample alternate maps that satisfy the user's specified design constraints.

Figure 10.7: Generated player base for *Warzone 2100*, depicted with in-game 3D graphics. Note how the far corner of the base's enclosing walls have been warped from linear to conform to the nearby cliffs in the terrain.

ground-based attacks. Accordingly, DIORAMA combines these concerns into a single design space model, effectively solving for terrain, base, and well placement all at the same time. The `baseLocation(Player,X,Y)` and `oilLocation(Well,X,Y)` predicates complete the minimal structure of the generated artifacts at this stage.

To express a preference amongst the myriad possibilities that conform to terrain design rules, the AnsProlog program declares that the solver should first (globally) maximize the distance between player bases, and then, amongst solutions with that maximal base distance, find a placement of oil wells that maximizes the minimum well–well and well–base distances. Additional constraints optionally enforcing partial cliff-based defensibility of bases and wells are also active in this process.

I learned that ASP actually replaced a genetic algorithm solution for this search-intensive phase in a previous version of the generator [19]. Without an ASP solver, such multi-layer global optimization would be difficult to express with a procedure that had separate generate and test phases.[10]

Though DIORAMA used a fixed priority scheme to layer different levels of preferences at this stage, a system of numerical weights could have (but was not) used to express trade-offs between preferences at the same level, e.g. that one point of resource distance unfairness is a safe trade for two points of cliff defensibility unfairness. In this case, the AnsProlog program would have asked the solver to simply enumerate maps that maximize the sum of the trade values.

Having committed to bases and oil wells on a naked height map with known traversability, DIORAMA performs several non-ASP passes to improve the aesthetics of the final map. To break the unnaturally straight lines of the original cell grid, the map is warped in a post-processing phase and traversable cell boundaries are smoothed. A plausibly designed road network is overlaid which visually guides players from their bases to oil wells and randomly placed abandoned towns. These phases primarily add visual flair and cannot break the map's gameplay, so they proceed in a non-backtracking fashion, enriching the map in-place.

Generating a smoothed version of the abstract terrain is an example of a subproblem in terrain generation for which ASP is not particularly applicable. Instead of selecting artifacts with particular properties from a vast-but-finite space, continuous smoothing is a process more easily

[10] If the generation subsystem employs a complete algorithm that is guaranteed to eventually produce every possible artifact, global optimization is feasible. The tester should simply examine all artifacts and, in the end, return one that was not dominated by any other artifact. Altering this ideal into an efficient system involves either immense integration of generator and tester (yielding a backtracking-like setup) or giving up global optimization and, with it, any guarantees on the properties of the system's output relative to the stated design goals.

described imperatively in a general purpose programming language. That is, while geographical features are modeled properties at the (coarse) cell level, DIORAMA leaves the aesthetics of (fine) tile level details unmodeled and trusts the imperative passes to cover them in a feed-forward manner.

In a gameplay mode of *Warzone* that allows players to start a match with generous bases pre-built for them, maps may include a customized layout of essential buildings and fortifications. DIORAMA has an option that will cause an ASP-based base layout phase to be injected into the generation process after the terrain has been warped and smoothed. At a high level, `location(Building,X,Y)` facts are generated using choice rules for the origin point of each building, from which blocked tiles are deduced and the number of lined-up buildings is deduced. Overlapping buildings are easily rejected with an integrity constraint. The size of a boundary zone around blocked areas is calculated and globally maximized, secondarily maximizing the modeled tidiness property of the arrangement. The resulting layouts have an organized-looking grid layout, fit to locally warped terrain features while ensuring ground units can still navigate around the base.

In order to perform terrain-adapted base layout, this phase needs access to terrain details already committed in the previous phase. DIORAMA dynamically assembles a specialized generator for the situation by concatenating AnsProlog fragments with facts describing the committed world details. Such dynamic construction of design space models is very common, and it represents a kind of adaptability to design problems that ASP provides that goes far beyond parameterization via numeric parameters (which are also used in DIORAMA, e.g. in specifying the number of players for which a map should be designed).

In this system, ASP was used to encapsulate two search-intensive subproblems as feed-forward generators in the context of a larger, multi-paradigm generator. Overall, DIORAMA's generative procedure is a non-backtracking pipeline of generate-only components; the test procedure, if any, is human inspection of the final maps. Even though map design does not involve a finite space (as terrain heights have a continuous domain), ASP was still able to solve the most difficult subproblems (particularly those related to critical gameplay properties), leaving the other phases of generation free from any hand-written backtracking or generate-and-test search.

10.4.2 Anton

The state-of-the-art automatic music composition system, ANTON (Boenn et al. 2011), uses ASP to produce detailed melodic, harmonic, and rhythmic musical compositions informed by a seamless blend of local and global

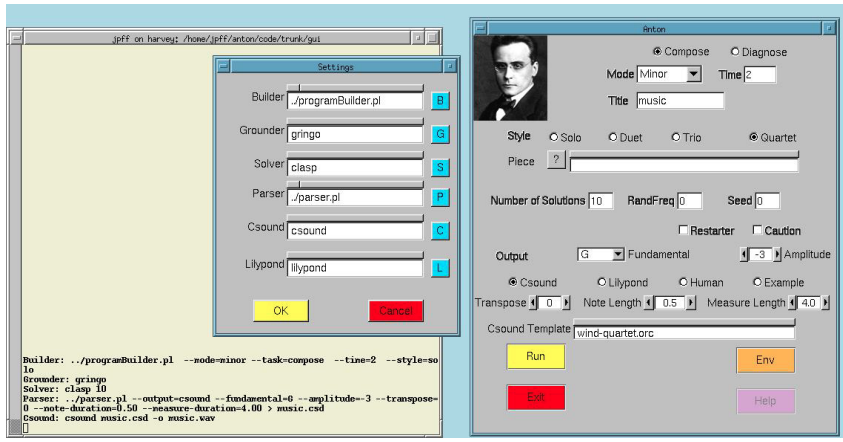

Figure 10.8: Screenshot of ANTONS's user interface. Don't let the brutalist Linux/`fvwm` visual aesthetic dissuade you—when invoked on the command line, ANTON is pretty slick. In one output mode, the system uses the LilyPond (`http://www.lilypond.org/`) tool to produce beautifully typeset (or, in musical terms, engraved) sheet music. In another, it renders high-quality wave file outputs using Csound (`http://www.csounds.com/`).

composition knowledge. Although the musical content is not consumed in the context of any game (instead in the form of printed music notation, rendered audio tracks, and even real-time performance), the application provides examples of potential new directions for PCG systems. A screenshot of ANTON's graphical user interface is shown in Figure 10.8.

So far, I have primarily focused on using modeled properties in integrity constraints, that is, requiring or rejecting properties of generated artifacts. The Palestrina rules of composition (a codification of renaissance counterpoint in western tonal music theory) in ANTON serve a secondary purpose: in addition to composing new music, the system can use the same rules to diagnose and informatively report flaws in an externally provided composition. It might report the message "middle note of triad doubled" and identify a particular part and time in the composition, or "invalid minor harmonic combination" citing another location. Different composition styles (such as solo vs. quartet) carry different error pattern definitions. As a body of *computational music theory*, it is natural to expect ANTON's declarative knowledge to be used in both analysis and synthesis [16].

The different modes of operation in ANTON are supported by a pro-

gram builder that dynamically assembles AnsProlog fragments to craft a design space appropriate to the problem at hand. In composition mode, with a certain style and other configuration set, the design space contains musical scores (represented with `chosenNote(Part,Time,Pitch)` terms) that strictly conform to the assembled rules. In diagnosis mode, a logical encoding of a human-created composition is combined with style rules to produce a design space of critiques (including `error(Part,Time,Reason)` messages), as opposed to musical passages.

Automated critique of human-designed artifacts can, of course, help clean up these artifacts, but it can also be used to understand and debug a design space (even suggesting particular constraints to relax in the iterative design of a generator). Producing intelligent answers to a query such as "Why won't you generate artifacts like *this one*?" is not an activity supported by any generators for game content so far, but perhaps it should be in the future. Answers to such queries can even point out inconsistencies in the background theory used by the generator (for example, it is possible that the Palestrina rules contain contradictions that are not obvious to human examination).

The ability for the composition mode to also accept partial human-created pieces allows the system to serve a number of purposes beyond tabula-rasa generation: supplying fragments allows the constrained generation of music that ends with a certain pattern or embeds a certain motif; supplying one part (the score for a single voice) and not others allows solving for a harmonization consistent with a given melody; and specifying only a chord progression allows for natural melodic improvisation. By incrementally committing to (or forbidding) details suggested by the system, the user can carry out a mixed-initiative interaction with the system (in the sense explored by the Tanagra system [202]). Nearly all ASP-based generators that employ dynamic program construction will gain this give-and-take capability by default.

Though the connection is subtle, there is a similarity between the music composition task and the task of designing platformer levels (rhythm has been identified as part of this link [38]). ANTON composes music by selecting a series of local moves a part should take: step up, leap up, rest, repeat, etc. The trajectory through the space of absolute pitches and times is derived as a side effect of these local moves. Rhythmic and melodic composition rules are often written in terms of the global trajectory and have a complex relationship to the local moves. Similarly, Launchpad, a platformer level generator, works by selecting a sequence from a set of local actions the player should take: move, jump, and wait [203]. The player's trajectory is made concrete by generating geometry that must fit rhythmic density and style constraints. The line critic in Launchpad can be seen to be functioning as a melodic composition rule,

with ANTON's harmonic composition rules speaking to the gameplay of as-yet-unconsidered platformer levels with interacting, parallel tracks.

10.4.3 Toast

TOAST [45] is a *superoptimization*[11] system for machine code. Given a sequence of machine instructions (completely defining the function of appropriate artifacts), TOAST uses ASP to find the absolutely shortest possible alternate sequence of instructions that computes the same result on all possible inputs. In this scenario, the form of artifacts is the literal sequence of instructions. The function of artifacts is to compute the same result as a particular larger program.

In reflective analysis of their problem domain, the developers noticed several fragments of input instruction sequences that were strictly suboptimal (they should never appear in an optimal program, regardless of what it was trying to compute). They noted that, if several of such sequences were collected, they could be included as new constraints (on form this time, independent of function) to accelerate search on problems other than those in which the constraint was originally discovered. When a piece of the function of an artifact can be captured as a constraint on form, much less effort needs to be expended in ensuring that artifacts emerging from the more restricted space have the appropriate function.

One of the interesting results of superoptimization is surprising new uses for machine instructions thought to be well understood. In a classic example of the unintuitive results, Crick offers the example of the C library function used to calculate the sign of an integer (the signum function):

```
int signum(int x) {
       if (x > 0) return 1;
       else if (x < 0) return -1;
       else return 0;
}
```

While a straightforward compilation of this function uses two conditional branch instructions amongst other instructions (e.g. numerical comparisons), an experienced assembly programmer can likely reduce this to just one conditional branch. Mechanical superoptimization, however, reveals a wildly unexpected result involving neither comparison nor conditional branch instructions. The following solution cleverly exploits sensitivity to a carry flag in the different SPARC V7 addition and subtraction instructions:

[11] In the world of compilers, "optimization" refers to result-preserving transformations of a program intended to make it incrementally faster or smaller. "Superoptimization" is the (somewhat unfortunate) term for compiler optimizations that result in an *optimal* program.

```
! input in %i0
addcc %i0 %i0 %l1
subxcc %i0 %l1 %l2
addx %l2 %i0 %o1
! output in %o1
```

Just as the above systems used ASP in several different ways to implement a larger tool, TOAST uses ASP to solve several different search-intensive subproblems. Given an example input and output pair for a sequence of instructions to be optimized, TOAST constructs a design space model capturing the set of all finite instruction sequences that compute the same output on the given input. Appropriateness (the focus of appositional reasoning), here, is the condition that the new instruction sequence be shorter than the original and agree with the original when evaluated on at least one input. In another subproblem, a candidate instruction sequence is tested for complete functional equivalence with the original instruction sequence. In this subproblem, input values are the artifacts of interest and the conditions of appropriateness are that different output values result from the two instruction sequences—this subproblem involves trying to find evidence of non-equivalence in the form of an input that witnesses the difference. If no such witness can be found, the two instruction sequences must be functionally equivalent.

In exchange for declaratively modeling the semantics of a set of machine instructions just once, the developers of TOAST gained the ability to perform several different program synthesis and analysis tasks. In any other domain where the artifacts of interest involve code-like constructs, an architecture similar to the one used in TOAST can be used to diagnose faults in given artifacts, produce new artifacts with equivalent function to given artifacts, or seek out the points of divergence between the behavior of two artifacts—all without inventing any new search algorithms.

10.5 Modeling Tutorials

Whereas the previous tutorial section (§ 10.3) was intended to bring a designer-programmer up to speed in programming with AnsProlog, this section is intended to convey design space modeling idioms. Those programming techniques are put to work in developing sketches of pre-existing game content generators. Each of these sketches functions as a computational caricature: they make a claim about what were some of the most salient features of the design spaces considered in these past systems. They also make an implicit claim about how to quickly convey these features. This claim rises from the style of caricature used in each sketch: building each one in ASP.

Although reflection and iterative refinement are critical pieces of the design process, the following examples distort this situation somewhat by presenting each scenario as a well-defined content creation problem. This is an intentional choice that I hope will serve to make these sketches more interesting to compare with the original systems than might a more accurate depiction of exploratory design (which might iterate off into territory distant from the inspiring examples).

10.5.1 Chess Mazes

My first sketch is inspired by Daniel Ashlock's chess maze generator, presented in "Automatic Generation of Game Elements via Evolution" [5]. This example models a design space in which only one type of choice is required.

Overview

Ashlock's chess mazes are a simple kind of puzzle that is played on a generalized Chess board (an n-by-n square grid). In this puzzle, it is the goal of the player, who controls one piece, to move their piece from a square on one side of the board to a certain square on the opposite side. The twist is that the player is not allowed to move their piece through any square that is under attack from (or covered by) any of the other fixed pieces placed on the board by the puzzle designer.

An example maze is shown in Figure 10.9 in which the player's rook, starting at the square marked S and navigating to the finishing square marked F, must not cross any square attacked or occupied by any of the five knights.

Automatically designing chess mazes like this, given the size of the board w and the number of knights k as parameters of the problem, involves only deciding the x/y position for each of the knights so that the

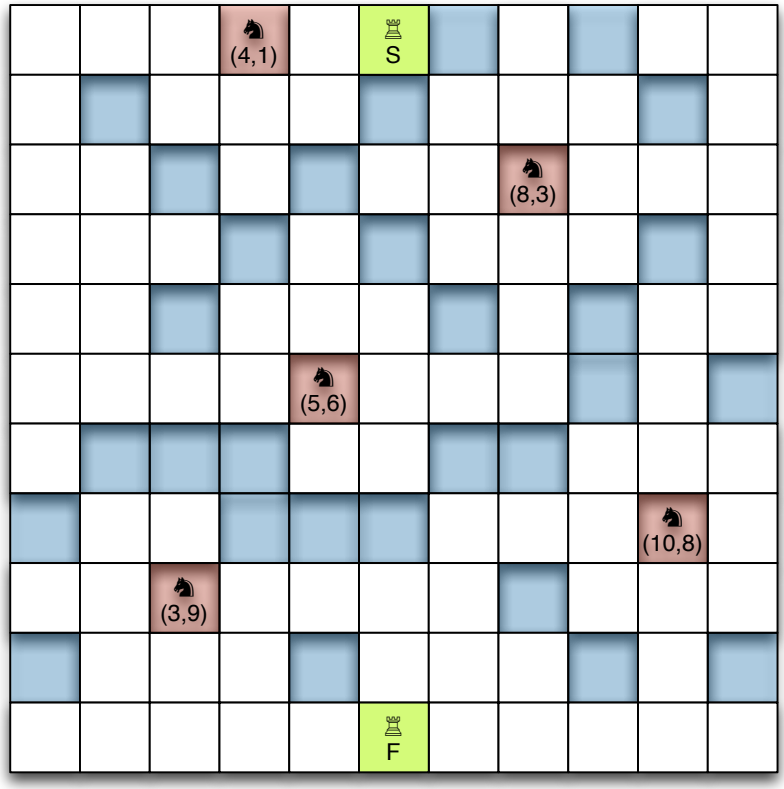

Figure 10.9: Chess maze example. The configuration of this maze was taken from Ashlock's paper [5], but it was also reproduced by the system described in this subsection. When giving this maze to a player to solve, no clues would be given beyond the position of the knights and the starting/finishing squares. The covered and attacked cells are intentionally made visible here to illustrate the structure of the maze as reasoned about in the design space model.

puzzle's definition is satisfied. Specifically, we are obliged to check that there exists some solution to the puzzle we design and, for a bit of finesse, we want to be able to control the length of the simplest solutions (as a proxy measure for the difficulty of the puzzle). Let d denote the desired minimum solution length.

Development

To begin, we should establish the problem parameters so we can refer to them later in the design space model. I have taken the default values for these parameters from the example maze shown in Figure 10.9 (an example taken from Ashlock's work):

```
#const w = 11.
#const k = 5.
#const d = 23.
```

Next, we should establish a vocabulary for the board. How are squares measured, and how are they arranged?

```
dim(1..w).
square(X,Y) :- dim(X;Y).
```

In Chess, knights move by taking two steps in one direction and one step in an orthogonal direction, forming the iconic "L" shaped pattern. On a two-dimensional board, a knight can move to up to eight possible destination squares. We will encode the relative offsets of these destinations with facts for the `attack_pattern(DeltaX,DeltaY)` predicate. However, instead of simply listing all eight cases (and risk forgetting or duplicating a case), we will use the ";" and ";;" pooling operators to generate all eight cases concisely.[12].

```
attack_pattern(-1;1, -2;2 ;; -2;2, -1;1).
```

A similar syntactic trick works to define the set of directions the player's rook can travel. Here, we are capturing the idea of up/down/left-/right movements on the grid by encoding a vector. For example, `direction(1,0)` refers to going in the direction of increasing `X` and holding `Y` constant—going right.

```
direction(-1;1,0 ;; 0,-1;1).
```

Before we start specifying the variable parts of these puzzles, there is one more bit of background vocabulary we need. Saying a rook can go in any of the four cardinal directions is not precise enough. We need to say that a rook can keep moving in the direction it has been heading for free

[12]Expanded, this becomes eight facts: *attack_pattern(-2,-1) attack_pattern(-2,1) attack_pattern(2,-1) attack_pattern(2,1) attack_pattern(-1,-2) attack_pattern(-1,2) attack_pattern(1,-2) attack_pattern(1,2).*

(i.e. it costs zero turns), but changing direction indicates the start of a new move. The rule that expresses this idea has two cases:

```
transition(DX,DY, DX,DY, 0) :- direction(DX,DY).

transition(DX1,DY1, DX2,DY2, 1) :-
  direction(DX1,DY1),
  direction(DX2,DY2),
  |DX1-DX2| + |DY1-DY2| > 0.
```

In the above code snippet, there is a symmetry between the treatment of the `X` and `Y` axes that makes for somewhat repetitive code. If we had been developing a three-dimensional variant of chess mazes, we likely would have altered the dimension predicate to bundle all of its arguments together in a tuple: `direction((X,Y,Z))`. If this were the case, it would be possible to write the first rule above more concisely (and independent of problem dimension): "`transition(D,D,0) :- direction(D).`" I have found that compact encodings of n-dimensional problems are often possible, but their compactness comes with increased conceptual complexity for the programmer that is not often justifiable for the $n = 2$ case.

From the player's perspective, the key design choices for a chess maze are where they start, where they must finish, and where the knights (forming their obstacles) are placed. If we know we always want puzzles in the style of Figure 10.9, we can simply declare the start and finish squares are always in the same place:

```
start( 1+w/2, 1 ).
finish( 1+w/2, w ).
```

To place exactly k knights, we use a choice rule to declare that between k and k of the squares have the knight-ness property:

```
k { knight(KX,KY) : square(KX,KY) } k.
```

At this point, we have a functioning generator for syntactically valid chess mazes. The lines above define a design space that includes every chess maze (of the right shape and number of knights) that we can imagine. Unfortunately, it also includes mazes that are too easy, too hard, impossible, and downright silly (such as mazes where a knight has been placed on the starting square). To sculpt this design space model into one that takes our solvability and target solution length requirements into account, we need to carve away those solutions with undesirable properties. In the end, this will come down to using some integrity constraints, but we need to define a model of the properties we want to constrain first.

Supposing that the answer set solver has guessed the position for each of the knights (by selecting the desired number of squares to have the knight-ness property), we can deduce the set of squares that can be attacked by those knights. This is where our `attack_pattern` predicate from above comes into play:

```
attacked(X,Y) :-
  knight(KX,KY),
  attack_pattern(DX,DY),
  X = KX + DX,
  Y = KY + DY,
  square(X,Y).
```

In the snippet above, the reference to `square(X,Y)` is actually optional for this problem. Without this, if there were a knight at the top-left corner (`knight(1,1)`), this rule would happily deduce that some square outside of the board was under attack (such as `attacked(0,-1)`). While this is not technically untrue, it can be distracting to see these nonsense facts when we are looking at answer sets that emerge from the solver while we are performing incremental development like this. Adding optional requirements like this can make it easier to validate that the program we write matches our expectations for what we are trying to model.

From the problem definition, the rook cannot pass through squares that are attacked or covered by the knights. The following pair of rules captures the set of squares that are blocked by these two conditions:

```
blocked(X,Y) :- knight(X,Y).
blocked(X,Y) :- attacked(X,Y).
```

Now comes (by far) the most complex part of this design space model. In order to deduce whether the puzzle is solvable and if it is solvable in too few moves, we could use a way to talk about how many moves it takes to get to each square. In the original evolutionary generator described by Ashlock, this process was carried out with a dynamic programming algorithm. In AnsProlog, it is enough to setup the recurrence relation that would have been expanded into the dynamic programming algorithm, and we can leave evaluating the recurrence to the solver's internals.

The predicate `at(Time, X,Y, DeltaX,DeltaY)` will indicate that the player can get the rook to move to square `X`,`Y` with as few as `Time` moves. `DeltaX` and `DeltaY` are needed to keep track of which way the rook is headed with its last move so we can account for long moves in a straight line. Assuming the solver can compute this predicate for some assignment of rook locations, the information we seek about the solvability of the level can be had by examining this predicate with the `X`/`Y` of the finish location.

The recurrence for the `at` predicate has a base case. We will set the base case so that the rook starts at the starting square. The `DeltaX`/`DeltaY` values are set as if the rook had just moved up so that any valid subsequent movement forces `Time` to increment (recall that we have pinned the starting square at the top of the board).

```
at(0, X,Y, 0,-1) :- start(X,Y).
```

With the help of the `transition` predicate from much earlier, the relation can be expressed with a single recursive definition:

```
at(T+DT, X2,Y2, DX2,DY2) :-
 at(T, X1,Y1, DX1, DY1),
 transition(DX1,DY1, DX2,DY2, DT),
 X2 = X1 + DX2,
 Y2 = Y1 + DY2,
 square(X2,Y2),
 not blocked(X2,Y2),
 T < d.
```

Take a moment to reason through this snippet. It says that if we are `at` some square `X1`/`Y1`, heading in some direction `DX1`/`DY1`, we can `transition` to a new direction at some cost `DT` (zero for keeping the same direction and one for changing it). If we take a step in the new direction, we land at `X2`/`Y2` (but only if those coordinates name a valid square on the board). This move should not be possible if the destination square is `blocked`. If the move was valid, we know we can now be `at` the new square `X2`/`Y2` with as few as `T+DT` moves (having just moved in the `DX2`/`DY2` direction).

It seems wrong that we must engineer this procedural-seeming flood-fill search in a declarative setting. We want to formulate the problem as something like "guess a sequence of moves for the player, and forbid that maze if there is a way to solve the level in too few moves (as evidenced by the guessed move sequence)." Unfortunately, getting the quantification right (that there exists a maze such that there does not exist a too-short solution) involves formulating the problem at a higher level of complexity than handled by most answer set solvers. *Disjunctive* answer set solvers can handle this kind of problem, but formulating the concerns above in their input language (exploiting a very odd sense of logical disjunction) is currently a bigger hassle than producing the procedural encoding above. Until this surface-language issue is resolved,[13] staying within traditional answer set programming (formulating problems in NP) is the practice to follow. The good news is that most constraints on the form of artifacts are expressible without straying into this territory. The strategy above is essentially a manual reduction of the concerns of function to the concerns of form.

Back in the snippet above, the final "`T < d`" check at the end of this rule seems like it might be optional in the same way that the square check was optional in the attacked predicate. In this case, including this semantically optional check has a very solid engineering motivation: it allows the grounder to terminate.[14] Without the check, the rule above functions like a convoluted version of this definition of the set of natural

[13] I have built some promising prototypes on top of the *metasp* project (`http://www.cs.uni-potsdam.de/wv/metasp/`) that make that make me very optimistic for the future of expressing problems with this kind of elevated complexity.

[14] In all ASP systems, the solver is guaranteed to terminate in finite time. Whether the grounder is guaranteed to terminate depends on which language extensions the tool supports.

numbers:

```
%% commented out for your safety: the naturals
% natural(1).
% natural(N+1) :- natural(N).
```

Approaching the end, now that we have fully defined the *at* predicate, we can use it to deduce solvability of the maze. One last deductive rule suffices, defining for which move counts (less or equal to the desired count) the puzzle can be finished:

```
finished(T) :- finish(X,Y), at(T, X,Y, DX,DY).
```

From here, one integrity constraint ensures that any chess maze in this design space can be solved in exactly d moves:

```
:- not finished(d).
```

Just one more ensures that none of these mazes can be solved in any *fewer* than d moves:

```
:- finished(T), T < d.
```

Voila! We now have a fully functional design space model for chess mazes with the size, shape, and properties of the maze shown in Figure 10.9.

To put this design space model to work in synthesizing appropriate chess puzzles, here is the shell command I used to generate a $w = 11$, $k = 5$, $d = 23$ maze in a few seconds[15] of wall-clock time using an eight-way parallelized answer set solver from the Potassco project [78]:

```
$ clingo chess.lp -l | clasp --configuration=crafted -t 8
```

In practice, one rarely gets a program right on the first try. One way to gain assurance that the design space model matches expectations is to force the solver's hand and examine the result. If the five facts below are appended to the program above, we force the solver to place the knights in the exact configuration shown in Figure 10.9. If we were overzealous with our constraints, this known valid instance might be blocked.

```
knight(4,1).
knight(8,3).
knight(5,6).
knight(10,8).
knight(3,9).
```

[15] Because parallel answer set solving is inherently non-deterministic (it depends on how the operating system schedules the solver's threads), running times will vary. In this case, I report a crude average of running times for a Mac laptop with a 2.2 GHz Intel Core i7 processor (a quad-core device with two-way hyperthreading). The flag "–configuration=crafted" was suggested by the solver's documentation to set appropriate default settings for use on "crafted" problem instances, resulting in puzzles in about half the time required by the "–configuration=frumpy" default.

To get a sense of whether we have *under*constrained our design space, one strategy is to add only a nearly-complete definition of a reference artifact (adding, say, only four of the five lines from above). Seeing how the solver squirms under pressure (getting it to enumerate alternative variations on a familiar artifact) can highlight missing constraints without needing to make sense of entirely new artifacts. This same trick, of forcing a partial set of known-good design choices, is also useful in getting quick feedback on a program that takes a long time to solve without specific constraints. Constraints that force critical choices like this do a large portions of the solver's work up front, leaving a much smaller space left to explore.

Discussion

Comparing this sketch with Ashlock's original system, the biggest difference is that Ashlock formulates the problem as an optimization process whereas I formulate it as a system of hard constraints. In the genetic algorithm version, meta-heuristic optimization tries to guide a fixed size population of candidate mazes in the direction of a target d value. In the ASP version, I ask that the conditions involving d be satisfied exactly. In reality, the difference between a $d = 22$ maze and a $d = 23$ maze is miniscule (and there are likely more pressing concerns from the aesthetics of the puzzle that we should be considering instead), so writing hard constraints on d is an oversimplification. The sketch above could be adapted, using a `\#maximize` directive, to function more like the original system. This would allow us to watch the solver, as it finds candidate puzzles, creep towards a target value and halt the search process early if we spot a candidate with a "good enough" score.

Another key difference between the systems is that the sketch above has made hard commitments that the player controls a rook and that the designer places only knights as obstacles. In Ashlock's system, the puzzles could mix and match the different piece types from Chess. Adapting the sketch above to support different pieces involves altering almost every rule, though much of the overall structure is retained. Additional background rules are required to describe the attack patterns of the alternative piece types and the movement pattern of the player piece, additional choice rules are needed to select the piece type associated with each obstacle and the player, and a more complex formulation of `attacked` is required that combines this information. From here, the `direction` and `at` predicates need to be similarly re-formulated to account for the type of piece the player controls.

The final (and perhaps most salient) point of major contrast between the systems is that the above sketch is backed by a complete search algo-

rithm. This implies that when we ask "what is the smallest[16] k (number of knights) for which we can require $d = 23$ solutions on a $w = 11$ board?" we can be sure that the answer tells us something about the problem, not about the algorithm used to examine it. Probing such upper and lower bounds (possibly in conjunction with certain parts of an artifact held fixed) is not part of appositional reasoning (it hopes to produce limits, not artifacts), but it is part the reflection process: examining the implications of the problem-as-formulated.

10.5.2 Strategy Game Maps

The next sketch is inspired by the real-time strategy game map generators in "Multiobjective Exploration of the StarCraft Map Space" by Julian Togelius and others [221] and "Limitations of Choice-Based Interactive Evolution for Game Level Design" by Antonios Liapis and others [125]. This example models the entangled concerns when multiple types of design choices define an artifact.

Overview

RTS map generators, such as the systems cited above, often work by placing critical buildings onto a height-field representing the game world's terrain. Because features of the terrain (such as the presence of sharp cliffs or smooth ramps between different height levels and impassible water zones) control how combat units navigate around a map, terrain design and building placement are entangled design concerns. Although the complex RTS games played by the public often involve continuous domains for the heights in the height-field (and sometimes for building placement as well), it is convenient to do coarse-level design for these maps using a grid representation and then produce the target smoothness and fine visual detail with a post-processing stage (as done in the DIORAMA system described in the previous section).

Inspired by the systems cited above, the sketch below places two kinds of buildings: player starting locations (called bases) and resource points. This placement happens at the level of a grid representation that also encodes a coarse representation of the map's terrain. In place of a more detailed height-field, grid cells may either be passable or impassible.

In Togelius' system (which formulated map design as optimization), key features of a map were captured with metrics like base distance (the

[16] In this case, proving that $k = 5$ is the minimum boils down to removing the lower bound on the only choice rule and adding a `#minimize` statement over the set of knight-filled squares. If there had been a simpler maze, per my artifacts-as-communication, I would have been very interested to see it.

shortest-path length on passable cells between player bases), base space (the passable area around a base available for later building construction), distance to the nearest resource (for each base), and resource fairness (a measure of imbalance associated with distances between bases and resources). In Liapis' system (another optimizer), metrics similar to those above were augmented with overall compositional metrics (unrelated to strategic fairness) such as the percentage of passable cells and the horizontal and vertical bias of their concentration. In the sketch below, we will develop a design space model that accounts for similarly structured concerns, but we will not be concerned with exactly recreating the metrics used in these existing systems.

Before jumping into the development of this design space model, see Figure 10.10 for a preview of the kind of maps that we will be capturing here.

Development

To begin, we will set up a two-dimensional grid similar to the one used in the puzzle generator above. The `adj(A,B)` predicate will encode the relationship between adjacent cells (organized as anonymous 2-tuples, ordered pairs, of `X`/`Y` values) on the map.

```
#const grid_width = 8.
dim(1..grid_width).
cell((X,Y)) :- dim(X;Y).
adj((X,Y),(NX,NY)) :- dim(X;Y;NX;NY), |X-NX| + |Y-NY| == 1.
```

Guessing which cells of the grid should be passible could be as simple as the one choice rule "`{ solid(P):pos(P)}`." This would say that between zero and all positions could have the solid-ness property (encoding ground unit traversability, perhaps liquids are impassible). However, there are various aesthetic constraints that we would like to enforce. In particular, we would like a large portion of the map to be solid, but we would be disappointed to see that constraint achieved by lumping all of the solid cells on any one side of the map. The system of overlapping choice rules below encodes the idea that about a bounded fraction of the cells on each half of the map (top/bottom/left/right) should be solid.

```
#const n = grid_width * grid_width.
2*n/5 { solid((X,Y)) : cell((X,Y)) : X>grid_width/2 } 3*n/5.
2*n/5 { solid((X,Y)) : cell((X,Y)) : X<=grid_width/2 } 3*n/5.
2*n/5 { solid((X,Y)) : cell((X,Y)) : Y>grid_width/2 } 3*n/5.
2*n/5 { solid((X,Y)) : cell((X,Y)) : Y<=grid_width/2 } 3*n/5.
```

Having chosen which cells are solid, it is easy to determine the traversability between cells:

```
traversable(P1,P2) :- adj(P1,P2), solid(P1), solid(P2).
```

Moving next to resource placement, the following snippet assigns each resource to unique cells. This is done by ensuring each resource is assigned to exactly one cell and ensuring each cell is assigned to at most one resource point.

```
#const resources = 7.
res(1..resources).
1 { resource_at(R,C) : cell(C) } 1 :- res(R).
0 { resource_at(R,C):res(R) } 1 :- cell(C).
```

Of those correspondences established by the mapping above, we should forbid those that place resources on non-solid cells. Further, as a matter of taste, we will forbid the placement of resources on adjacent cells.[17]

```
resource_covers(C) :- resource_at(R,C).
:- resource_covers(C), not solid(C).
:- adj(C1,C2), C1 < C2, resource_covers(C1), resource_covers(C2).
```

Placing player bases works almost exactly like placing resources, although we will additionally check that bases and resources do not overlap.

```
#const players = 4.
base(1..players).
1 { base_at(B,C) : cell(C) } 1 :- base(B).
{ base_at(B,C):base(B) } 1 :- cell(C).
base_covers(C) :- base_at(B,C).
:- base_covers(C), not solid(C).
:- base_covers(C), resource_covers(C).
```

At this point, we have a minimally functional map generator. It can mark cells as solid or not in an approximately balanced way, and it will place the required number of bases and resource points without overlapping. However, it may decide to place all of the resources near one player's base or create disconnected regions that isolate one player from the others. We need some control over the global relationship between bases and resources to establish a basic level of fairness in these maps.

To determine how the terrain features control the reachability of certain cells by each player, we will use a similar flood-fill formulation to the one used in the chess maze generator. Compared to the previous version, we are now treating `X`/`Y` pairs with a single variable, but we need an extra parameter to keep track of which player base `B` we are using as a reference point for measuring distances.

```
base_reaches(B,C,0) :- base_at(B,C).
base_reaches(B,C2,T+1) :-
  base_reaches(B,C1,T),
  traversable(C1,C2),
  T < grid_width.
```

[17]The less-than check is not strictly necessary in this integrity constraint, but it helps the grounder avoid grounding two versions of a semantically equivalent constraint between two cells. The less-than operator is defined over all logical terms, so it implies some ordering (unimportant for us) the ordered pairs we use to identify cells.

This rule measures how far each resource point is from each player base:

```
distance_to_resource(B,R,T) :-
  base_reaches(B,C,T),
  resource_at(R,C).
```

Using this information, we will enforce a relatively primitive model of fairness by requiring that each player can reach the same number of resources in the same number of moves (this sketch ignores resource contention in the interest of brevity). To do this, we need to know how many resource points a particular player can reach by traversing a particular distance. The following rule performs this counting operation over a fixed search radius:

```
resources_at_distance(B,T,N) :-
  base(B),
  T = 1..grid_width,
  N = #count { distance_to_resource(B,R,T):res(R) }.
```

A map is unfair if there is some distance for which different players can reach different numbers of resource points. Stated differently, in a fair map, there should be a unique number of resources available at each distance. The following snippet ensures this uniqueness:

```
resource_count(T,N) :- resources_at_distance(B,T,N).
:- T =1..grid_width, 2 { resource_count(T,N) }.
```

Now we have a model of the map design space that admits only fair maps. Unfortunately, this encoding fails to capture one major concern. It allows for the generation of maps where each player, stocked with equal resources, is isolated on their own island—the map is technically fair (according to the above definition), but it has no potential for interesting conflict between players.

To tie up this last concern, we should first define the space of cells that a given player can touch (with fixed exploration radius). Two players can reach one another if there is some cell they can both touch, but they are too close together if one can directly touch the other (in that radius). A final snippet enforces this design policy:

```
base_pair(B1,B2) :- base(B1;B2), B1 < B2.

base_touches(B,C) :- base_reaches(B,C,T). % any T is good

bases_touch(B1,B2) :- % they both touch cell C
       base_pair(B1,B2),
       base_touches(B1,C)
       base_touches(B2,C).

bases_overlap(B1,B2) :- % B2 is placed on a cell B1 can touch
       base_pair(B1,B2),
       base_touches(B1,C),
       base_at(B2,C).
```

```
:- base_pair(B1,B2), not bases_touch(B1,B2).
:- base_pair(B1,B2), bases_overlap(B1,B2).
```

Running the above program through the same parallel solver setup used in the chess maze example yields a description of the map shown in Figure 10.10 after about two seconds.

In this particular example, the solver has decided to ensure that every player has one resource point that is one step away from their base, two within five steps, and three within seven steps. It is tempting to examine the map and think "it looks like the player who owns 'B2' has an advantage here because his two closest resource points are uncontested and there is a single bottleneck protecting his territory." However, without trying this map in any particular game, let alone specifying even a sketch of the game mechanics, this is idle speculation. It may be the case that we should complain "the player owning 'B1' is over-privileged in this map because her air units have better access to harass enemies at the resource points than any other player." Map design, particularly for strategy games, can be a very subtle art. In time, exploratory design practice can dig out the deep properties of interest in this domain.

Discussion

Similar to the setup in the chess maze example, I have captured some of the key ideas from an optimization-based system as hard constraints on appropriate artifacts in the sketch. Sometimes incremental optimization is used as a way to exploratorily discover artifacts that satisfice (we do not what threshold on a metric value defines "good enough," but we can watch artifacts improve until we see one with which we are satisfied). In other applications, particularly Togelius' system operating in the domain of playable StarCraft maps, the intent of the multi-objective optimization was to map out the space of possible tradeoffs between different metrics we might want to apply to the maps. For example, an easy way to achieve good resource fairness score in that system is to place all of the player bases close together, but this diminishes the base distance metric. Maximizing one metric often comes at the cost of reducing others.

If there were only one metric of interest (perhaps a weighted sum of scores from simpler metrics), then we could readily use a `#maximize` statement to globally optimize the overall metric. However, this would likely yield an unsatisfying compromise that allows some bases to be close, some resources to be reached unfairly, and some skew to the distribution of reachable squares. Looking to probe the limits of our design space (as we did in the previous example by minimizing the number of knights that present a given level of modeled difficulty) entails searching for artifacts on the Pareto frontier: artifacts for which none of the metric scores can be

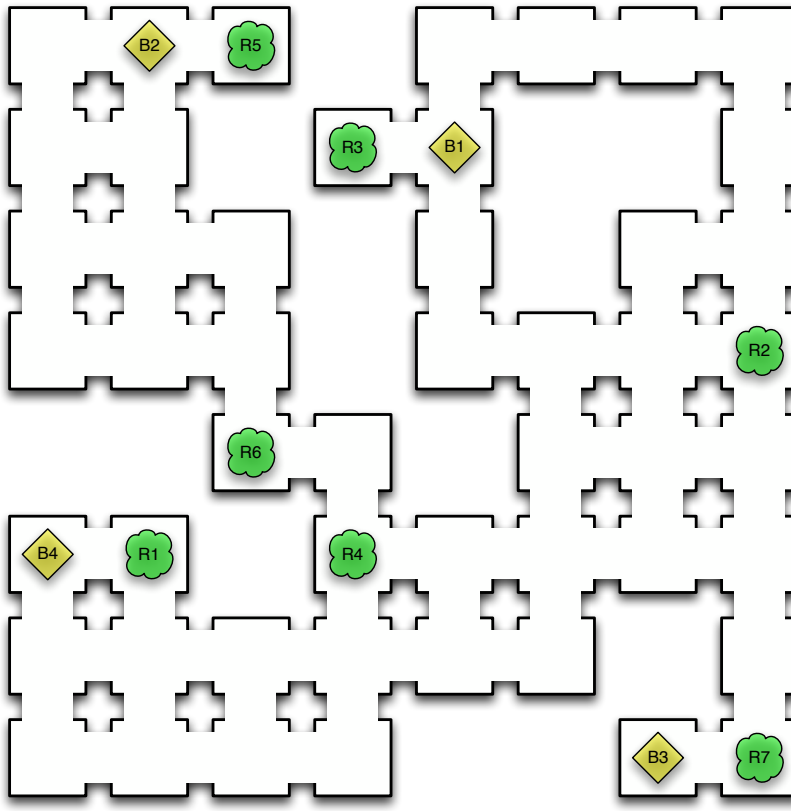

Figure 10.10: Generated strategy game map for no game in particular. Yellow diamonds represent the starting base locations for four players. Green clouds represent seven resource points that players might capture. The notion of *fairness* at work in this map is that every player has the same number of resource points accessible within the same number of steps on the grid (up to some maximum radius). Here, players can reach one point in one step, two points in five steps, and three points in seven steps. Other concerns such as the distribution of passable grid cells and the distance between bases have also shaped this map's form.

improved upon without reducing another. Finding Pareto efficient solutions within answer set programming is a subtle topic (it involves solving an NP^{NP}-complete problem), but it is not impossible. The *metasp* library from the Potassco project provides the building blocks for modeling Pareto efficiency as one of the concerns in a design space model [77].

In Liapis' system, the optimization process starts with human-designed maps as a form of expert input. Although ASP provides no way to suggest that search start from any given point, we can recycle a trick from the previous example to ask for new maps that are variations on a given map. To explore variations on a hand-crafted terrain map (ignoring bases and resources for now), we should collect up a set of facts like `crafted_solid(C)` that note whether a cell was solid in the crafted map and append these facts to the design space model. The snippet below will ensure that the resulting design space only includes artifacts for which the terrain differs in a few places (say, 5) from the crafted input.

```
#const tolerance = 5.
disagreement(C) :- cell(C), 1 { solid(C), crafted_solid(C) } 1.
:- not 0 { disagreement(C) : cell(C) } tolerance.
```

Another feature of Liapis' system is fitting a model of user preferences to feedback given on machine-generated maps. As this adds a machine learning component to the system, I have left this feature out of the sketch above. The natural equivalent for this sketch would be to learn symbolic constraints from user input. This is exactly a task for inductive logic programming (ILP), which can be implemented using ASP [43]; however, sketching such a learning system here would distract from the much more direct way of involving the user in the generation process examined in the next example.

10.5.3 Platformer Levels with Support for Mixed-Initiative Interaction

My final sketch is inspired by "Tanagra: Reactive Planning and Constraint Solving for Mixed-Initiative Design" by Gillian Smith and others [202]. This example couples a design space model to the rich collection of external constraints that might come from a user in interaction with a graphical design tool. It also models a design space using a numerical constraint answer set solver.[18] Even though constraint answer set solving is an emerging topic, it is interesting to explore building something with technology that, in time, may become more stable and mainstream.

[18]Specifically, this example uses language features introduced in Clingcon v2.0.2 (`http://sourceforge.net/projects/potassco/files/clingcon/2.0.2/`).

Overview

Recall from § 9.4.2 that Tanagra is a prototype level-design tool for platformer games. Tanagra explores the potential for mixed-initiative interaction between a human designer (who knows the high-level requirements for a design project and can make aesthetic judgments without the need for formal justification), and a machine designer (that knows numerical constraints rising from a game's mechanics and can be trusted with the tedious search process involved in ensuring that a level is playable). Although our sketch will not be integrated with a graphical interface, it is designed to function as exactly the kind of internal system that would support this mode of interaction.

In Tanagra's model of platformer levels, a level be can broken down into discrete *beats* during which the player performs a single action (such as killing an enemy or waiting for a stomper to clear from the path). The system's beat-timeline reads like a high-level script for how to complete the level: jump up, jump up, kill enemy, wait, jump down, and so on. In the game world, beat activities are realized by placing *platforms* with numerically parameterized geometry (specifying their left/right endpoints and their height). To ensure that a level is valid and playable, a constraint solver inside of Tanagra verifies that all of the platform parameters are internally consistent (e.g. their right sides are indeed to the right of their left sides) and that the width and height of gaps between platforms are within stated jump feasibility tolerances. Our sketch will support the same local validity and playability constraints that Tanagra used, and it will use the same library of basic activities: jumping, springing, killing, and waiting.

Beyond simply placing platforms to ensure playability, Tanagra makes use of multi-beat patterns inspired by level design motifs in popular platformer games to enforce higher-level structure on the generated levels. In the sketch below, we will only account for one example of Tanagra's multi-beat patterns: the valley (described later).

With no specific constraints coming from the user, Tanagra functions like a traditional level generation system, producing a new level from scratch each time it is invoked. Our sketch below will support this *tabula rasa* mode of generation as well. However, the intent of Tanagra is to allow the user (assumed to be a novice) to get help on her own design project, not to accept the machine's output as it first appears. In place of allowing the user to graphically paint constraints on where platforms should be placed, our sketch will require its users to manually type in their intent as logical facts at the end of the program. This process would be automated in a graphical system.

When the user of Tanagra paints new platforms that break the work-

ing level's playability, Tanagra searches for a local adjustment of platform geometry that will restore playability. If no suitable local adjustment is possible, Tanagra widens its search to nearby platforms (potentially breaking multi-beat patterns), examining more and more drastic edits to restore playability. If the search process cannot quickly find a repair without undoing any of the designer's work, it gives up. In the sketch below, we will not be so gentle. The resulting system will restore playability of the level at all costs (or, rather, at minimum cost). In consolation, it will attempt to leave as many features of the previous level design unchanged as possible.

Before we launch into the development of this sketch, the reader should recall that the focus of this sketch is to exercise mapping a design space onto external constraints that are not known at development-time. Making a design space model controllable in this way goes beyond providing tweakable numerical parameters; it involves exposing a language of constraints that others can use without requiring them to first understand the rest of the design space model's encoding. As before, look ahead at Figure 10.11 for a visual clue of where this sketch is headed.

Development

As with the other examples, we will start by defining some numerical constants. These particular numerical parameters would only be adjusted by someone wrapping a graphical interface around this system, not by the end users of the system, so we can imagine that they are really constants.

```
#const num_beats = 10.
#const min_platform_width = 4.
#const max_platform_width = 16.
#const jump_height = 4.
#const jump_width = 4.
#const spring_height = 8.
#const spring_width = 8.
```

Clingcon's language requires us to specify a domain of values for the integer variables that it will manage.[19] While we could set the limits of this domain to some very large value without much impact on solver performance in this problem, we will set it to a reasonable number motivated by Tanagra's user interface. In Tanagra, platforms are painted with discrete blocks (and a typical gap size between platforms might span two block units). In our sketch, we will model a canvas that is 100 units wide:

```
#const units = 100.
```

[19] Any syntax in this example involving the $ sigil is specific to this experimental answer set solver, not a standard AnsProlog language feature. See `http://sourceforge.net/projects/potassco/files/clingcon/2.0.2/clingcon_language.pdf` for a brief review of the language features available in the version of Clingcon used in this example (v2.0.2).

```
$domain(0..units-1).
```

Before we place platforms, we should establish the vocabulary of activities a player might perform in each beat of the level.

```
action(jump).
action(spring).
action(kill).
action(wait).
```

Jumping and springing are treated specially because some multi-beat patterns depend on the direction of the activity (e.g. springing up vs. jumping down).

```
directional(jump;spring).
dir(up;down).
```

The following snippet will assign every beat a unique action, and it will optionally tag directional actions with a direction:

```
beat(1..num_beats).
1 { activity(B,A):action(A) } 1 :- beat(B).
0 { direction(B,D):dir(D) } 1 :- activity(B,A), directional(A).
```

With just the above rules, our sketch captures a space of beat timelines, but we will need to map beat activities to platform geometry before we can examine any concrete levels. Unlike many constraint programming languages, Clingcon does not require us to declare constraint variables—we just start using them.[20] In the snippet below, "`left(B)`" (or "`left(2)`" when grounded for the second beat) is not a reference to a predicate; it is the identifier for a constraint variable (e.g. "`left(2)`" is the integer constraint variable that holds the horizontal position of the platform associated with beat #2). This snippet enforces a minimum and maximum width on platforms as a relation between the left/right variables associated with each beat:

```
right(B) $- left(B) $>= min_platform_width :- beat(B).
right(B) $- left(B) $<= max_platform_width :- beat(B).
```

Similarly, this snippet ensures that beat geometry is ordered from left to right:

```
left(B) $< right(B) :- beat(B).
right(B) $<= left(B+1) :- beat(B).
```

To ensure that the generated level geometry spans the whole 100-unit canvas, we can pin the edges of the first and last beat to the edges of the canvas:

```
left(1) $== 0.
right(num_beats) $== units-1.
```

[20] Clingcon can get away with this because it supports only integer constraint variables. Richer constraint programming systems have a wider vocabulary of constraint variable types, making variable declaration useful to avoid ambiguity.

If we want our canvas to look like the short and wide canvas used in Tanagra, we need to put an upper limit on the height parameter associated with each platform:[21]

```
height(B) $<= 20 :- beat(B).
```

More importantly, the heights for each platform should agree with the direction specified in the beat timeline:

```
height(B) $< height(B+1) :- direction(B,up).
height(B) $> height(B+1) :- direction(B,down).
```

The height difference associated with directional jumps should not be too extreme—the player needs to be able to make these jumps.

```
$abs( height(B+1) $- height(B) ) $<= jump_height :- activity(B,jump).
$abs( height(B+1) $- height(B) ) $<= spring_height :- activity(B,spring).
```

Similarly, the width of a jump (the distance between the right side of one beat's platform and the left side of the next beat's platform) should not be too extreme either.

```
left(B+1) $- right(B) $<= jump_width :- activity(B,jump).
left(B+1) $- right(B) $<= spring_width :- activity(B,spring).
```

To encourage the player to jump, we will ensure there is a minimal width to the gap they must cover.

```
left(B+1) $- right(B) $>= 1 :- activity(B,jump).
left(B+1) $- right(B) $>= jump_width :- activity(B,spring).
```

The above numerical constraints only covered jumping and springing activities. To make sure no spurious gaps show up in beats tagged with the kill and wait activities, we should add the constraint that platform geometry for successive beats be contiguous in these cases.

```
adjoined_action(kill;wait).
height(B) $== height(B+1) :- activity(B,A), adjoined_action(A).
right(B) $== left(B+1) :- activity(B,A), adjoined_action(A).
```

We now have a functional platformer level generator—or at least we have a generator for playable yet often uninteresting levels. As a gesture in the direction of capturing interesting high-level structures, the next snippet shows how detect instances of a multi-beat pattern called a *valley*. In a valley, the player jumps down to kill and enemy and then immediately jumps up to a platform of the original height before continuing.

```
detected(valley,B) :-
  direction(B,down),
  activity(B+1,kill),
```

[21] What's wrong with platforms with height set to 100? Why isn't this set in a `#const` directive? When I was viewing levels generated from this design space as generated ASCII-art in my terminal, a maximum height of 20 allowed me to comfortably examine two levels and their associated metadata without scrolling. In this example, the canvas is an abstraction that would be altered to fit the graphical constraints of some other tool as needed. Exploratory design can be a little messy.

```
  direction(B+2,up),
  height(B) $== height(B+3).
```

Unfortunately, we have no *a priori* reason to want a valley to be detected in our levels. Whether there should be a valley and where it should be situated is a matter of the user's taste. In the next snippet we forbid levels missing valleys, but only if the user (who we can imagine is able to append only ground facts) cares to ask us to enforce this policy.

```
required_detection(valley,3).
:- required_detection(Pattern,B), not detected(Pattern,B).
```

As encoded above, the presence of a valley at beat #3 is a hard constraint. As much as the generator will be willing to tear up the rest of the current level design to ensure playability, it will never consider failing to form the required valley. Although this seems like an overly strong requirement for a relatively minor pattern, I have included it in this sketch primarily for contrast with the soft constraints that we will model next.

Imagining that our user has painted some geometry of their own design on one side of the level, how should we capture the intent of the as-local-as-possible playability repairs that Tanagra produced? First, we will represent what the user has done so far:

```
previous_activity(1,jump).
previous_direction(1,up).

previous_geometry(left(2),12).
previous_geometry(height(1),8).
previous_geometry(height(2),3).
```

They seem to have given us conflicting design goals. It looks like they want beat #1 to involve jumping up,[22] but they are also asking for the platform in beat #2 to be lower. Knowing that when given no hard constraints beyond that of playability, the solver will only emit playable levels, we can start by simply detecting when the solver makes a choice that disagrees with the user's previous inputs.

```
activity_tweak(B) :- previous_activity(B,A), not activity(B,A).
direction_tweak(B) :- previous_direction(B,D), not direction(B,D).
geometry_tweak(G) :- previous_geometry(G,V), G $!= V.
```

At least now the solver can notice when it disregards the user's intent. To get the solver to prefer local repairs, we should have it determine a sense of the size of the repair it is considering and have it (globally) optimize that metric. We will give priority[23] to solutions that preserve (or do

[22]Would users really be manually specifying the beat activities like this? It is possible, but I imagine automatically detecting the activities implicit in the geometry that they paint. In this way, the solver can still attempt to maintain the intent of what they painted even if it needs to alter the geometric properties of every platform nearby.

[23]This prioritization is done through the mechanism of lexicographic optimization, a form of multi-criteria optimization that is related to but distinct from Pareto optimization [169].

not tweak) activities and their directions over those that preserve precise geometric parameter values. In the resulting levels, there is no alternative repair that tweaks fewer geometric parameters while still preserving the same number of structural features, and there is no repair that tweaks fewer structural features.

The encoding is simple enough:

```
#minimize { geometry_tweak(B) @ 1 }. % lower priority
#minimize { activity_tweak(B) @ 2, direction_tweak(B) @ 2 }.
```

A similar prioritization strategy could be used to implement several levels of soft constraints (perhaps even allowing the playability constraint to be relaxed if needed in the interest of getting some visible result on which to attempt manual repairs).

Defined in this way, our design space model is wrapped around our hypothetical user's current working design (encoded as logical facts that can be generated from his graphical input). Whether an artifact has appropriate form (or perhaps how appropriate its form is) depends on how well it resembles an under-construction reference artifact with which the user is already very familiar. When a particular design project requires artifacts to have subtle and difficult to formalize properties, we need not immediately dive into trying to encode those properties in AnsProlog. We can instead model the design spaces of artifacts that look like minimal edits to some artifact that is perhaps aesthetically appropriate but missing a key structural property that would be tedious to reliably ensure.

Combining all of the above snippets into a single program and running them with the following command[24] results in the level design visualized in Figure 10.11 in less than half a second on a much older machine than the laptop used in the two previous examples.[25]

```
$ clingcon dathon.lp --rand-freq=1
```

Discussion

Although not apparent from my description of Tanagra so far, one of the key differences between Tanagra and the sketch above is one of software architecture (the topic of the journal article on Tanagra [202]). Tanagra's

[24]Why is the system called *dathon*? Dathon, the Tamarian space cruiser captain, was the alien who, through non-verbal means, was able to convey the core meaning of the story of "Tanagra" to the wounded Captain Jean-Luc Picard, allowing the two to cooperate, fighting off a violent creature and surviving to be rescued. Well, Picard was rescued and Dathon died—but the point is that this Dathon guy was able to communicate something deep about the "Tanagra" story. See also: `http://en.memory-alpha.org/wiki/Darmok_(episode)`

[25]Conveniently precompiled binaries for Clingcon v2.0.2 were only available for Linux.

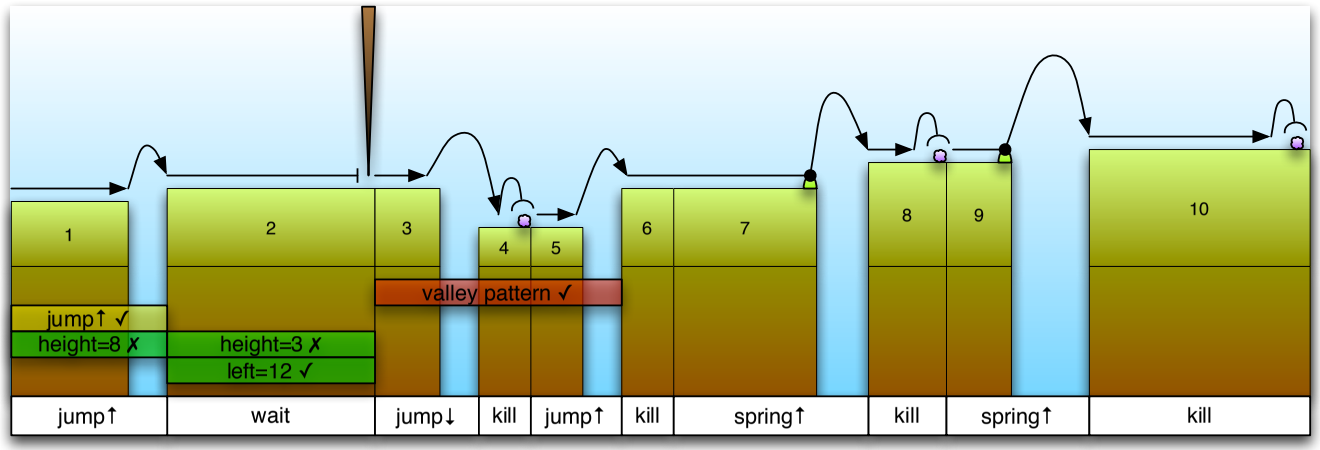

Figure 10.11: An example generated platformer level. This example comes from a design space that was mapped onto a set of externally-provided user constraints. In the diagram, red constraints are mandatory. Yellow constraints (just one above) are preferred, and green constraints are preferred with a lower priority. The user has absolutely required that the multi-beat *valley* pattern is present starting on beat #3, and she has requested that beat #1 involve a "jump up" activity. Conflicting with this request, she has also asked that the platform in beat #1 be numerically *higher* than than of beat #2. To control the width of the first beat, she has also requested that the left side of the platform in beat #2 begin at block 12. Finding a candidate level that satisfies hard constraints, and globally maximizes the number of soft activity-level requests, the answer set solver finds a solution in which the two height requirements must be overridden but the left = 12 constraint is satisfied.

search through the space of possible levels is split between two loosely coupled components: a reactive planner (written in ABL [135]) that manages that search through structural features (beat activities and multi-beat patterns like valleys) and dedicated numerical constraint solver (`choco` [30]) that manages the search for numerical parameters satisfying a very specific query posed by the planner. In the system we sketched above, the structural and numerical search processes are tightly interleaved[26] in the single invocation of the answer set solver.

In Tanagra, when the planner is searching for a consistent repair to the level, it does so in the open. We can watch it add and remove platforms from the canvas in real-time. In a hypothetical mixed-initiative design tool built around the system sketched above, the machine would play a much more cautious and contemplative role in the conversation. It would only report back an edit to the level when it was sure there was absolutely no better alternative. Even if we visualized sub-optimal results emerging as the search progressed, we would only ever see the machine imagining (more drastic than necessary) edits that fully restored playability. In between these complete solutions, examining the state of the solver's decision variables will often yield results that are difficult to visualize. It may have decided that one platform is below another one but not know where either are placed on the canvas. Similarly, it may have decided that the activity for a certain platform is *not* springing without having yet decided what alternative to pick instead. The solver's ability to manipulate partial artifacts is a both a major strength and point that makes its search process difficult for human interpretation. For a large and complicated map, this means the machine might sit uninformatively silent as vast spaces of alternatives are being tested and invalidated. Even when Tanagra's visibly frantic search cannot find a solution, observing it struggle and examining the options it is trying may inspire the human user to find a way around the impasse. Because the sketch above was designed to hide exactly this sort of detail, it is unclear how to get it back when it might be useful in communicating with the human designer.

Although the problem did not arise in this sketch because of the small number of beats considered, adding and removing user constraints to an existing answer set program can involve paying the cost of grounding the program each time (even though only a small piece has changed). The system of external predicates supported by the Potassco tools [78] mostly avoids this problem, though it is not without subtleties. The "reactive

[26] Clingcon uses its internal numerical constraint solver, GECODE [184], quite similarly to how Tanagra uses Choco: it adds and removes (called posting and retracting) numerical constraints as part of the outer search over Boolean properties such as those describing the beat activities. When the inner constraint problem becomes infeasible, it forces the outer search process to backtrack and try another alternative.

answer set solver" Oclingo [74] is designed for building interactive design spaces such as the one sketched above, however it has not yet been integrated with the numerical constraint solving system that powers Clingcon.

Chapter 11

Applied Systems Overview

The previous chapters made the case that answer set programming *should* be useful in producing design automation systems on the basis of language affordances and self-contained examples. By contrast, the following chapters examine the interaction between my design automation practice and several larger, applied projects. Each of the systems I present in the next chapters function as computational caricatures, offering a vision of mechanized exploratory game design capturing one or more varieties of game design spaces.

A slight variation on Chapter 12 was previously published as "Ludocore: A Logical Game Engine for Modeling Videogames" as joint work between myself, Mark Nelson, and Michael Mateas for the 2010 Conference on Computational Intelligence and Games [199]. The design spaces at work in LUDOCORE are primarily those of gameplay traces. That is, while a game designer seeks to craft a game, managing the space of possible play traces that the game allows is hefty task—one that could make use of some mechanized assistance.

Historically, LUDOCORE was the first of my systems that hinted at the general use of ASP for design automation purposes and, in particular, its use for level design automation. In the internal work that lead up to LUDOCORE (previous to and independent of Martin Brain and Florian Schanda's use of ASP in the *Warzone 2100* map generator) I was primarily using ASP as a slightly more convenient way to access the event calculus than the tools chosen by my colleague (Mark Nelson tended to use Mueller's *decreasoner*[1] system, e.g. in "Recombinable Game Mechan-

[1] `http://decreasoner.sourceforge.net/`

ics for Automated Design Support" [155]).

Chapter 13 is a variation of "Computational Support for Play Testing Game Sketches" published with the above authors at the 2009 Artificial Intelligence and Interactive Digital Entertainment Conference [198]. Although the BIPED system it describes predates LUDOCORE in the publication record, LUDOCORE is in fact one of BIPED's two legs. This chapter explores building out support for human playtesting of the very same models that LUDOCORE supports under machine playtesting. Although BIPED makes use of no additional design space models beyond those used in the previously described machine playtesting subsystem, it does highlight a major source of new constraints to exploratorily apply to design space models: live interaction with artifacts. Playing with a game, realized with even placeholder graphics and sound, can make the gap between what-you-modeled and what-you-thought-you-modeled immediately obvious.

Chapter 14 represents the first of my published work (previously published as "Variations Forever: Flexibly Generating Rulesets from a Sculptable Design Space of Mini-Games" [194] and presented at the same venue as LUDOCORE) that acknowledged ASP as a means of synthesizing new game elements (rules and content) instead of as a technology to assist in the analysis of hand-crafted game designs. Although I never returned to development of the game that the chapter describes, *Variations Forever*, the intent to use an ASP-based content generator in a deployed game was resurrected in the next project.

Chapter 15 is a variation on "A Case Study of Expressively Constrainable Level Design Automation Tools for a Puzzle Game" presented at the 2012 Conference on the Foundations of Digital Games with my new colleagues at the Center for Game Science at the University of Washington (Erik Andersen and Zoran Popović) [192]. In my previous application projects, requirements on generated artifacts always arose from my personal interest. In developing a set of level design automation tools for *Refraction*, I encountered rigid, external constraints that, before the project began, seemed like they might be difficult to tackle with answer set programming. Indeed, the three tools I created were intended as replacements for hand-crafted search tools created by the game's designers. Could ASP compete with the pre-existing, project-specific tools and go on to provide additional benefits? The outcome was positive, and results suggest a new practice for the development of in-house design automation tools. At the time of writing, an enhancement to the tools described in this chapter is being embedded into the ongoing development of a sequel to *Refraction*.

Chapter 12

Ludocore

My first applied system resembles a general game playing system: it produces sequences of player actions that achieve some goal. However, instead of being used for competitive play, it is intended for use by designer in understanding his or her own games. The primary design spaces to be modeled in this scenario are spaces of play where the appropriateness of a given play trace is subject to the designers interest, often involving figuring out how a particular kind of player might play.

LUDOCORE is a logical "game engine," linking game rules as reasoned about by game designers to the formal logic used by automated reasoning tools in AI. A key challenge in designing this bridge is engineering a concise, safe, and flexible representation that is compatible with the semantics required by videogames.

12.1 Overview

While the term "videogame" brings to mind graphics, sounds, and story worlds, at the core of every game is a formal rule system. I am interested in declaratively modeling these games so that the emergent properties of their rule systems can be understood. The tools of symbolic AI hold promise for bringing such properties to light, but have not been used for design or analysis of videogames, in part because of a mismatch between how designers think of their rule systems and the way logical specifications are usually written. I propose the use of a logical "game engine" to ease and accelerate the modeling of game worlds in formal logic. The engine provides a set of primitives and abstractions that link game-level concepts to the first-order logic understood by AI reasoning tools. Specifically, the engine supports automatically generating game-

play traces, integrated player modeling, and both temporal and structural queries. Finally, games modeled in LUDOCORE can be directly read in, without compilation or translation, to a custom, prototyping-focused game engine (described in the following chapter) allowing them to be played as real-time, graphical games.

An existing way to represent purely abstract games in logic is GDL, the game description language [130]. GDL was designed to specify games for the general game playing competition [82], in which computers competitively play unfamiliar, turn-based games. Thus, the language itself was designed primarily to write infrequently-edited specifications to be read and understood by computer players as state-transition systems. In contrast, when modeling videogames, designers are much more concerned with the ease with which they can incrementally build up a game's rule system through a series of experimental tweaks [73, p. 14]. LUDOCORE supports this kind of modeling, at least for technically savvy designer-programmers.

My iterative-design motivation also contrasts with the goals of formal verification. While verification attempts to prove that a set of desired properties hold, it is not clear during a game's prototyping phase which properties are even desirable. In fact, coming to understand what properties a game has, and which ones would be desirable, is the main challenge.

My bridge from games to formal logic is based on the event calculus, a logical formalism commonly associated with commonsense reasoning. In previous research [156], my colleagues proposed the event calculus as an attractive basis on which to build informative, queryable prototypes. Since then, in building games with the event calculus, we encountered various design patterns and idioms, common to all of our games; it is this experience that prompted the design of LUDOCORE.

12.2 Building Games on the Event Calculus

Standard game-design texts discuss a game's mechanics in terms of objects with dynamic properties (the state of the game), and triggered behaviors (events and conditional state updates) [1, p. 295][73, p. 112]. While state and events can be modeled with several formalisms, or even ad-hoc logical encodings, a common challenge is the well-known frame problem, commonly solved by adding frame axioms. In a language like GDL, frame axioms explicitly declare when state doesn't change from turn to turn. An alternate proposed solution is based on the *commonsense law of inertia* [187], which reads that states retain their values until an event changes them. This mirrors the usual imperative game programming assumption that variables stay set to the same value until changed by an active be-

havior.

Given my desire to model state and events, and avoid frame axioms that are tedious to maintain, the event calculus (EC) is a natural choice. Additionally, there is a large body of literature discussing the use of EC as a practical knowledge representation, e.g. Mueller's book [151] on applying it to commonsense reasoning problems involving interaction of objects over time, for which inertial state is the common case.

The discrete event calculus [150] is based on fluents (predicates whose truth values vary over time) and events, which happen at particular integer-valued timepoints and can change the truth values of fluents. Its key predicates are: `happens(E,T)`, which says that an event happens at a timepoint; `holds_at(F,T)`, which says that a fluent is true at a timepoint; and `initiates(E,F,T)` and `terminates(E,F,T)`, which map event occurrences to changes in fluent truth values. In addition to state that does not change without cause (inertial fluents), the circumscription used in the event calculus implies that events have no effects besides those that can be explicitly derived. Together, these two kinds of default reasoning give the event calculus a degree of elaboration tolerance [140], the ability to modify a knowledge representation without re-engineering it, because new assertions override defaults.

To realize EC in a computational setting, I use an encoding of the EC axioms in answer-set programming (ASP), an approach proposed in unpublished notes accompanying Mueller's book [151],[1] which Kim et al. [109] proved preserves the expected EC semantics. Answer sets are sets of literals that represent acceptable beliefs in an abstract world [126]. When applied to programs in the discrete event calculus, these answer sets amount to assertions about what was true and what happened at each timepoint. Further, for games modeled in EC, the narrative of events amounts to what I call a gameplay trace. While there are more direct ways to extract a single trace from a game, ASP does not forward-simulate a game, but rather reasons abstractly about the space of possible executions, which is far more flexible.

The combination of EC+ASP is an interesting tool for game-related AI because ASP provides fast inference to models of a game's execution, and EC is a solid knowledge representation for abstract worlds with time. Together they facilitate generating traces from concise declarative descriptions. This trace inference is in fact more expressive than forward-search based methods such as Monte Carlo rollouts; for example, it can definitively prove certain properties of a game, rather than simply showing that they are unlikely.

[1] `http://decreasoner.sourceforge.net/csr/ecas/`

12.3 The Logical Game Engine

There are some drawbacks to modeling games directly in EC+ASP. In particular, nearly all of the `T` variables are superfluous because, outside of the event calculus axioms, the logical rules of a game tend to refer only to the current timepoint. Secondly, crafting more complex games directly in the event calculus formalism leads to duplication of common preconditions for an event in each of its `initiates`/`terminates` clauses. This duplication creates a maintenance challenge for the game's author, who might want to make simple changes to the conditions for a particular event. Finally, expressing the fact that some set of game events conflict with each other, despite being independently possible, requires mapping a complex idiom (of threading special conditions through most game logic) over each new game design. Mutually conflicting events are a common occurrence even in simple turn-based games, but describing this constraint directly is, in my experience, error-prone. These annoyances are resolved in LUDOCORE while retaining the advantages of EC+ASP.

Programming computer games is an immensely difficult task, and only through the continued application of software engineering practices has a sequence of increasingly powerful game engines made possible the rich games we expect today [13]. Game engines attempt to provide standard solutions for game programming problems without prescribing particular rules or settings for a game. These solutions are exposed by a set of APIs that games can be programmed against. Some engines, such as Torque[2] and Unreal,[3] go so far as to provide custom programming languages to ease the integration of a game's specification with the engine's services. LUDOCORE is modeled after this richer variant of game engines: it provides not only APIs that encapsulate solutions to difficult logical modeling problems (such as conflicting events) but effectively provides a new language that is tailored to the application of specifying logical game worlds. Games produced with the engine are not only smaller than their equivalent specification using only the event calculus, but also easier to maintain throughout meaningful design changes, and easier for the game's author (or even automated tools) to analyze.

My game engine is essentially a background theory for logical game descriptions. In the rest of this section I will describe the logical predicates of our game engine and how individual games can leverage them. Figure 12.1 gives an overview of how our engine builds on the event calculus, and in turn supports modeling games on top of it. The event calculus axioms provide the base semantics for discussing state and events over time. The engine adds higher-level abstractions for modeling games than

[2] `http://www.torquepowered.com`
[3] `http://www.unrealtechnology.com`

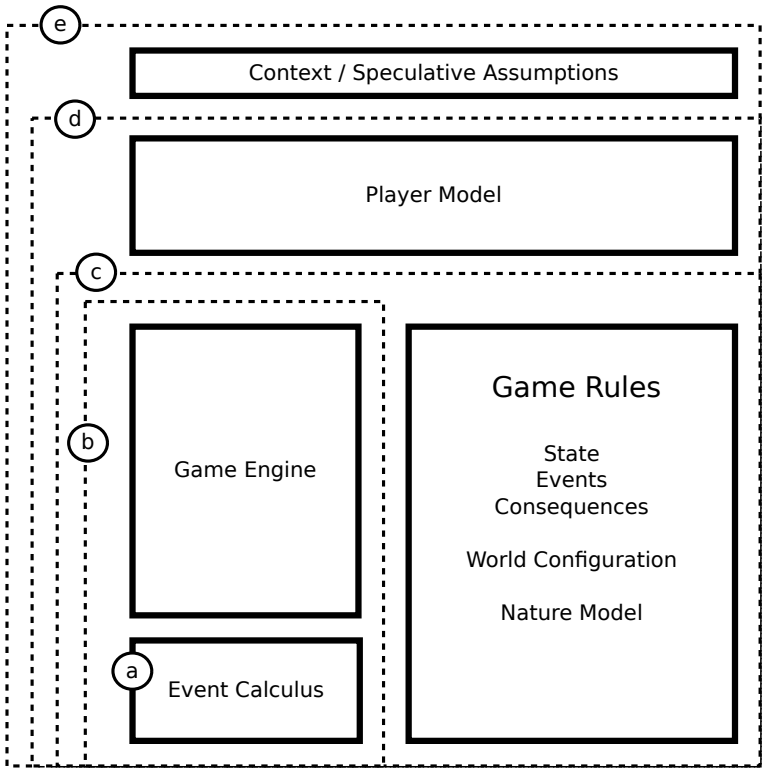

Figure 12.1: Block diagram illustrating how the logical theories involved in a complete application of LUDOCORE fit together to form a logic program: (a) provides a temporal logic basis; (b) expands the temporal logic to include videogame-level concepts; (c) is a complete specification of a particular game; (d) provides a model of a certain class of players playing this game; and (e) represents a focused view of particular situations that could arise in this type of player's play.

the raw event calculus primitives do (discussed below).

To produce a completely specified logical game that admits automatic gameplay trace generation, the author adds a particular game's rules: the specific state and events that make up the game, the consequences of events happening in the game world, the configuration of entities within the game world (e.g. level layouts), and a model of when things take place in the game world without the player's intervention (caused by "nature"). While this fully specifies a game world, a player model additionally adds assumptions about what kinds of actions a player can take, in which combinations—either due to actual restrictions (e.g. if the author has in mind an input mapping[4] that would make certain actions impossible to perform simultaneously), or due to a desire to investigate certain kinds of player behavior. Finally, speculative assumptions can be added that restrict the generated traces to certain kinds of situations that the author wants to investigate.

12.3.1 State, Events, and Consequences

Since state and events are natural elements of a game's definition, I expose the event calculus's fluents and events in the game engine with the predicates `game_state` and `game_event`, respectively. A game state assertion in LUDOCORE looks like this:[5]

```
game_state(at(A,R)) :- agent(A), room(R).
```

This assertion reads that *at* is conceptually a table that records the relation between agents and rooms (answering, "is agent A in room R?"). Elements of this table stay set until changed otherwise, a property inherited from the event calculus's commonsense law of inertia.

For state that should be updated dynamically as a function of other state, I provide the `state_helper` mechanism, which provides a computed view on inertial state. State helpers are, semantically, event-calculus fluents with inertia disabled, which cannot be directly initiated or terminated. This stratification into primary and derived fluents that I enforce syntactically is one safe idiom for avoiding the ramification problem in the event calculus [188].

The `state_helper` assertion below provides a convenient view on the `at` state for checking when a particular agent is at their starting location (`starts_in`) without having to name that location. This example also

[4]My colleagues discuss elsewhere [154] why, when modeling games, input mappings make sense to model separately from the mechanics that define a gameworld.

[5]Note that the code examples given in this chapter are from LUDOCORE's custom logic programming language, not standard AnsProlog. The translation from LUDOCORE's language (a subset of Prolog) to AnsProlog is fairly transparent, but do not expect this code to function as-is.

illustrates how rules can be conditioned on the current game state using the `holds` predicate. This state helper rule implies that `home` is a dynamic (time-varying) property of the game world:

```
state_helper(home(A)) :- holds(at(A,R)), starts_in(A,R).
```

Game-event declarations are relatively straightforward. The assertion below describes the event of an agent performing a healing action:

```
game_event(heals(A)) :- agent(A).
```

However, game events can have significantly more structure in LUDOCORE than in the event calculus alone. For example, the conditions under which a game event is possible (e.g., the legality of moves in a board game) are specified using the `possible` predicate. In this example,[6] agents may only heal in their starting room:

```
possible(heals(A)) :- home(A).
```

Possibility assertions are used not only for defining the rules of the game, but also to control event selection when generating gameplay traces. Importantly, possibility is deeply intertwined with the event conflict mechanism. An assertion[7] like the one below can keep two otherwise *possible* events from happening together:

```
conflicts(heals(A), moves(A,R)).
```

By providing `possible` and pairwise `conflicts` conditions for game events in a game model, the required event-selection logic can be implemented and debugged once in the game engine, instead of by each game author. By default, actions that are not marked to conflict are safe to co-occur (whereas basic GDL enforces exactly one player action per turn, which is unsuitable for modeling a more general class of games).

Game events can be tagged with whether they are direct player actions or spontaneous actions that can only be caused by a non-player entity, thought of as "nature" or "the game master." The `player_event` and `nature_event` predicates signify this tagging. Although these annotations do not correspond to what we normally think of as game rules, they play an important role in scoping the applicability of player-modeling rules, described with the `player_asserts` and `player_forbids` predicates (and corresponding `nature_` predicates), discussed later.

[6] After translation to AnsProlog, this rule takes the form "`possible(heals(A),T) :- holds(home(A),T).`" Note the injection of `T` variables and the translation of the state-helper into a check on a fluent.

[7] After translation, this assertion is given meaning in AnsProlog an integrity constraint: "`:- conflicts(E1,E2), happens(E1,T), happens(E2,T).`" Although this single assertion successfully blocks co-occurence of conflicting events in generated gameplay traces, additional internal complications arise when implementing the desired semantics for the player and nature models described later.

Linking game state to game events, the `initiates` and `terminates` predicates from the event calculus are exposed to game authors with minimal change. The only modification my engine uses is that these predicates are defined in a time-invariant fashion, always referring to the current game state. The example[8] below asserts that the `moves` event causes an update to the `at` game state, and that the `hits` event causes the target of the hit to no longer be `alive`, under certain conditions:

```
initiates(moves(A,R), at(A,R)).
terminates(hits(A1,A2), alive(A2)) :-
       holds(armed(A1)),
       not holds(armed(A2)).
```

Finally, the initial state of the game world is given by the `initially` predicate, which is often conditioned on world configuration. Even in the examples above, I have suggested the existence of `room` and `starts_in` predicates as background knowledge used to control the state-and-event logic. Many predicates of this kind can be thought of as specifying a static world configuration: time-invariant facts about the game world, such as level geometry, item properties, and tables of weapon effects.

12.3.2 Player Model Interface

In addition to tagging the subset of game events that are player actions with `player_event`, I build an additional interface for specifying models for players' behavior.

Simply allowing a certain set of events to be considered by the player, by tagging them as player events, does not immediately yield an expressive modeling interface. The `player_asserts` and `player_forbids` predicates can be used to express stronger preferences that an action be taken or should never be taken under some circumstances. The example below illustrates these assertions in the context of a hypothetical game in which picking up objects is a player action:

```
player_asserts(pickup(O)) :- kind(O,gold).
player_forbids(pickup(O1)) :-
       kind(O1,K),
       holds(carrying(O2)),
       kind(O2,K).
```

This model describes a player who never misses a chance to pick up a gold object, but never picks up duplicates (even of gold objects).

The combination of such assertions allows for the specification of a complex, nondeterministic player action policy that automatically re-

[8]The `terminates` case expands to the following AnsProlog rule: "`terminates(hits(A1,A2),alive(A2),T) :- agent(A1;A2), holds(armed(A1),T), not holds(armed`
Ok, enough of this compiler-in-the-footnotes game. The reader can imagine how `T` variables and appropriate domain predicates are inserted as needed to form a valid AnsProlog from here on.

spects the game's mechanics. The default player model is maximally permissive (it reads that the player considers all player events) which allows meaningful play traces to be extracted from games even when no effort has been put into player modeling. Every clause of `player_forbids` or `player_asserts` serves to pare down the space of gameplay traces to those that are more reasonable to expect (thus appropriate) given the provided knowledge.

Identical in structure to the player-modeling interface, I provide `nature_asserts` and `nature_forbids` predicates to operate on the events marked with `nature_event`. Whether the nature model is used to model an opponent, a game master, a collection of non-player characters, or even used at all, is largely a game author's choice. Many of the rules that end up in a nature model could be pushed into the game's core rules, resulting in an equivalent logical model. Leaving certain elements of the game in the realm of the nature model, however, makes it easy to experiment with modification to these policies without modifying the accepted base rules. Having such flexibility is crucial when modeling videogames with a live world, full of enemies, moving platforms, and other active, non-player entities.

12.3.3 Relation to the General Event Calculus

In comparison to the general event calculus, I disallow `holds` and `happens` from ever being used at the head of a rule. In event-calculus terms, that means I disallow state constraints and triggers.

In general, state constraints are not game mechanics. For example, the state constraint that the hero is never at the bottom of a pit while alive might be true of a bit of design, but it is not itself a game rule. There may be rules that produce this state constraint: for example, saying that the hero dies when he hits the bottom of pits (a `terminates` clause); and so would a rule that prevented the hero from ever falling off ledges into pits (a `possible` clause). The result is that every element of state is either only changed by `initiates` and `terminates`, or is known to be completely passive, in the case of state helpers. Disallowing general state constraints ensures that games have well-defined, consistent, procedural meanings (which is required for them to be human-playable).

By disallowing a game's author from directly specifying when an event `happens`, I can ensure that the engine has complete control over event management, allowing me to implement `possible`, `conflicts`, and the player and nature models. The logic required to implement the required semantics of these predicates is highly circular and rather counterintuitive. Since such a mechanism is required for almost any interesting videogame model, by implementing it once and for all in LUDOCORE, a game author need not create an ad-hoc reimplementation of similar constructs for each game

they model.

While I disable rules with `happens` at the head, the engine permits a conceptual equivalent to the use of triggers (rules that specify that an event happens whenever a particular combination of state holds) through the nature model. For example, to model a triggered collision event, the nature model could include a line that says `nature_asserts` the collision event between two objects if their positions are identical at the current time.

In summary, my engine provides a modified view of the event calculus that is designed to make game-level assertions easy to express while minimizing the possibility of a game author (modeler) accidentally introducing purely logical bugs such as deductive loops and contradictions. This lets exploratory designers focus on whether their set of game rules is an interesting game, and how it functions, as opposed to spending much time worrying about whether their set of logical assertions specifies a game at all. It should be clear from Figure 12.1 that a complete, inspectable model of play includes much more than just the event calculus.

12.4 Ludocore in Action

Having defined the major ideas in the game engine, I now use it to generate interesting gameplay traces, and to perform incremental rule modification, world configuration changes, and player modeling.

12.4.1 Gameplay Trace Inference

The simplest use of the engine is to simply generate any gameplay trace compatible with a game's definition. One of the difficulties for game designers is understanding the potential consequences of rule interactions. When initially crafting game models, these unconstrained traces provide easy access to possible interactions within the game, quickly revealing undesirable behaviors resulting from game-design bugs without having to conceive of an interesting query to begin. The output of my tools provides not only a log of game events, but the complete game state at all times to ease diagnosis.

Figure 12.2 shows an automatically generated event trace from the LUDOCORE model of a popular online game, *Motherload*,[9] in which a mining robot recovers ore and treasures from a network of caverns without running out of fuel (my playable model of the game is visualized in the next chapter). In this case, I show both player-action events, such as `mine(a1)` (the player's robot mined the rock named `a1`, which removed

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

```
happens( mine(a1), 0).
happens( drain, 1).
happens( drain, 2).
happens( trade, 3).
happens( refuel, 3).
happens( mine(a2), 4).
happens( mine(a0), 5).
happens( down to(a), 6).
happens( mine(space_canary_corpse), 7).
happens( mine(c0), 8).
happens( down_to(c), 9).
happens( down_to(f), 10).
happens( up_to(c), 11).
happens( up_to(a), 12).
happens( down_to(c), 13).
happens( down_to(f), 14).
```

Figure 12.2: Generated gameplay trace for *DrillBot 6000* (reviewed in detail in Chapter 13), generated by asking the analysis engine for an arbitrary 15-time-step trace with no constraints.

it from the world and added it to the robot's inventory), and natural events, such as `drain` (the robot's energy drains by one unit). The player here mines a single rock reachable from the starting location, and trades it in and refuels. Then, he embarks on a longer mining journey, mining a few rocks and moving downwards into the ground, before finishing with some fairly aimless wandering up and down. The values of each piece of game state at each time step can also be viewed, but we focus here on just the event trace.

While Figure 12.2 shows a trace generated with no special constraints, the most common use of LUDOCORE is combining a game with a set of hypothetical or speculative assumptions (SAs). SAs are commonly used to assert that certain events do happen or never happen, or that a certain condition is met by a certain timepoint (e.g., victory within 20 turns). This helps explore the space in a much more focused way than simply looking at random gameplay traces. For example, while the random wandering up and down in Figure 12.2 is a common feature of unconstrained traces, it does not illustrate anything particularly interesting about the gameplay possibilities. SAs allow a designer to express what would make a gameplay trace particularly appropriate to see at this point in their design process.

The mechanism of specifying speculative assumptions (based on integrity constraints) provides a simple, modular querying interface for narrowing down a model to more interesting traces, and asking questions about gameplay possibilities. "Is my game winnable?" requires only one SA to specify (asserting that `victory` happens at some timepoint). To scope the question further, asking "is my game winnable with

avatar health never dropping below 5?" requires only one more SA.[10] In my model of Motherload, I used SAs to look at extreme kinds of gameplay possibilities: speed-runs by players beating the game in as short a time as possible, or what kinds of gameplay would result if a player never refueled, or refueled cautiously. I compared these traces to traces from human playtesters, who gave me an idea of how beginners would play the game before being familiar with it, which showed me, for example, that my players were *much* more cautious on refueling than necessary.[11]

In generating these kinds of constrained traces, the logical inference approach taken in EC+ASP shines. Had I used randomized forward search in the game's state space, constraints on happenings of the final timepoint would be difficult to encode and inefficient to compute, requiring exhaustive search to prove nonexistence. ASP lets me "run the game backwards" (or even sideways, in a sense) as needed to satisfy trace constraints or quickly prove them impossible.

While gameplay traces can be highly informative (because each includes a complete narrative of events), even the absence of traces can be informative. If adding a particular SA yields no traces for a game that previously admitted many, then it has been proven (by exhaustion) that the SA is incompatible with the game's rules, i.e. that the assumption is false in the game's abstract world.[12]

Extraction of gameplay traces is the primary function of the LUDOCORE engine. The affordances I describe below for modeling and modifying games all serve to give the game's author the ability to sculpt the space of traces that the system will generate.

12.4.2 Modifying Rules and Configuration

Recall that game design is often an iterative process, adding, removing, and tweaking rules as a design progresses. Rules, in LUDOCORE, are represented by the `game_event`, `game_state`, `possible`, `conflicts`, and `initiates`/`terminates` predicates. A game's author might make changes to these rules (and examine the traces for the new game) as part of a major intentional design change, or simply as a way of quickly testing the implications of alternative formulations of a particular mechanic in the game. Often it is useful to disable (by simply commenting-out) several rules to focus on sub-parts

[10] Together, these two SAs look like this:

```
:- not happens(victory,T):timepoint(T).
:- holds(health(H),T), H < 5.
```

[11] I discuss this complementary use of human and machine playtesting in more detail in Chapter 13.

[12] Thielscher (2009) also uses termination without models of an ASP solver to prove properties of a game via contradiction.

of a game in isolation without distracting interactions, e.g. disabling ammunition consumption on weapons when examining health-point management. In a language without elaboration tolerance, this would require more complex editing than simply commenting out a rule. Creating a variant of chess without castling in GDL, for example, cannot be done simply by removing the castling rule, but requires other edits as well.

To illustrate rule modifications, I draw on two examples from my LUDOCORE model of *Motherload*.

If I wanted to add a fixed inventory capacity to the mining robot (constraining an existing event on the basis of existing game state), I would modify the `possible` conditions for the game's `mine` event to depend on a count of the number of items for which the `bagged` state for rocks held. This single change would not only stop traces that violate the new rule from being generated but also reject traces when even an externally assumed narrative (expressed with SAs) included too many mining events:

```
state_helper(have_space) :-
       count(R,holds(bagged(R)),N),
       N < 10.

possible(mine(R)) :-
       holds(present(R)),
       touching(R),
       fueled,
       have_space.
```

To modify the game to charge players energy points for mining instead of moving, I would drop the `initiates` and `terminates` rules linking the moving event to the energy state and create two new ones linking mining to moving (one to terminate the previous energy value and another to initiate the diminished value). This re-wiring task incurs only a four-line change to the game source when using the engine, due to the conciseness of expressions enabled by the game engine. With access to gameplay trace inference, I can quickly verify the effectiveness of the rule modification without the need for human play testers.

This ease of rule modifications directly derives from the elaboration tolerance afforded by the underlying EC formalism. Recall that McCarthy [140] describes elaboration tolerance as the ability (of a knowledge representation) to accept changes to known facts without having to "start over," which is exactly what I realize for the space of game rules. Further, in other research applying EC to games, my colleagues have shown how entire modules (corresponding to game mechanics and vocabularies) can be swapped in and out without trouble [155].

In addition to swapping rules, it is quite easy to pair a fixed game with different sets of configuration such as map layout, static item properties, and tweakable constants. For example, a simple map can be used for early testing, and a more complex one used later for detailed player modeling.

In addition to manually specified game configurations, it is possible to let the answer set solver reason across possible configurations for the game. In my model of the game *Minesweeper*, I am able to reason over possible placements of mines that are consistent with player observations. In a simple dungeon crawl game, I allowed the solver to manipulate the presence and connectivity of rooms on a map. The ability to make structural queries of this sort is a novel feature for game engines. In the larger context of this dissertation, this ability of LUDOCORE is clearly just another instance of modeling design spaces with ASP.

12.4.3 Using Player and Nature Models

While I tend to think of speculative assumptions as convenient but throwaway constraints, specifying complex assumptions can be tedious. The player-modeling interface provides a more straightforward way of building larger player models that will be retained and modified, rather than used in only a few queries. Sets of `player_asserts` and `player_forbids` clauses are collected to carve out the space of behavior that can be reasonably expected of the kinds of players being investigated.

A custom nature model can similarly be used to carve the behavior of other in-game entities. In the dungeon crawl game, I used the nature model to control monster wandering behavior. I had tagged the monster movement event as a nature event, implying that it should always be considered in trace finding when using the default nature model. To effect patrolling behavior, I used the following assertion, which yields monsters who wander only within their own marked areas:

```
nature_forbids(move_to(M,R)) :- not patrolled_by(M,R).
```

12.5 Applications Enabled

LUDOCORE enables a number of broader applications, some of which have already been investigated in other forms.

12.5.1 Game Design

Designing games in a logical game engine, or at least using one to prototype game ideas, can provide designers with insight about design pros and cons, and suggest possible improvements. Many logical formalisms can allow verification that desired properties hold, and undesired properties do not. However, designers often work in an iterative, exploratory manner, and find exact yes-or-no queries somewhat difficult to formulate

[156]. The experimentation with gameplay traces that the engine supports can help designers understand the possibilities of their design by iteratively "zooming in" on specific kinds of traces using SAs, and observing how different player models shape emergent properties of the game. I have already begun pursuing this application to larger design process, creating a game the prototyping system BIPED in which human-playable, board-game-like prototypes can be built, using the logical representations described in this chapter.

Logical game engines may also form a core component of emerging research in automated videogame design. In a subsymbolic approach, Togelius and Schmidhuber [219] generate *Pac-Man* variants from a parameterized space and use reinforcement learning to demonstrate gameplay in the resulting games. Games implemented in LUDOCORE can be varied in a much more open-ended and incremental manner than with a parameterized space of variation, and its elaboration tolerance addresses the problem of brittleness of symbolic representations, which was in part responsible for the move towards parameterized numerical representations in content-generation research.

Furthermore, given sets of traces, inductive logic programming (ILP) can be used to induce models for the style of gameplay exhibited in those traces. The player-modeling interface in LUDOCORE makes this more feasible by having player models built from only two key predicates. Many popular ILP systems, such as Progol [153] can only learn single predicates at a time. In LUDOCORE's player-modeling interface, single-predicate ILP systems can learn the `player_asserts` predicate from example gameplay traces. Some research has even been done on learning EC rules themselves [149], which, for games, would open up the possibility of inducing new game rules from a collection of desired gameplay traces.

Procedural level design can be done by specifying and solving constraints for what constitutes a good level, sometimes as part of a larger search process. Particularly relevant here, a community-driven project recently[13] added procedurally generated maps and base layouts to the real-time strategy game *Warzone 2100*, using ASP to specify and solve constraints on the maps' layouts.[14] Designing a good level, however, often means designing a level that supports good gameplay, and these sorts of static constraints only indirectly speak to gameplay. Other approaches, such as that of Togelius et al. [217], generate levels and score them with a fitness function that predicts whether they support interesting gameplay. For games written using my logical game engine, a constraint-based approach can easily include these kinds of constraints on gameplay in addition to constraints on static level properties, because the level-generation

[13] Well, it was recent as of the time of this research.

[14] `http://warzone2100.org.uk/manual-diorama`

search and play-trace search are unified into the same query mechanism.

12.5.2 Crafting Game Playing Agents

Despite my motivating focus on analyzing game designs, my logical game engine has applications to playing games as well. Because gameplay trace inference in the engine corresponds exactly to what is commonly known as EC planning [186], it is possible to use the engine directly in a general game player. Thielscher [214], for example, has already argued for the applicability of ASP with temporal-logic models for the contemplation phase of GGP competitions.

The player-modeling interface I provide is designed to accept incremental additions of knowledge about how a player (or their opponents) make choices. Though this interface cannot express a minimax-like policy (that includes quantification over models at every timepoint), it does coalesce overlapping asserts/forbids cases into a unified move-set selection policy that would allow minimax to operate in the more restricted space in which the modeled opponent is really playing.

The ease of adding and removing rules in our games has another benefit for those wishing to craft general game players. The elaboration tolerance of the representation makes syntactic construction of novel games much easier. By building all combinations of a fixed set of add-on mechanics, a generative space of games can be realized, providing a much wider selection of games for agents to be tested on.

12.6 Conclusion

In this chapter, I introduced a new concept: the logical game engine. My logical game engine, LUDOCORE, much like a traditional game engine, both provides a higher-level language to describe games, and centralized solutions to tedious or error-prone programming tasks. By virtue of using declarative logic, I gain not only a concise representation of a game's mechanics, but also the ability to automatically generate interesting gameplay traces that meet meaningful constraints.

While clearly LUDOCORE serves as a bridge from the concerns of game design to symbolic AI tools, it has also served as the basis for implementing interactive prototypes. In the next section, human-playtestable prototypes with real-time interaction and graphics are written with the LUDOCORE engine, enabling the same game specifications to both be used with logical tools (objective, machine playtesting) and as gameplay demos (subjective, human playtesting).

Chapter 13

Biped

My second applied system extends the gameplay models expressible with LUDOCORE with an interactive, graphical interface suitable for direct playtesting with human players. While this system does not employ any design space models beyond those in LUDOCORE, it provides an example of linking the highly abstract artifacts reasoned about by answer set solvers to the playable systems designers expect to be manipulating in game design.

In this chapter, I describe a language designers may use to sketch simultaneously interactive and formally queryable games, how they might use the tool in the two modes of playtesting (human and machine), and how the prototypes are computationally realized. Additionally, I study using the system to prototype a game and examine it in human and machine play tests.

13.1 Overview

Prototypes, both physical and computational, are an essential tool in a game designer's arsenal. In a popular game-design text, Fullerton [73] defines a prototype to be "a working model of your idea that allows you to test its feasibility and make improvements to it." At the earliest stages, the designer can operate purely conceptually, thinking up new mechanics and imagining how they might interact with each other and the player; but at some point all designers need to try them out. This has been described as getting "backtalk" from a design situation: the designer begins with some idea of how her design should work, but by trying out a prototype, she uncovers unforeseen new information on how it actually does work [182]. In game prototyping, that new information can be both in terms

of player experience, observing player's excitement or hesitation to take a critical action; and in terms of how the game's rule system operates, laying bare gameplay exploits and additional solutions to counterintuitive puzzles.

Physical prototypes are an early-stage game design and playtesting tool, in which low-fidelity versions of a game's core mechanics are mocked up. Even a complex real-time videogame can be prototyped using cards, tokens, dice, and similar physical objects [189][73, chap. 7]. These prototypes aim to let the designer get backtalk after minimal effort, before committing to the cost of building a computational prototype (though it may provide higher-fidelity backtalk). An important aspect of low-commitment techniques is the ease of exploring related ideas.

Recall that there is some gray area between physical prototypes and computational prototypes. One common way of using a computer to run the game's rules while keeping things lightweight is to use a spreadsheet, either stand-alone or as a computational aid to physical prototypes. The designer writes the game rules in a scriptable spreadsheet, and has it update game state as players take actions [73, p. 216, 221, 246]. This style of prototyping is best suited to numerically oriented games, such as physical or economic simulations. However, many of the elements used in physical prototypes do not transfer well to spreadsheets. Instead, there are discrete cards and tokens, and rule systems with many cases and spatial rearrangement of the tokens, rather than equations that update numerical fields.

I propose a game sketching approach to provide computational support for the class of physical prototypes that use board-game like elements to represent complex videogames. Through this approach, game designers can access formal analysis afforded by computational approaches while designing prototypes similar to those made with physical methods. The designer specifies game state, possible player actions, and state update rules (in the language of Chapter 12's LUDOCORE system). Elements of game state can be mapped on to a simple set of board-game primitives—connected spaces and tokens—and user actions are mediated by UI affordances such as clicking and dragging, constituting a fully playable game suitable for early playtesting. Using logical reasoning, I also allow the designer to query formal properties of their sketch, such as getting examples of possible gameplay traces resulting in specific outcomes. My contribution here is a technique that supports both machine and human playtesting, using a single game-sketch definition, to provide designers with the synergy of two complementary sources of backtalk. It is these two sturdy legs for which the system presented in this chapter, BIPED, is named.

13.2 Playtesting Background

What can designers get out of playtesting, and why would they need both kinds? I review discussions of human playtesting in the game design literature, and proposals for machine playtesting in the artificial intelligence literature, to provide a background for the kinds of design backtalk that have been proposed as particular strengths of each approach, especially as applied to early game prototyping.

13.2.1 Playtesting with Humans

Fullerton [73, chap. 7] has an extensive discussion of early prototyping and the kinds of design questions playtesting with them can answer. She describes four main stages of prototyping. In the first, the designer focuses on foundations, looking at only a loose version of the core mechanics and avoiding filling in details like how many squares a player can move; the designer self-tests to validate and flesh out the design. In the second, the focus is on the game's structure, specifying rigid mechanics such as combat-resolution rules. The third stage fills in formal details and tweakable constants, such as starting health level and hit percentages, as well as minor elaborations on mechanics. In the last stage, the designer focuses on game flow and understanding how features of the game contribute to the play experience.

Implicit in most discussions of playtesting is that important elements of gameplay come from intrinsically subjective human reactions. Koster's theory of fun [114] focuses in particular on fun and engagement and their relation to an individual's process of learning a game's mechanics. Eladhari and Mateas [64] discuss feedback from paper prototypes testing a game mechanic derived from personality theory, with playtests focusing on how the mechanics connect to players' personalities and subjective gameplay experiences.

13.2.2 Playtesting with Machines

Although the subjective human experience of games is the key to their success, designing games involves crafting formal rule systems and understanding how they operate. Salen and Zimmerman [176, chap. 12] discuss emergent properties of game rules, since a small set of rules might actually imply a larger set that are not explicitly specified; understanding these implications can help a designer decide how to revise the game towards its intended subjective experience. Since the formal properties of game rules are amenable to automated reasoning, Nelson and Mateas [156] study what kinds of objective questions designers would find useful

to ask; automatically providing answers to those questions would free up the designers to focus on more subjective design issues.

Salge et al. [177] apply artificial intelligence to playtesting, going as far as to argue that automatic testing can be superior to human testing, because human testers instinctively try to play in a balanced and fair style. An automated agent, by contrast, could look for exploits in the game rules without preconceived notions of how the game should be played. Although this perspective is overly pessimistic about insights derived from human playtesters, there is legitimate value to be derived from how machines play *differently* than their human counterparts.

In work predating LUDOCORE, Nelson and Mateas [155] propose that designers prototype their early rule systems in a formal language, in exchange for which they can receive abstract gameplay traces. These play traces may illustrate interesting or undesirable gameplay possibilities that motivate a change, finding early issues with the game rules before spending time on human playtesting (or even writing code for input and graphics).

13.3 System Overview

The architecture of BIPED, with its dual support for human and machine playtesting, is shown in Figure 13.1. A designer using the system begins by writing a game sketch. This sketch is combined with an analysis engine (in reality, the LUDOCORE system) to produce a formal rule system on which the designer can perform machine playtesting. To do so, she specifies combinations of queries and constraints and can get back abstract gameplay traces with specific characteristics, determine implied properties of the rule system, find exploits, and list solutions to puzzles. The sketch can also be combined with a more traditional game engine (one supporting graphics and sound) to produce a playable prototype, which, when the designer or human subjects she recruits play it, can give feedback on more subjective properties of the game, such as player engagement, fun, or hesitation, as well as traces of actual gameplay. From these two sources of backtalk, design insight may prompt the designer to perform another iteration. I should emphasize that my approach focuses on early design prototypes, not late-stage prototype implementations of a full game.

13.3.1 Game Sketching Language

Game sketches are defined in a subset of Prolog (chosen both for the logic-inspired style and the similarity to the internal languages expected by the

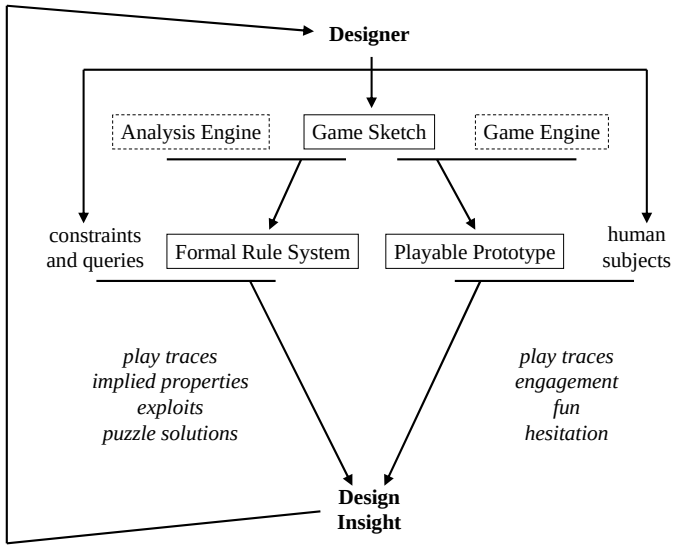

Figure 13.1: Architecture diagram for the BIPED system.

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

Figure 13.2: Snippet of the game sketch language defining part of an inventory mechanic. The last rule, for example, says: the game event that an agent drops an item is only possible if the agent has the item, for any agent or item. For the specification of a game's mechanics, the language of BIPED coincides with the language of LUDOCORE.

formal analysis and human gameplay subsystems); an example defining an inventory mechanic is shown in Figure 13.2. The designer can use logical predicates and constants to specify the entities in the game world and their properties. In this example, there are two agents, the player character and a thief; as well as two items, a bunch of loose coins and set of assorted gems. Entire levels can be described in a similar manner, specifying, for example, the rooms of a dungeon and hallways between them.

The language comprises a superset of those predicates used in LUDOCORE. These can be used to specify the dynamic parts of the sketch, organized around *game state* and *game events*. The game state is a set of properties that vary over time, under the influence of game events. The game engine and analysis engine are implemented so as to provide a common semantics for these predicates, ensuring what is possible in the human-playable version is possible in the logical version and vice versa. Returning to the example in Figure 13.2, the *has* game state specifies, for any agent and item, whether the agent has that item. The next rule specifies a *drop* event (for any agent and item). The following rule gives the event's effect: it terminates the *has* state (sets it to false). The final rule specifies that a drop event is only possible when the *has* state *holds* (is true).

13.3.2 Interface elements

To complete the prototype, a set of interactive visual elements are available to specify a game's playable representation. Based on the kinds of representations often used in simple physical prototypes, the main interface elements are clickable *tokens* and *spaces*, with optional connecting *lines* between spaces. Figure 13.3 shows an example of the interface presented to the player. Figure 13.4 gives a simple example of setting up interface elements and connecting them to a game world. In this code example, there is a visual board space for every room in the game world (as well as a token for every item). Clicking on a board space that represents a room is set to trigger a `move_to` event in the game world for that room (similarly for tokens and grabbing of items).

As with actual physical prototypes, there are many ways of using the visual elements to represent the state of the game world, and mappings need not all be as direct as in this example. Spaces can instead be used to represent inventory, with tokens on the space representing items in the inventory; or they can be buttons, with clicks causing a state change that may be reflected in a different part of the visual representation; or space and token combinations can represent turns or phases of the game (just as a physical dealer token is passed around a poker table).

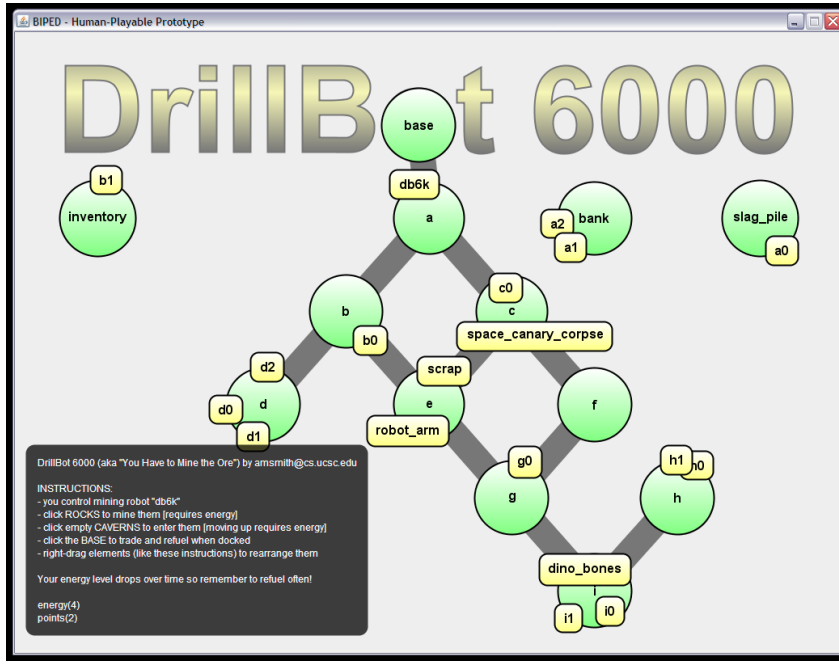

Figure 13.3: Human-playable prototype for our example game: *DrillBot 6000*, a logical *de*make of *Motherload*.

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

Figure 13.4: Bindings from UI elements to a game world. All *ui_** predicates are specific to BIPED's human-facing side and would have no meaning for LUDOCORE alone.

Similarly, connecting lines need not only represent how tokens can move between spaces; they might represent other relationships between entities represented by the spaces, or elements on which a button would operate when pressed. As a result of this flexibility, the designer does need to recall that there is no automatic meaning of the UI elements in terms of the game word: instead, it is their responsibility to give the representation elements game-relevant semantics via triggers that attach them to the game's abstract mechanics.

In addition to the visible elements, the event system is a major representational tool. Time in the game world logically stands still until an event happens. The only sources of events are mouse interactions and expiration of timers. The effect of a living game world can be achieved using a *ticker* to create a regular stream of tick events, which can trigger interesting game-world events. The *ui_triggers* predicate defines these mappings from mouse-interaction and timer events to game-world events; it checks to ensure that it only triggers game-world events if they are inferred to be *possible* at the time.

I have been able to mock up a wide range of physical prototypes using this set of elements, and have avoided introducing a larger range of special-purpose elements in order to keep the prototypes simple and easy to modify. I have, however, included a number of aesthetic elements that do not directly represent game state, such as instructions, title, victory/loss animations, and background music, that can be used to provide an interpretive framework for the human playtesters. I acknowledge that aesthetics of even early-stage prototypes can set the mood for the game and impact the subjective play experience, even if they are not the main focus [189]. In particular, end conditions that support a subjective distinction between good and bad outcomes are a key feature of games [101] (though many of my BIPED prototypes did not exercise the related UI support for this).

13.3.3 Supporting Playtesting

To playtest games created using BIPED, the designer may begin with either machine or human playtesting; I have found that each process informs the other, so alternating between them is strongly recommended.

Human playtesting often begins with self-testing. The designer loads a game definition that she has created into the game engine, and is presented with an initial playable game. Even before mechanics are fully specified, it is often gratifying to see on-screen a level previously only described in the abstract. As the mechanics of a paper prototype are added to the first computational prototype, many design decisions have to be made to create the desired rigid rule system. A lightweight cycle of revision

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

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

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

display_to(time(1),jack,
       health(1),
       enemies_at(left)).

display_to(time(1),jack,
       health(1),
       self_at(right)).

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

display_to(time(2),jill,
       health(2),
       enemies_at(right)).

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

Figure 13.5: Partial trace from a multiplayer shooter game prototype, illustrating events and game state over time.

followed by self-testing can allow the designer to quickly flesh out these rules while playing the game themselves, giving a first glimpse at the gameplay possibilities afforded by their sketch.

When testing with others, it is important to formalize any parts of the game that the designer may have been keeping in her head during self-testing. By utilizing timers, on-screen instructions, and background music, the designer can externalize the desired dynamics and mood of the game (e.g. fast-paced and frenzied). The designer, who can then observe player engagement and hesitation, as well as make verbal clarifications, best mediates human playtesting. Because playable prototypes from BIPED are user-tweakable, standalone sketches, however, they can be sent to and played by remote individuals as well (unlike physical prototypes or computational prototypes with extensive dependencies).

Playtesting with humans is a well-established practice; in contrast, playtesting with machines is a relatively new, speculative idea. Thus, I have opted to focus on support for extracting gameplay traces, because these seem to be strongly tied to player experience (as opposed to generating game-playing agents).

A designer can specify scenarios and conditions, and the analysis en-

gine will provide her with gameplay traces (if any exist) starting from those scenarios and meeting those conditions. Figure 13.5 shows a short excerpt of a trace from a multiplayer shooter prototype, in which an agent, interestingly, inflicts damage on himself. To look for an exploit, the designer might ask for traces starting in a particularly tricky scenario, which end in victory only a few timesteps later. If any traces exist, the designer has a specific example of the behavior to forbid in the game rules; if not, the engine has proved that no such exploit is possible (which would be difficult to determine with certainty using only human playtesting). In cases where there are many uninteresting traces, the designer can restrict the conditions in order to "zoom in" on more plausible gameplay (using a mixture of player modeling and speculative assumptions). Alternatively, the designer can run both human and machine experiments in which the scenario and conditions are held constant, and some element of the rules is changed. This gives the designer backtalk regarding a possible rule change rather than more detailed inspection of a particular rule set.

13.3.4 Implementation

My implementation is primarily split between two engines. The interactive game engine, supporting human-playable prototypes, was implemented in the Scala programming language, which targets the Java Virtual Machine for easy redistribution of games. The rules of a particular game, implemented in Prolog, are interpreted using jTrolog,[1] a Prolog interpreter for Java, so game sketches can be modified by end users without recompilation. The game engine executes the game rules by advancing time in response to UI events, querying the Prolog interpreter to find whether visual elements have changed (e.g. whether tokens have moved), and which elements of game state hold on the next time step.

The analysis engine (LUDOCORE) was implemented in Lparse/Smodels,[2] an ASP system that has since been largely obsoleted by more recent advances in answer set solving. A source-to-source compiler written in Prolog translates the game-sketch definitions into AnsProlog. Additionally, a small AnsProlog library ties the rest of the game sketch (excluding UI) into the event calculus semantics. This compiler is also responsible for checking that a game sketch is complete (defining a minimal set of required predicates) and checking any additional designer-specified static properties (e.g. that there are no rooms in the level description without an adjoining hallway).

An interesting result of this translation is that when the analysis engine is looking for traces, it treats time as a symbolic constraint rather

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

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

than simulating game worlds forward in time. In this way, it is as easy to put constraints on initial conditions as it is on end conditions, or on any point in between. In the game engine behind human-playable prototypes, logical time, while discrete, behaves much more intuitively, advancing step by step in response to timers and human interaction (effectively finding a single trace).

13.4 Example Prototype

To exercise playtesting with BIPED beyond those prototypes created with it during the development of the system, we created *DrillBot 6000* (previously illustrated in Figure 13.3). In this game, the player moves a mining robot through underground caverns, drilling out rocks of mixed value, while managing energy usage by periodically returning to the surface to refuel and trade items. This game was designed as if to be an early prototype version of the popular Flash game *Motherload* from XGen Studios.[3] My game focused on the core mechanics: moving around underground, mining, and refueling (whereas *Motherload* additionally includes shopping for upgrades and story elements).

13.4.1 Game Mechanics

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

13.4.2 UI Bindings

While the game mechanics described above are enough to allow machine playtesting, for human testing I needed to expose the abstract game world

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

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

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

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

A game definition also specifies several non-interactive elements. A large title and an original[4] background music track set the tone for a lively real-time mining game. On-screen, written instructions lay out both the premise of the game and explain how the player can use the mouse to take actions in the world of rocks, caverns, and a robot. Some elements of game state not mapped to spaces and tokens are depicted textually, in this case energy level and score. Finally, when the game determines that no additional actions are possible, a few tokens fly onto the screen to announce the game's outcome.

[4]Because I spent *far* more time making the one-minute loop of background music than I did on the rest of *DrillBot 6000*, I feel compelled to share a link to the result: `http://adamsmith.as/music/reason/rsmy-db6k.mp3`

13.4.3 Human Playtesting

As suggested by Fullerton [73, p. 252], since I was testing the foundations and structure of the game, I primarily tested *DrillBot 6000* by self-testing and testing with confidants. Self-testing revealed that my early iterations had allowed unintended gameplay; for example, one could mine a rock at arbitrary distances. Additionally, I found the first version of the game too simple, and decided to add several additional rocks and caverns. When testing with others, one tester initially felt pressured by the speed of the automatic energy drain. While I could have adjusted the speed of energy drain or the maximum energy level, at this stage of the prototype I was interested in more foundational questions, rather than game balancing. To get feedback on the game's other mechanics, I showed the tester how to convert the game to a turn-based one by removing the ticker. All three testers claimed to enjoy the game (i.e. the game's abstract depiction was not so jarring that the artifact could not be related to as a functioning game), and could reach rocks at the deeper levels after some practice. Interestingly, no testers related to the game as a puzzle or path-planning challenge, even in turn-based mode; all focused instead on the action aspect of the game motivated by the continuously draining energy.

13.4.4 Machine Playtesting

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

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

In the setup for this experiment, when looking at general gameplay

traces, I found undesirable traces revealing that several rocks from the same cavern could be mined simultaneously (even worse, with only a single unit of energy drain). This revealed that I had not completely specified when game actions should conflict. That issue was not revealed in human testing, because the UI bindings happened to make it impossible to mine several rocks at the same time (which would not be the case if mining had been tied to clicks on spaces instead of on tokens). Finding that this design choice was incompletely specified both forced me to think about what mechanics I actually did want, and avoided the persistence of a UI-masked bug that could reappear in later prototypes.

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

13.5 Conclusions and Future Work

I have proposed a computational prototyping approach to combined support for human and machine playtesting of early-stage game sketches. This approach has been cashed out in Biped, a tool that uses declarative game descriptions to produce both standalone, playable games and formal rule systems that admit automatically finding gameplay traces with specific properties (while minimizing design commitments, yet allowing sufficient interpretive affordances to support useful human playtesting). I explored the process of designing with Biped by prototyping *DrillBot 6000*, an action-oriented game in the style of *Motherload*. In doing so, I illustrated the synergy of human and machine playtesting for insight into the game's design.

The *semantics* of my game-sketching language are based on a knowledge representation that is optimized for designer exploration, while the syntax used here was chosen in large part to minimize the distance to the two declarative reasoning engines involved. In future work, I intend to address the language's accessibility, particularly to game designers with-

out a logic-programming background; it is best to think of the language presented in this chapter as an intermediate representation (a target of compilation from a more designer-friendly format). In addition, future work should investigate other reasoning back ends for the analysis engine with better scaling and numeric processing capabilities.

Several miscellaneous improvements to this approach could be unified in a game-design "workbench" that bundles all the tools necessary for a designer to carry out exploratory game design, including a better query language paired with more intuitive visualization of results. Additionally, such a tool should provide a more fluid way of modifying a game's mechanics than editing a logic program.

A tool like BIPED is an interesting target platform for automated game generation research (whether aiming to create whole new mechanics or just tweaking level designs). Moving beyond generation, the addition of detailed instrumentation to the human-playable prototypes would admit collecting enough information from human play testers to inform the analysis used in an intelligent game-design agent that might learn from how humans play the games it produces.

Finally, an open problem is how to extend the human–machine playtesting approach to later stages of the game-design process. For now, my approach is focused on allowing designer-programmers to take large creative leaps by getting rich feedback from simple, early-stage prototypes.

Chapter 14

Variations Forever

LUDOCORE and BIPED consumed rule systems produced by game designers, but producing these rules in the first place is, of course, another area where design automation would be useful. In an instance of *PCG-based game design* (in the sense examined by Gillian Smith and others [201]), this section investigates the co-formation of a novel game and the playtime design automation needed to support it.

Variations Forever is a novel game in which the player explores a vast design space of mini-games. In this section, I present the procedural content generation research that makes the automatic generation of suitable game rulesets possible. My generator, operating in the domain of code-like game content exploits answer-set programming as a means to declaratively represent a design space as distinct from the domain-independent solvers that I use to enumerate it. These design spaces are powerfully sculptable using concise, declarative rules, allowing me to embed significant design knowledge into the ruleset generator as an important step towards a more serious automation of the larger game design process.

14.1 Overview

Automatic generators exist for many game content domains: 2D textures, 3D models, music, level maps, story segments, ships and weapons, items and quests, character attributes, etc. In terms of a distinction between code and data, these kinds of content feel like data and they are interpreted by the fixed code inside game engines. However, some kinds of content such as location-based triggers on a map, behavior trees, or the contents of game scripts blur the line between game data and game code (between representational and behavioral components). The field of game

research known as procedural content generation (PCG) can be expanded to include richer aspects of game design if the "content" that is generated includes the kind of conditionally-executing logic that we would otherwise call a game's mechanics.

While PCG is often motivated as a means to reduce development effort or costs for game content [168], it can also provide access to richer, and more personalized, play experiences than could be reasonably hand-authored by human designers. The rich worlds of *Dwarf Fortress* (Bay 12 Games 2006) include procedurally generated multi-level landscapes and thousands of years of dwarven history. Meanwhile, the player-designed creatures of *Spore* (Maxis 2008) are enhanced with unique, procedurally generated skin details and body animations. Further, these personalized creatures are used to populate a vast, procedurally generated galaxy reminiscent of the seminal PCG work seen in *Elite* (Acornsoft 1984). These games had impressive generated data, but employed hand-authored mechanics.

Though the automatic generation of game mechanics is an important and underexplored component of *automated game design*, it is important not to collapse the part with the whole (which has been done in the literature [94]). Nelson & Mateas [154] proposed a factoring of game design into four domains: abstract game mechanics (abstract game state and how this state evolves over time), concrete game representation (the audio-visual representation of the abstract game state), thematic content (real-world references), and control mapping (relation between physical player input and abstract game state). While I can imagine a procedural generator for the content of any of these domains, even this would miss out on an opportunity to illuminate several commonly accepted processes in game designs that cross-cut these domains. Conceptualization, prototyping, playtesting and tuning are essential parts of game design [73]; there is no compelling reason to think they should not also be addressed in a nuanced automation of game design. However, addressing only a piece of the whole game design process, the research presented in this section focuses on flexibly generating a variety of abstract mechanics, utilizing hand-authored components for concrete representations, thematic content, and control mapping.

Variations Forever (VF) is the name of both a work-in-progress[1] game and the research project of developing the technology necessary to implement it. In the remainder of this section I will distinguish these two projects.

[1] As of 2012, this game project is actually long dead. It turns out that the design decisions made in many game engines are incompatible with deep reconfiguration of game mechanics as play-time.

14.1.1 *Variations Forever* as a Game Project

VF, the game, aims to provide players with the experience of exploring a generative space of games as an in-game activity. The premise, visual style, and themes employed by the game are inspired by a set of recent, independent games. *ROM CHECK FAIL*[2] is a glitch-laden arcade game in which the player's avatar, movement mechanics, level design, theme music, and enemy mechanics shift at regular intervals. In the unique, emergent meta-game, the overarching goal is simply to survive the onslaught of new mini-games. In VF, however, the meta-game will involve unlocking new mini-game design elements which reshape the space of mini-games.

Warning Forever,[3] *Battleships Forever*,[4] and *Captain Forever*[5] each have a space combat setting with glowing vector art. Beyond aesthetics, they share the theme of recombining elementary parts in novel ways in the player's major choices (assembling a spacecraft from standard modules such as girders, thrusters, and weapons). In VF, the recombinant nature will shift from ship design to ruleset design.

This chapter presents a prototype of VF (depicted in Figure 14.1) in which I have realized a large space of varied mini-games. This prototype does not yet include player-control over the design space; however, I will show how the approach supports such functionality.

14.1.2 *Variations Forever* as a Research Project

The goal of VF, the research project, is to create a means to automatically explore a generative space of game rulesets that supports both the variety of mini-games I desire and the hooks needed to place the exploration into players' hands. My emphasis is on flexibility of generation, and I leave evaluation of game quality for future research.

I have adopted a symbolic approach to representing games because I believe that breaking free of the parameter-vector paradigm pervasive in PCG will be required to address the larger automation of game design. Reasoning through the intentional creation of prototypes, the identification of useful mechanics and generation of hypotheses to validate in playtesting requires a representation which resonates with the symbolic, modular, non-parametric medium used to implement every videogame: *code*.

In this chapter, I also describe a flexible approach to game ruleset generation that should also be of interest to those PCG researchers working in

[2] `http://db.tigsource.com/games/rom-check-fail`
[3] `http://db.tigsource.com/games/warning-forever`
[4] `http://db.tigsource.com/games/battleships-forever`
[5] `http://db.tigsource.com/games/captain-forever`

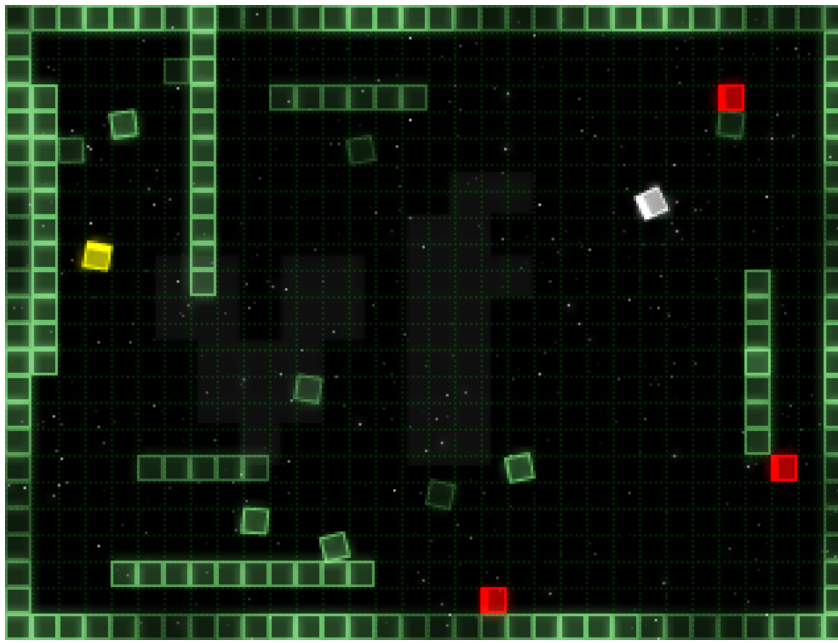

Figure 14.1: Screenshot of gameplay for a generated mini-game in the *Variations Forever* prototype. The player controls the white character using an *Asteroids*-inspired movement model, trying to touch all red characters which move via a *Pac-Man*-inspired movement model. The encircling walls, random-walls and random-blocks algorithms have generated dangerous obstacles which can harm the players square. This particular games rules also define the stars and grid backdrop details as well as a third kind of character (yellow) which drifts on its own.

traditional content domains such as level or map generation, etc. While I will use VF as a running example, the schema I provide for creating generators is not inherently tied to the ruleset generation domain.

My contribution here is a content generation approach based on constraint logic programming that allows the declarative specification of design spaces and uses domain-independent solvers to sample these spaces. My application to ruleset generation demonstrates the representational flexibility and ease of incremental modification that comes with the use of a symbolic representation. Such flexibility and modifiability are critical to integrating the generation of rulesets into larger-scale game design automation efforts.

14.2 Related Work

Automated game design has been studied from a number of different angles. An important bit of vocabulary I can use to reconcile these efforts is "generate and test" a popular discussion topic on a PCG community mailing list.[6] Generation tells us how artifacts of interest come into existence, and testing tells us how these artifacts are separated from their less-interesting neighbors in some space (generally involving both evaluation and filtering).

A recent example of generating and testing simple game designs is seen in the work by Togelius and Schmidhuber [219] in which a space of *Pac-Man*-like games is automatically explored using evolutionary computation. Game variants, represented as fixed-length vectors of integer parameters which encode quantitative properties of a game such as its time and score limit as well as basic qualitative data such as which movement logic or collision effects are used by the various "things" in the world. The mechanics of the games generated by this system are the result of combining the parameter vector with some fixed rules defining the meaning of each parameter. The mechanics of the mini-games I consider in VF are directly inspired by this work.

Togelius' parameter vectors use an operationalization of Koster's theory of fun [114] based on reinforcement learning. The system illustrates that while games may be generated by some simple, syntactic method, they must be played to understand their semantics and to assign them value. While acknowledging the depth of automatic playtesting as both a computer-science and game-design problem, VF focuses on bringing flexibility to generation. Where Togelius' games all involved exactly four colors of "things," VF's generator is capable of choosing an arbitrary num-

[6] `https://groups.google.com/d/msg/proceduralcontent/TUVfuus1zKw/GzHdGLAKzTIJ`

ber of characters types and populating the appropriately sized collision effects table to describe their mechanics (such scalability in terms of set cardinality is awkward for fixed-length parameter vectors).

Hom & Marks' [94] project in generating balanced board games also took an evolutionary approach, opting for three-way crossover as the means of generating games represented by a tuple of board type, piece type, and victory condition. Testing, in this project, was done using general game playing software to look at the relative win rates for the first vs. second players.

In addition to the main tuple, the representation also allowed optional rule modification tags that could be appended to a game ruleset. While these tags could have been folded into additional Boolean values in the genetic description, the authors decided to treat them specially to simplify the crossover logic of their generator. This also points to awkwardness in using parameter-vector techniques to represent rulesets. Incidentally, the general game-playing tool they used already expected a code-like representation as input.

Automation of ruleset generation has also been addressed without including a distinct test component. The METAGAME system [165] and the EGGG system [161] are both capable of generating games as complex as Chess without any testing. However, this ability comes at the cost of an immense knowledge engineering effort to embed as much intuition about the game design space into the generator as possible. METAGAME's author refers to the generator as "long and complicated and full of *special cases*". These systems illustrate that generators can actually contain and represent large amounts of domain knowledge. It is desirable to have a generator that is improvable, that is, can be easily augmented with new knowledge which helps it avoid uninteresting or problematic regions in the design space. My generation approach indeed aims to ease the process of adding new knowledge to a generator (particularly that which focuses the generative space).

In a filtering-heavy approach to designing games, Nelson & Mateas use a common-sense knowledgebase to filter out only those combinations of mechanics and representational art that "make sense" from a larger generative space [154]. Because the mechanics used in their *WarioWare*-style games are so simple, detailed playtesting is not required for evaluation. Rather a common-sense knowledgebase is effective in filtering a game where a duck avoids a bullet shot by a gun (a reasonable premise) from a games where a person fills a piano with ducks (less reasonable), both of which are in the latent generative space tested by this system. For a given target concept, such as "duck," the system might produce any of a small space of games which both make sense and include the target concept. This project illustrates that filtering one generative space to make

another is an important technique in design automation.

Finally, though it only consumes and does not produce rulesets, the dual human-and-machine playtesting supported by my BIPED system is relevant. This playtesting tool reads in rulesets represented as logic programs and produces gameplay traces in a similar logical representation that are derived either from human players or from logic programming tools which can solve for edge and limit cases in a game's design. This project shows that both humans and machines can comfortably use symbolic, logical representations for game rulesets and they can serve as an effective interchange format for the playtesting stage of design.

14.3 Representing Game Rulesets in Logical Terms

To put ASP to work in the ruleset generation domain, I need to somehow represent elements of game rulesets in the heads of ASP's choice rules and encode the logical conditions which ensure the generated rulesets are valid into the bodies of these rules. This implies a representation of ruleset elements as logical terms.

Though logical terms are a standard knowledge representation format in AI, I review them here because they have not been used in published PCG work previously. A logical term is either an atom or a compound term. Atoms are symbols, numerical constants, or logical variables. Compound terms combine a symbol called a functor with a sequence of logical terms as arguments, as in *afraid_of(6,7)*.

An example of a logical term encoding an element of a game script is the following: *scripted_event(spawn(boss_creature, temple), 120)*. This term, if it were asserted in an answer set, might mean "a boss-class creature should be spawned in the temple after two minutes of play". The meaning given to `scripted_event` is given by the code that consumes it, which might be other rules in the answer set program or the game engines that delivers this content to the player.

The rules for the kind of games I am considering for VF contain various types of information: collections of objects that participate in the game and their properties, policies for handling events that arise during play, conditions under which we can consider the game to end in victory or defeat, and, additionally, miscellaneous procedures and configuration details that I can use to add to the variety of play experiences. Lists, variable sized look-up tables, and nested expressions are all difficult to represent with fixed-length parameter vectors.

Each of the elements I would like to include in rulesets has a straightforward representation in logical terms. Lists, such as a list of valid move

types, are represented by a pattern of terms that may be instantiated several times: "*move(rock). move(paper). move(scissors).*" (a numerical argument may me added to represent a strict ordering if needed). Tables, such as a mapping from event to a handling character's response action with a performance modifier, are represented by simply asserting the presence of each tuple of the data the table contains: "*on(poke, giggle, quietly). on(jab, yelp, loudly). on(stab, die, slowly).*". Code-like nested expressions are also natural:

```
when( equal(health,0)
    , go_state(defeat) ).
```

While I could use a simple, context-free grammar to syntactically generate masses of such terms, building my generator as an answer set program gives me the means to powerfully sculpt the space of generated rulesets using the same language used to define it. For example, when generating terms of the form "*on(poke, giggle, Adverb)*" I can require that the adverb come from a table of modifiers compatible with the giggle action or forbid the use of certain adverbs in conjunction with the poke event by consulting a blacklist of known problem cases. Further, such a table or blacklist might itself be the output being simultaneously produced by another part of the same generator. The ability to specify defaults and override them with many levels of specialization, important for flexible modeling, comes from the non-monotonic reasoning used in ASP solvers.

14.4 VF's Generative Space

Having introduced a representation of game rulesets in its terms, I now describe how the concrete elements of rulesets for VF are generated and how these elements influence one another. The generative space of the VF game prototype is meant to exercise the expressiveness of the generator's logical formulation and does not yet represent the complete set elements in the other *Forever* games I intend to reference. Figure 14.1 provides a visual guide for one element of VF's generative space.

A basic element used by all mini-games in VF is the play space. It is always rectangular, but has a numerical grid resolution parameter that is used by certain character movement models and obstacle-placement policies. Specifying the simple selection of a numerical parameter looks like this:

```
resolution_factor(2;3;4;6;8;12;16).
1 {space_resolution(4*F) :resolution_factor(F)} 1.
```

This snippet asserts that it is true that several numerical symbols are valid resolution factors, and that exactly one ground clause of the `space_resolution` predicate should be emitted in answer sets. The clause in

curly braces is a choice rule that gives permission to emit terms of a certain form (where the F variable has a domain given by the *resolution_factor* predicate in this case).

My mini-game play space has an overall topology that is either toroidal (like *Pac-Man*), spherical (strange but nonetheless distinct), or flat (resulting in "falling off" the edges of the world) if not otherwise specified. The generator outputs between zero and one instances of the *space_topology* predicate (as dictated by numerical bounds on the choice rule) using a scheme similar to the above with symbols to name the various topologies instead of numbers.

Related to the space, but not an element of the game mechanics, the game may utilize (or not) any of two background layer display algorithms (twinkling stars or dotted grid lines). The generator code for this aspect introduces dependence between a generated element and a flag that might be toggled via player exploration in future VF prototypes:

```
tech(backgrounds).
{background(L) :background_layer(L)} :- tech(backgrounds).
```

The play space is primarily populated by characters, identified by their color. The generator internally selects an active subset of colors from a larger set, and whether a color is active or not is used as a logical precondition for the rest of the character-related generator rules.

Every active character color is assigned a unique movement model (determining their response to keyboard input and, eventually, autonomous behaviors). The VF prototype includes *Asteroids*, *Pac-Man*, and *Rogue* inspired movement models. The generator code to support this combines quantification over multiple variables to produce a one-to-one mapping specific to the active characters is shown below:

```
1 { agent_movement(C,M) :movement_model(M) } 1 :-
       active_color(C), color(C).
```

In addition to a movement model, characters have another required property called their spawn model that dictates whether exactly one of them should spawn at the start of the game or a larger, random number. The generator code has identical structure to that of the movement model.

Much more interesting is the character–character collision effects table. This table, which describes only active characters, produces *agent_collide_effect*, a predicate which is not only used by the game engine during mini-game execution, but also by the generator itself as I will describe later. The table describes which collision resolution behavior should be applied to the character of the first color if it hits another of the second color. There are *kill* and *bounce* options with a default of simply passing through on collision. In future versions of VF, the set of collision resolution behaviors that are considered will be conditioned on player exploration as well.

Beyond the basic space and characters, if the generator has the *obstacles* exploration flag enabled, the game will consider any combination of three obstacle placement patterns: an encircling wall, stick-like scattered walls, or isolated blocks with slight rotations (all three are active in Figure 14.1). The selection of these algorithms is described with the same schema used to select background art layers. If obstacles are enabled, an optional collision resolution behavior is selected for each character and encoded as the *obstacle_collide_effect* predicate (obstacles themselves cannot be "killed"). In this case, not just the size but also the very existence of a table in the ruleset is conditioned on other generated outputs (the active set of colors).

In order to make games in this space playable, I need to assign the player control of one of the characters. This assignment is based on color; if the player controls a character with the "many" spawn model, then they will only control one such character and the rest will perform their default behavior.

At this point, games include a player who can fly a character around a variously configured world, bumping into other characters and obstacles to trigger effects out of tables, but there is still no goal to my mini-games. The final element of the mini-game description gives it one. The *goal* predicate must have a single instance in each game design to enable victory-condition checking. Its form may either be *goal(kill_all(Color))* which monitors for when all characters of a given color are killed or *goal(escape)* which monitors for when the player character reaches the world boundary (which only exists in the flat space topology).

This final output illustrates a clear representational flexibility that my symbolic representation has over fixed-length parameter vector representations: some of my game goals are parameterized by active character colors, while others such as *escape* are not. There is no architectural penalty for mixed structures such as this. The *escape* goal, in particular, is additionally forbidden in games utilizing the encircling wall obstacle generator (from which it is impossible to escape). This is an example of capturing special-case knowledge in the generator extracted from experience playtesting broken games.

The complete generated ruleset for the "kill all the red guys" game shown in Figure 14.1 is represented by these logical statements produced by the generator:

```
space_resolution(32,24).
space_topology(spherical).
background(grids; stars).
active_agent(red; yellow; white; cyan).
agent_movement(red,asteroids; white,asteroids;
               yellow,roguelike; cyan,pacman).
agent_population(red,many; white,singleton;
                 yellow,singleton; cyan,many).
```

```
agent_collide_effect(red,white,kill;
                     cyan,yellow,kill).
player_agent(white).
obstacle_distribution(enclosure; random_walls;
                      random_blocks).
obstacle_collide_effect(red,kill; white,kill).
goal(kill_all(red)).
```

14.5 Zooming in on Games of Interest

While the exclusion of known problematic interactions between mechanics is something that could be encoded directly into the preconditions in the body of the choice rules used in the generator, there are many occasions in which I would like to temporarily scope down my generative space without disturbing any existing logical formulae. The mechanism of integrity constraints in ASP provides exactly this functionality.

I can use integrity constraints to zoom-in on a subspace of games in which I am interested via several methods, but the simplest is to simply require that certain ruleset elements be present in all answer sets I see. I can simply append an integrity constraint rejecting the absence of the required configuration. If I wanted to tweak the implementation of the *Asteroids* motion model in VFs game engine, I might add this collection of constraints to always give myself control over a red character in a primitive field of asteroids to navigate, regardless of other mechanics:

```
:- not player_agent(red).
:- not character_movement(red, asteroids).
:- not use_obstacles(encircling).
```

Another use of integrity constraints to sculpt the generative space is to reject co-occurrence of mechanics known to interact poorly. The special-case knowledge for the *escape* game goal is encoded with the single integrity constraint:

```
:- goal(escape), obstacle_distribution(enclosure).
```

Integrity constraints need not only operate on the same elements that are exported by the generator, they may involve complex deduction. The following snippet of code (slightly condensed from the version active in the VF prototype) zooms in on rulesets in which the game is reasonably winnable by indirectly pushing characters into each other to achieve the stated goal:

```
pushes(A,B) :-
       on_collide(A,B,bounce),
       on_collide(B,A,bounce).

kills(A,B) :- on_collide(A,B,kill).

indirectly_pushes(A,B) :- pushes(A,B).
```

```
indirectly_pushes(A,C) :- pushes(A,B), indirectly_pushes(B,C).

winnable_via(indirect_push_kill(A,C)) :-
       indirectly_pushes(A,B), kills(B,C).

compute {
       player_agent(A), goal(kill_all(B)),
       winnable_via(indirect_push_kill(A,B)) }.
```

In the above example, I have shown an elementary attempt at *engineering emergent gameplay*. As new mechanics are added to VF's design space and the *winnable_via* predicate is augmented with a simple complexity metric, it becomes possible to write single-line integrity constraints which translate to statements akin to "only show me rulesets for games in which a reasonable plan for victory involves an indirect chain of at least 5 steps utilizing at least 3 different low-level interaction types." Games complex enough to satisfy this constraint would likely come from a space that also includes many broken games. Fortunately, additional integrity constraints may easily be added to carve away such failure cases as they arise in playtesting. It is this use of additional deductive rules to capture complex relationships between low-level mechanics that demonstrates the power of using a declarative representation over attempting to capture the equivalent design space with a custom generation procedure.

14.6 Generating Playable Mini-games

At this point I have described how to create, enumerate, and expressively sculpt the generative spaces of game rulesets. However, I have yet to show how to transform such sets of assertions about how a game should work into functioning games that operate as the generator designs them. In this subsection I will describe the concrete software components that bring ASP-based ruleset generation into contact with the player in our prototype of VF: the game generator and the game engine.

14.6.1 Game Generator

My game (ruleset) generator is manifest in two distinct parts. The first is the logical definition of a design space using an answer set program as described above, and the second is the software we use to enumerate games in the design space. I have adopted the freely available LPARSE and SMODELS[7] tools that, respectively, translate first-order logic programs into simplified, grounded logic programs and solve for the desired number of answer sets, outputting each as it is found.

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

To surface the functionality provided by these highly obscure (from a game programming perspective) command-line tools, I created a minimal web-service wrapper that allows any HTTP-capable program to request a stream of answer sets to a given answer set program.[8] This wrapper allows me to be much more flexible about the kind of game engine I use to consume the generated game designs.

The result of organizing my generator in this way is that the designer of the generative space does not need to think about (nor necessarily understand) the underlying generation algorithm. Indeed, different solvers that consume LPARSE groundings may employ radically different algorithms while being indistinguishable at the level of answer set enumeration.

14.6.2 Game Engine

Realizing the outputs of my game generator in the glowing vector-art aesthetic (and supporting low-level mechanics such as movement with momentum and collision detection/resolution) requires the use of a game engine. I built my game engine using the Flash game library Flixel[9] as a base. To this base, I added skeletal support for the elements I knew the ruleset generator would like to instantiate: abstractly depicted characters which can roam about, bumping into each other and obstacles, victory condition templates, basic level generation algorithms, and background image generation.

At each major design decision that would normally be hard-coded in a particular game (such as how big the game world is, which character the player controls, etc.) I added code to consult a configuration object (to be provided by the generator). Though making an engine that supports many possible games is significantly more difficult than making a particular game in isolation, the task is similar in complexity to the integration of a scripting language[10] which many complex games already possess to ease the development process.

The general flow of my prototype works as follows. On startup, the engine sends the internally stored AnsProlog program to my ASP web-service and begins streaming in solutions over the network. The player can randomly sample mini-game rulesets, given a basic textual preview of the game (describing what they control, the goal of the game, and other details selected by the generator). Upon selecting a mini-game, play begins. The mini-game's rules are fixed across restarts triggered by

[8] `https://github.com/rndmcnlly/aspaas`

[9] `http://flixel.org/`

[10] Unforseen at this point in my research, exposing sufficient customizability for game mechanics through this scripting interface quickly becomes the bottleneck for similar projects.

when the player character is "killed." The player rejoins the initial game selection screen upon victory or intentionally abandoning a difficult (or broken) game.

In the design of VF beyond the prototype, I envision performance in mini-games linked to a resource which can be spent to either unlock new reaches of an initially small mini-game design space or buy constraints which enforce interesting patterns. Through incrementally refining the possibilities open to the generator the player can slowly come to understand the interaction between the various modular mechanics set into the VF universe. Each new game design element the player unlocks results in new `tech(T)` assertions being simply appended to the text of the internal logic program, which, in concert with existing integrity constraints and preconditions, means the design space of mini-games can be dramatically reshaped during play at a scale unmatched by any other game.

14.7 Discussion

14.7.1 Coupling between Generator and Engine

While the definition of the design space is strongly separated from the means of sampling the space in our approach, there is a strong coupling between the content generator and the game engine which consumes the content. This mandatory coupling is not problematic if one considers the generator and the engine to be two parts of a single program. In the VF game prototype, the game engine and ASP code used for ruleset generation are combined into a single binary that runs in the player's browser while the (project agnostic) answer set solver runs independently on remote server.

14.7.2 Tradeoffs in Levels of Abstraction

The relative expressiveness of the generator compared to the engine depends on the level of abstraction used by the logical terms with which they communicate. As the generator takes on more responsibility for defining the mechanics of games (e.g. working with lower-level terms to *implement* movement models instead of simply *instantiating* them), it becomes even more critical that one be able to sculpt the design space to avoid the swath of well-formed yet meaningless or broken (in terms of gameplay) constructions which a grammar might admit.

At the low-level extreme of generating the equivalent of machine instructions, clearly an astronomically tiny fraction of such "rulesets" would represent valid games and we would have a hard time writing down constraints that have any useful effect. Meanwhile, at the high-level extreme

of parameterizing a game by only a few configuration values, the space of games (even if all were guaranteed to be valid) would be uninteresting for a player to explore. The challenge, which I have taken but a single step towards, is to work at the lowest, code-like level possible (enabling the richest variety) while not losing control of the design space and consequently asking the player to play a structurally broken game.

14.7.3 Evaluation

Towards gauging my level of success in this project, I am most interested in expressivity of the generation approach and the constraints it employs to shape generative spaces for the application of procedural content generation.

In my experience with the VF prototype, I found a 100-source-line ruleset to be abundantly generative, generating rulesets which exposed legitimate bugs in my game engine and raising important game design issues (such as how the momentum of *Asteroids*-style characters should change in collision with *Rogue*-like characters). With integrity constraints, it was both easy to both zoom in on failure cases for testing and to forbid the occurrence of situations I had not yet resolved.

In terms of expressivity of constraints, recall the space of indirect-push-kill games (described previously). This scenario demonstrates how I can encode additional knowledge about a game design space (such as how to produce a high-level plan to win games in it) into the generator itself. Knowing that high-level descriptions such as "*winnable_via(indirect_push_kill(red,blue))*" are present in the games definitions allows me to write single-line constraints requiring or forbidding high-level patterns. Such in-line self-analysis of games could also be used to prepare a small manual for each generated game. EGGG [161] was able to produce documentation for the game it generated, also by virtue of having so much design knowledge baked into the generator.

Evolutionary methods such as Togelius' system aim to produce rulesets optimized by some metric (with Browne's LUDI system [22] even striking commercial success in autonomously designing *Yavalath*[11]). However, in PCG, there is an inherent demand for sizable spaces with significant, player-visible variety. My declarative approach allows easy manipulation of generative spaces of arbitrary size, while evolutionary approaches only maintain a fixed-size population (on the order of tens or hundreds) during their search for a single, optimal individual. While both approaches are highly declarative in their own sense, I believe my approach is distinctly more space-oriented, and is therefore better suited for PCG.

[11] `http://www.cameronius.com/games/yavalath/`

As a final form of evaluation, I can look at our ruleset generator as a creative system (though it was not designed as one). In the field of computational creativity, a standard means of evaluating an artifact generator are to look for novelty and value in the artifacts it creates(Pease, Winterstein, and Colton 2001). One instance of novelty I experienced was a seemingly unwinnable game. After some effort I realized the game could actually be won by indirectly pushing an intermediate character into the characters it was my goal to kill. Excited by this occurrence which was both unexpected (novel) and fun (valuable), I quickly devised the indirect kill detection logic described previously to zoom in on other games of this variety, transforming the generative space.

14.8 Conclusion

In seeking the technology to support a novel game design, I have developed a new content-generation approach and applied it to the challenging domain of game ruleset generation, producing a large space of playable mini-games. The flexible representations afforded by ASP allow concise-yet-powerful modifications to this design space. This representation schema has also prepared us for a more serious attempt at automating the larger processes in game design.

Chapter 15

Refraction Tools

Thus far, my applied systems have been applied to problems of my own design. It is natural to ask whether my program of modeling design spaces in ASP can survive contact with pre-existing problems coming from real, ongoing, complex game design projects. In this chapter, I report on the experience of being embedded in a larger game design team at the Center for Game Science at the University of Washington working through how to produce a future, player-adapted version of their game that requires deep, play-time design automation.

Some problems in procedural content generation for games involve hard constraints (e.g. that a generated puzzle is necessarily solvable). Common techniques for generator design lack a way to specify crisp (yes or no) constraints on what counts as a valid content artifact and guarantee these constraints are satisfied in the generator's output. In this section I present two independent implementations of three diverse level design automation tools for the popular online educational game *Refraction*. All of the systems guarantee key properties of their output. Applying a constraint-focused generator design perspective in depth, I found that even emergent aesthetic style properties were straightforward to directly control. Our results with *Refraction* provide further concrete evidence for the claim that the expressive power of constraints and the ease with which they can be incorporated into suitably designed generative processes makes them a powerful tool for producing reliably-controllable generators for game content.

15.1 Introduction

In procedural content generation (PCG) for games, the topic of what guarantees a generator makes about its output often goes unaddressed. When seeking to apply PCG to a future, player-adapting version of the popular online educational game *Refraction*,[1] we encountered a strong need for assurances regarding generated content. *Refraction* is a Flash puzzle game in which players arrange devices on a grid to construct networks of laser beams. By requiring the player to construct beams of varying power levels, the game aims to teach mathematical skills, such as proportional reasoning, while exercising spatial problem solving abilities.

With regard to guarantees, we require that puzzles generated for a given player be not just solvable, but solvable under conditions appropriate for that player's progress: with precisely dictated size, complexity, and mathematical and spatial skills (such as being able to understand fraction simplification or use three left-turns to make a right-turn). Further, we want to prescribe aesthetics of the visual composition to continue the standards of control used in the creation of the game's current hand-authored puzzle sequence. While some of these requirements flow from the game's educational goals, it should be clear that methods for addressing these requirements will have use well outside of the realm of educational games.

Content generators are often designed as either directly constructive processes or generate-and-test systems [220]. Constructive processes guarantee some properties of their outputs *by construction*, however other properties can only be enforced by carefully hand-picking from sampled outputs. Generate-and-test systems attempt to automate this process, however they are often realized as open-ended optimization processes (such as genetic algorithms) that still require human intervention to decide precisely when generated artifacts are sufficiently fit for use in gameplay. Crisp thresholds (sharp boundaries defining what content is acceptable or not) are not defined in the problem formulation used by these systems because picking a single acceptance threshold on artifacts' computed fitness (usually a single scalar value) is difficult to impossible.

Ideally, we would have many examples of generators with crisply defined output spaces to draw from when designing new systems. These example generators should handle the full complexity of well-known games to provide realistic references for familiar problems. Towards generality, they should demonstrate direct control over a wide variety of features of interest (e.g. low-level structural validity, user-specified control parame-

[1] As of November 2012, Refraction had over 500,000 plays at `http://kongregate.com/games/GameScience/refraction`

ters, and high-level aesthetic concerns). Schanda and Brain's DIORAMA,[2] a highly controllable map generator for *Warzone 2100* (Pumpkin Studios 1999), is one such system in the domain of real-time strategy games, but it has only been briefly reviewed in the literature [195].

In this chapter I describe six systems that guarantee key output properties. I consider three diverse level design automation problems in *Refraction*: generation of high-level missions (under educational and gameplay constraints), transforming missions into spatially realized puzzles (which must be solvable in particular ways), and producing alternative solutions to pre-existing puzzles (allowing us to probe the requirements of our hand-made levels and those generated with different goals). For each problem, I review two system implementations. Our initial implementations were based on either constructive techniques or familiar complete-search techniques (bounded depth-first search). Exploring recently proposed techniques from my own research program [195], our subsequent implementations used answer set programming (ASP), a declarative programming paradigm that targets difficult combinatorial search problems with state-of-the-art algorithms.

Because each of our systems are *complete* by design (in that they always produce content conforming to input requirements upon termination when it is logically possible to do so), we focus our analysis of these systems on their uncontrolled aspects: code size, running times, accidental style features, and authoring affordances. We found that our ASP-based tools often produced example outputs which directly drove refinement of our problem formulations, causing us to better understand the deeper issues of puzzle design in games like *Refraction*. Because we were able to rapidly iterate on the specification of constraints, our later ASP-based tools are significantly more controllable and thus more useful in the face of our design automation problems.

The primary finding of this case study is the unexpected expressive power that resulted from the in-depth application of a constraint-focused generator design perspective. My results provide further evidence for the claim that declarative languages with first-class constraints such as those available in ASP are powerful tools for producing *expressively constrainable* generators, systems that accept a wide range of hard constraints as part of their input while providing theoretical guarantees for the production of that content if it is feasible. By treating aesthetic failures (e.g. poor compositional balance of a puzzle) as equivalent to gameplay failures (e.g. an unsolvable puzzle), I not only raise the stakes on a question Togelius et al. [220] identify as a major research challenge in search-based PCG (*How can we avoid pathological failures?*), but provide multiple real-world

[2] `http://warzone2100.org.uk/about.html`

examples as answers.

15.2 Related Work

In previous research into automatically generating puzzles, there is often a search algorithm in the core of systems that works to separate broken or uninteresting puzzles from those that are well formed and elegant. Colton [35] identified puzzle generation as a creative task, requiring a designer to produce an artifact (the puzzle) that would cause the solver (the player) to make a personal discovery (finding an interesting solution). Much of the search in Colton's system is dedicated to ensuring that the puzzle's intended solution is derivable from the clues provided and that are no simpler solutions.

Focusing specifically on the problem of controlling the complexity of a puzzle's simplest solution, Ashlock [5] demonstrated an evolutionary algorithm for generating Chromatic and Chess mazes (both are spatial puzzles) with preferentially long shortest-path solutions. Oranchak's Shinro puzzle generator [160] also used an evolutionary algorithm. However, instead of optimizing solution length, Oranchak's system optimized a metric that balanced structural validity (which is non-trivial for Shinro, involving global agreement of puzzle clues) with closeness to a set of user-specified parameters that expressed a target number of pieces and solution steps. Though these measures of a puzzle's fitness provided informative evolutionary pressure to guide the search process in the direction of desirable puzzles, they alone do not guarantee eventual generation of suitable puzzles by these systems (an inherent property of metaheuristic optimization techniques [229] that also applies to the feasibility constraints of FI-2Pop [110]).

Gebser's Sudoku puzzle generator [81], by contrast, provides strict theoretical guarantees. This 38 source-line generator, based on answer set programming, defines the structural properties of desired puzzles (including the minimality of clue sets with respect to ensuring a unique solution) and then uses an off-the-shelf answer set solver to deterministically enumerate all (and only) those puzzles with the required properties.

A number of related systems have explored content generation with the application of hard constraints. To our knowledge DIORAMA[3] is the only one to enforce these constraints through search without the need to have defined a new search algorithm. Tanagra [202] (a platformer level generator that uses an off-the-shelf numerical constraint solver to enforce reachability for all of a map's platforms), SketchaWorld [191] (a declarative 3D

[3] `http://warzone2100.org.uk/`

modeling tool for terrains that foregrounds constraints in its user interface), and the layout solver described by Tutenel et al. [225] (rule-based system for arranging building-interior scenes under layout constraints) are highly relevant projects whose applicability to a deployed game remains to be seen.

15.3 Answer Set Programming

While a best-first heuristic search algorithm such as A* ("A-star") is likely to be familiar to game developers, the search algorithms[4] underlying some of the core tools in our systems are less so. We have made extensive use of answer set programming (ASP), a logic programming paradigm that borrows syntax from Prolog and search algorithms from solutions to the Boolean satisfiability (SAT) problem [7].

Like regular expressions for string matching or structured query languages (SQL) for retrieval from databases, AnsProlog (the common language for answer set solving systems) is a highly declarative language for solving combinatorial search and optimization problems, not a general-purpose programming language such as C++ or Java. Most ASP systems work by translating the programmer-provided problem definition into a low-level, domain-independent representation through the process of grounding (also called instantiation). Then a high-performance combinatorial search algorithm solves the ground problem.

One of the goals of ASP is to allow programmers to construct solutions to complex search problems without the need to develop and maintain advanced combinatorial search infrastructure. The imperative details of the answer set solver's underlying algorithm (which are easily reconfigured with command-line settings) are unimportant so long as suitable outputs are produced in an acceptable amount of time.

15.3.1 ASP for PCG

In a recent journal article [195], I described the general approach of applying ASP to PCG problems, offering a tutorial introduction to answer set programming and a review of four existing applications using the technique.

Regarding the software engineering practices around using ASP for PCG, I noted that properties of artifacts produced during the development of a generator will often inspire changes to the design space defi-

[4]The answer set solver we used employs conflict-driven nogood learning (CDNL), a state-of-the-art, complete, backtracking, heuristic search algorithm *very* loosely inspired by the Davis-Putnam algorithm for Boolean satisfiability [79].

nition, motivating the need for flexible generation systems which admit sculpting the space of outputs without an overall redesign of the generator. Some of these changes involve "zooming in" on content exhibiting patterns of interest or rejecting content with easily describable flaws.

In contrast with modern multi-paradigm languages (e.g. Python), the structure of answer set programs is relatively simple. These logic programs contain two constructs: facts and rules. Facts are statements (akin to data literals or documents in a data language like XML) that can be used to describe bulk configuration or the properties of an input problem instance. Three types of rules control the production of new facts. Choice rules specify how to *guess* a description of a candidate solution. Deductive (Prolog-like) rules specify how to *deduce* the properties of a guessed solution. Finally, integrity constraints *forbid* solutions exhibiting or not exhibiting certain deduced properties.

For the purposes of high-level design, the programmer can imagine the answer set solver runs a generate-and-test process, repeatedly *guessing* solution candidates, *deducing* their properties, and then *testing* if they should be forbidden. In actuality, solvers will propagate constraints forwards and backwards through the rules in a non-trivial manner that further includes learning of new constraints (called nogoods) on the fly from dead-ends discovered by the live search process. Further, whole spaces of partial solutions that exhibit forbidden substructures are often eliminated before any are completely assembled. When building a content generator with ASP, the programmer focuses almost exclusively on how the content design space is declaratively defined, treating the solver as an uninteresting black box.

15.4 Refraction Puzzle Design

The premise of *Refraction* is that the player must arrange devices to form a network of laser beams that will restore power to animals stranded in underpowered spaceships. The power of laser sources and the power required by targets are mismatched, so the player must split and recombine beams to provide power in the correct proportion, indicated by a fraction. Figure 15.1 shows an example puzzle and solution.

When play begins, the 10-by-10 grid is clear except for laser sources, laser targets (animals in spaceships), and blockers (asteroids or other space debris). The position and orientation of these pieces are fixed. Additionally, sources and targets are annotated with the fractional power that they emit or require to be satisfied.

All player-movable pieces (beam manipulating devices with a fixed rotation) start in the panel on the right. There are four broad piece

Figure 15.1: Screenshot of gameplay in *Refraction* depicting a puzzle solution involving benders, splitters, combiners, and an expander. This puzzle is used as running example in the rest of this chapter.

categories. Benders simply apply a 90-degree deflection to a beam without changing power. Splitters consume one input beam and produce two beams at one half of the input power (or three beams at one-third power depending on the number of outputs on the splitter). Combiners (which come in two-input and three-input varieties) produce an output beam with a power that is the sum of all of the input beams (but only if all inputs are filled and the input fractions share the same denominator). Expanders (which do not deflect) facilitate combining of unlike fractions by multiplying the numerator and denominator by a common factor. Expanders are available with factors of 2, 3, and 5 such that applying a 3-expander to the fraction 1/2 results in the (unreduced) fraction 3/6.

Not all of the pieces provided to a player are necessary to form a solution. Most puzzles will include extra pieces that are intended to distract the player. Distractors are not always useless because they may be used to construct alternate (often more elaborate than necessary) solutions.

The designer's challenge is to produce a progression of puzzles that incrementally introduces the player to the spatial and mathematical reasoning challenges of the game and eventually prepares them for the game's full complexity, requiring fluent use of many types of pieces simultaneously. In the context of this progression, it is clear that what makes a level acceptable (as the product of generation) depends far more on its relevance to the player's progress than any simple measure of its solution length or the like.

The challenge for deployable level design automation in *Refraction* is to produce a generator that can recreate levels in the style and complexity of each point in the current, hand-authored linear progression. Even this requires generators with highly controllable output sufficient to express what makes a puzzle appropriate for the very beginning, the very end, or, say, the first level that introduces expanders. Beyond this, we are interested in enabling non-linear, player-specific concept and difficulty progressions.

15.5 Problem Formalization

To make the challenge of level design automation for *Refraction* more concrete, we have broken it down into three artifact generation problems. To structure our puzzle generator, we have adopted Dormans & Bakkes' [57] distinction between missions and spaces. A mission is a logical order of the goals a player must accomplish to complete the level, and a space is the actual physical layout of the level. Our first two problems are concerned with producing missions for *Refraction* and subsequently embedding those missions in a puzzle grid. The final problem is concerned

with seeking alternative solutions to existing puzzle designs, regardless of the mission for which it was originally designed.

15.5.1 Mission Generation

The intent of the mission generation problem is to capture the high-level design concerns of a *Refraction* puzzle: Which pieces are active? How big is the imagined solution? What level of mathematical knowledge will be involved? Because fractions are integral to the game's educational goals, mission generation includes working out which fractions should be constructed and how the construction might proceed.

The primary input to our mission generators is a set of mathematical expressions that the player should construct during play. The set { (1/2)+ (1/4), (1/4) + (1/4), (((1/2)/2)/2) } suggests the need for adding twice (once with the use of an expander to build a common denominator) and repeated splitting by half. The mission generator is also given a target number of blockers (24), benders (7), and distractor pieces (7) to modulate difficulty. These were the inputs to the mission generation process that eventually resulted in the puzzle and solution shown in Figure 15.1 and used as a running example in the rest of this chapter.

In the ASP-based mission generator, an upper bound and optional lower bound on piece counts are specified for all piece types, along with style constraints affecting the presence or absence of arbitrary mission subgraphs. These constraints were not expressible in the initial mission generator (described later) without a major redesign.

The output of mission generation is an annotated directed acyclic graph (DAG) where there is a node for every piece in the imagined puzzle and an edge for every solution-critical laser beam connecting pieces. Nodes describe a piece's mathematical type (such as 2-splitter or 5-expander) but not its spatial type (such as having an input from the west and an output to the north). Source and target nodes are labeled with the fraction power they emit or require. Figure 15.2 shows an example mission satisfying the constraints above.

15.5.2 Grid Embedding

In the grid-embedding problem, the intent is to realize a puzzle with sufficient detail to be played in the live game. That is, embedding resolves the spatial concerns ignored in the mission generation problem. While the current version of *Refraction* is played on a discrete, rectilinear, two-dimensional grid, a version for play on, say, continuous spaces, hex maps, or three-dimensional grids would not disrupt our high-level problem for-

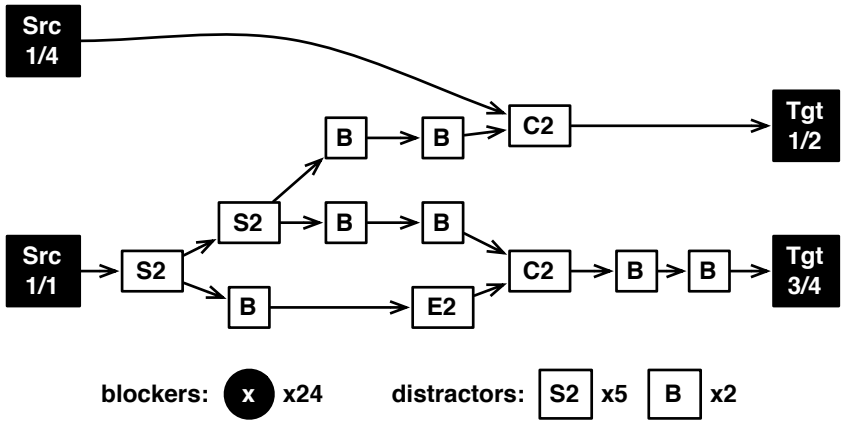

Figure 15.2: Mission DAG containing several 2-splitters (S2), benders (B), 2-combiners (C2), and a 2-expander (E2). Several blockers (x) and distracting pieces will also be present in any spatial realization of this mission.

mulation or solution methods.[5]

The primary input to the embedding problem is the same as the output of the mission generation problem. These mission DAGs may come from either of our mission generation systems, extraction from the hand-designed levels, or original human authoring effort. The ASP-based embedder also accepts additional style constraints describing spatial properties such as symmetry, compositional balance, and piece spacing.

The output of grid embedding is the (x/y) position and (north/-south/east/west) input/output port configuration of all pieces such that one example solution is constructed that realizes the input mission DAG. Figure 15.3 shows an alternate embedding, to the one depicted in Figure 15.1, for the mission shown in Figure 15.2. Generally, there may be astronomically many valid embeddings of a mission, but we are concerned with producing only a single one.

15.5.3 Puzzle Solving

Finally, the intent of the puzzle-solving problem is to simply construct alternative reference solutions (at the spatial grid level). In addition to revealing which pieces and patterns are required in a solutions to a puzzle,

[5]Indeed, in subsequent exploratory design, I have realized variants on the ASP-based grid embedder for each of these exotic cases. For the time being, the plan is to stick with the plain and simple 2D grid in the public version of the game.

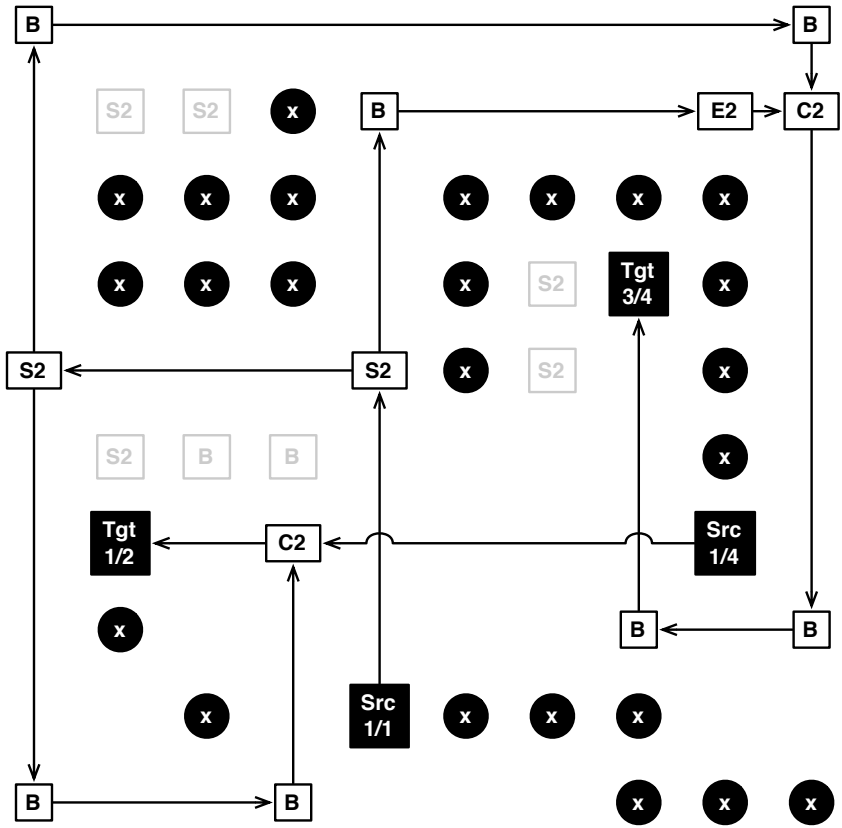

Figure 15.3: Embedding of the mission DAG from Figure 15.2 into *Refraction*'s 10-by-10 spatial grid under no special style constraints.

a fully automated puzzle solver can be used to provide feedback to players by telling them if their solution-under-construction can be extended to a complete solution without removing any piece already placed. This same type of partial-solution feasibility checking can be used in offline analysis of recorded game data to track how much time different players spend in infeasible regions of the game's state space.

The input to the puzzle-solving problem is a complete definition of available pieces and their mathematical and spatial configurations (excepting the positions of player-placed pieces, of course). The ASP-based puzzle solver also takes additional style constraints as input: requirements to use or avoid a certain piece or grid cell, to construct or not construct certain sub-networks of laser flow, and so on.

The output of puzzle solving is simply the position of every piece such that the resulting configuration satisfies all laser targets or an assertion that the puzzle has no solution under the additional constraints.

15.6 System Descriptions

In this section I describe the two implementations of each of our three level design automation tools. I review them in the order they were developed to help convey the idea that each was a legitimate best-effort research solution to the stated problems given the knowledge at hand. Each system was created with the intent of use in a production setting, not specifically for the purposes of comparison.

15.6.1 Feed-Forward Mission Generation

Addressing the first problem, that of generating high-level missions, our initial implementation adopted a constructive approach consisting of a seven-step pipeline:

1. **Expression Translation:** Mission graph fragments corresponding to the required input expression trees were generated though straightforward compiler techniques. For example, $\{(1/4) + (1/4)\}$ becomes a three-node graph with one 2-combiner node (to represent the "+") linked by edges labeled 1/4 from two untyped nodes.

2. **Opportunistic Combination:** In a randomized fashion, nodes are unified so that the output of one required expression could be the input to another. This process proceeds, avoiding cycle creation, until exhaustion.

3. **Target Completion:** To motivate (though admittedly not to guarantee) the player to construct the imagined graph so far, target

nodes are added to consume all laser outputs that are not consumed by another piece already.

4. **Expander Insertion:** Expanders are inserted so that the inputs to combiners all have the same denominator.

5. **Bender Insertion:** The requested number of bender pieces are randomly inserted into the graph on paths between sources and targets.

6. **Distractor Selection:** A number of randomly typed pieces are also added to the graph without adding edges.

7. **Obstacle Insertion:** Similarly to distractors, the required number of disconnected blocker pieces are added.

By construction, generated mission DAGs will describe feasible solutions to the mission generation problem (at least at the network level) that involve the required mathematical construction and the requested number of blockers, benders, and distractors. This system serves as an example how to guarantee certain properties of outputs through bespoke algorithm design. Note, however, that the above algorithm is carefully adapted to just those design requirements known at the time of its creation.

15.6.2 Grid Embedding with DFS

The problem of grid embedding immediately appeared to us as a highly constrained search problem (unlikely to be fruitfully addressed with feed-forward or simple generate-and-test approaches). While the problem somewhat resembles the place-and-route problem from electronic design automation (EDA) [211], the particular mechanics of *Refraction* made a hand-rolled complete-search implementation seem the most approachable solution at the time.

Our randomized depth-first search (DFS[6]) algorithm was configured as follows:

- **States:** list of embedded pieces and their positions; graph of remaining pieces to be embedded; list of outgoing beams with their directions

[6]Note that DFS is *complete* for search spaces with bounded diameter. In the grid-embedding problem, no paths have a length that exceeds the total number of pieces in a puzzle.

- **Successor Function:** place a piece with no un-embedded inputs from the mission somewhere along the beams to which it must connect and assign its input directions as necessary; if the piece has output directions, then assign them randomly at this time according to the piece's type (ensuring benders deflect the laser, etc.)
- **Goal:** no pieces remain to be embedded

When our DFS terminates at a goal state, that state necessarily represents a valid embedding of the mission DAG with respect to *Refraction*'s rules. While this implementation is sufficient to solve the problem, we later back-ported the use of a geometric *restart policy* (a common technique for boosting combinatorial search [85]) from our ASP-based embedder, resulting in observed performance improvements of up to four orders of magnitude for realistic problems.

15.6.3 Puzzle Solving with DFS

Based on the initial success with DFS as an implementation strategy for complete and correct embedding, we decided to address the problem of producing reference solutions with this algorithm as well. Puzzle solving involves a similar spatial search to the embedding problem. However, in embedding, a plausible solution graph (the mission DAG with fully-resolved mathematical concerns) is given and the piece input/output port configurations are flexible (to be generated). In solving, port configurations are constrained as part of the input and no solution sketch is provided, dramatically complicating the search problem.

Our DFS for puzzle solving was configured as follows:

- **States:** list of pieces placed so far and their positions; list of outgoing beams with their direction and power
- **Successor Function:** select an unused piece that has an input port that can accept an existing, unconsumed outgoing beam and place it somewhere along that beam; decide if the new piece will produce new beams, and compute their direction and power
- **Goal:** the simplified sum of beam powers entering every target matches its required value

As before, correctness with respect to input requirements on successful termination is assured by well-known results for DFS. The geometric restart policy was also back-ported to the DFS-based puzzle solver after our experiments with ASP, yielding solutions for previously intractable puzzles.

15.6.4 Grid Embedding with ASP

Having assembled and tested the previous three systems and integrated them into a research version of the game, we identified the existing grid embedding system as the biggest bottleneck for runtime performance and expressive control in our plans for a player-adapting revision to the game. Seeking to replace this system with a simpler (towards better adaptability) and potentially faster implementation, we adopted the following organization for our ASP-based grid embedding system.

- **Choice Rules:**
  - Guess absolute (x/y) positions for pieces.
  - Guess port configurations based on piece type.
- **Deductive Rules:**
  - Deduce relative (north/south/east/west) positions from absolute positions.
  - Deduce free paths from relative positions.
  - Deduce realization of beams (embedding for mission edges) from paths and guessed port configurations.
  - Deduce presence of style patterns (compositional balance, symmetry of blockers, etc.).
- **Integrity Constraints:**
  - Forbid two pieces overlapping.
  - Forbid lack of embedding for mission edges.
  - Forbid illegal port configurations (benders must bend, expanders must not, etc.).
  - Forbid violation of style policy (reject if balance or symmetry not detected, etc.).

Correct enumeration of all and only those embeddings that are valid is assured by the correctness of the answer set solver. The source code for our ASP-based embedder did not contain any descriptions of search algorithms, only a declarative description of the search space, artifact analysis definitions, and goal conditions.

15.6.5 Mission Generation with ASP

Success in using the ASP-based embedder as a drop-in replacement for the previous embedder was enticing, so we next looked at replacing the mission generator with an ASP-based variant as well.

Thus far, the mission generator and grid embedder had been run with an overall generate-and-test architecture (because some mission DAGs are formally impossible to embed, such as those containing triangular undirected cycles). ASP held promise for the ability to run the mission generator and grid embedder as an integrated whole under constraints that jointly bound both phases of generation. With the goal of upgrading our embedder into a complete puzzle generator, our ASP-based mission generator was designed as follows.

- **Choice Rules:**
  - Guess which pieces will be active.
  - Guess power level for laser sources.
  - Guess presence of edges between pieces.
- **Deductive Rules:**
  - Deduce a piece's emitted laser power from the power of all edges into it (using a recursive definition).
  - Deduce simplified power for all targets.
  - Deduce set of pieces that are upstream of active targets.
  - Deduce the presence of mathematical or other graph properties (*half_plus_quarter*, *triple_bending*, etc.).
- **Integrity Constraints:**
  - Forbid directed edges above port limits (only one edge into a splitter, only one edge out of a combiner, etc.).
  - Forbid edges to nodes not on a path to a target.
  - Forbid presence or absence of particular mathematical and style patterns.

To simplify the AnsProlog definition of mission generation logic, we used an auxiliary answer set program to pre-compute a table of all ways of manipulating beams of different powers. Parts of this program (notably Euclid's algorithm used in fraction simplification) were expressed in Lua [97], an embedded scripting language made available for performing arbitrary transformations of logical terms with imperative code.

The final ASP-based mission generator can be run standalone, as a drop-in replacement for the previous mission generator, or it can be combined with the ASP-based grid embedder (by simply concatenating the source for the two programs) to form a monolithic puzzle generator.

15.6.6 Puzzle Solving with ASP

With the accumulated logical modeling experience of producing the mission generator and grid embedder, creating a styleable puzzle-solver using ASP was straightforward.

- **Choice Rules:**
  - Guess piece positions (the player's only responsibility).
- **Deductive Rules:**
  - Deduce relative positions from piece positions.
  - Deduce free paths from relative positions.
  - Deduce beam flow from free paths and port configurations.
  - Deduce emission fractions from beam flow (recursively).
  - Deduce target power from beam entrance.
  - Deduce presence of solution-style patterns.
- **Integrity Constraints:**
  - Forbid two pieces overlapping.
  - Forbid leaving targets unpowered.
  - Forbid incorrectly powering targets.
  - Forbid violation of style policy.

In the puzzle solver, a piece's effect on fractions was again expressed in Lua. However, no table was pre-computed because, when pieces are fully specified, a much smaller and puzzle-specific set of fractions is encountered.

In addition to correctly reporting whether a puzzle is solvable under stylistic restrictions (yes or no), this puzzle solver is radically reusable for online and offline analysis of partial solutions and queries as to whether a particular piece type or placement is essential to solving a puzzle (regardless of the originally imagined mission for that puzzle).

15.7 Analysis

Our comparative analysis of the two sets of level design automation tools breaks down into a quantitative comparison of the software systems and a qualitative comparison of the inputs and outputs for each tool.

15.7.1 Quantitative Comparisons

When comparing our respective generator implementations side-by-side, the most apparent difference is in their language distribution and code size. The original tools consist of a moderate amount of Java code whereas the newer tools consist of a much smaller amount of code in AnsProlog and Lua. For the three tools, here are the code size[7] distributions:

- **Mission Generation:** 1,145 Java lines — 194 AnsProlog, 38 Lua lines
- **Grid Embedding:** 987 Java lines — 75 AnsProlog,[8] no Lua lines
- **Puzzle Solving:** 988 Java lines — 83 AnsProlog, 61 Lua lines

In another numerical comparison, we looked at the running time of the tools on a fixed set of inputs derived from the example shown in Figure 15.1, a high-complexity puzzle in the class of late-game challenges that involve several pieces from every major piece category. Configuring the tools as equivalently as possible (applying no style constraints for the ASP-based tools), we averaged times[9] for 1,000 runs with different random seeds.

- **Mission Generation:**
  - Feed-forward algorithm: < 1 ms. total
  - ASP: < 1 ms. search (530 ms. grounding/preprocessing)
- **Grid Embedding:**
  - DFS: 650 ms. search (negligible overhead)

[7] We counted (non-blank, non-comment) source lines. These line counts are intended to record all code needed to support each tool assuming the others already existed (thus, shared utilities such as parsers and printers for the XML level file format are not counted).

[8] In a later refactoring of the ASP-based grid embedder, I re-expressed the problem in an n-dimensional space and added a "`#const dimensions=2.`" statement at the start of the program. Unexpectedly, the new program turned out smaller: just 27 AnsProlog lines.

[9] Experiments were performed on a 2006-era ("Dempsey") Intel Xeon CPU at 3.0 GHz. DFS times record search-time for the implementation with restarts.

  - ASP: 110 ms. search (630 ms. grounding/preprocessing)

- **Puzzle Solving:**
  - DFS: did not find solution within 1-hour timeouts
  - ASP: 350 ms. search (340 ms. grounding/preprocessing)

Generally, for difficult search problems, the advanced search algorithms of the answer set solver (Clingo 3.0.3 [76]) are better suited than the hand-rolled search in the original tools (dramatically more so before the post-hoc addition of a geometric restart policy). In all cases, the ASP-based solutions spend a significant amount of time on non-search activities (predominantly in propositional grounding). This grounding cost need only be paid when input requirements change if the grounded program is cached. Although the time required for grounding grows with problem size (our current encodings are cubic in player-controlled piece count), the fact that *Refraction* is played on a *constant*-bounded scale (with no more than a handful of player-controlled pieces) means this growth is a only theoretical curiosity so long as results are swiftly found at the scale of interest. That the solver's worst case running time is bounded only by an exponential in the size of the grounded problem is similarly uninteresting for realistic problems such as ours.

15.7.2 Qualitative Comparisons

Resulting from the *target completion* phase used in the constructive mission generator's pipeline, every mission DAG generated by this implementation describes a puzzle solution that does not involve laser wasting (the situation where a beam emitted by one of a piece's outputs goes unused in the solution). While (in concert with distractor pieces) players are often capable of wasting lasers if they choose, this style quirk of the original mission generator is an interesting secondary effect of attempting to motivate players to construct a particular network. Knowing of the laser wasting aversion in the original generator, the ASP-based mission generator intentionally includes hooks for requiring or forbidding the presence of laser wasting at the mission level. Similarly, the greedy nature of the *opportunistic combination* phase meant that mathematical expressions would only be realized in a subset of all feasible ways, prompting the subsequent development of arbitrary mission subgraph constraints that could control how expressions were realized more generally.

In the ASP-based embedder, we found that running the answer set solver with a fast-but-simplistic heuristic often led to embeddings that compacted all of a puzzles pieces into a cluster near the one corner of the grid. Crudely addressing this concern by telling the answer set solver to

use more randomness in its search or to use a different heuristic was not a reliable solution. While the compacted embedding was in strict conformance with the definition of the embedding problem, the result was aesthetically unacceptable. Analysis of these compacted solutions prompted us to define a basic model of compositional balance: after deducing active hemispheres (e.g. "laser flows through a piece in the west half of the puzzle"), we forbid configurations that leave any hemisphere inactive. The pattern of "abutting" (where laser flows between two immediately adjacent pieces without revealing the beam) could also be controlled to produce cleanly spaced reference solutions (meaning that players, particularly novices, would not be required to abut their pieces to complete the puzzle). Finally, the pattern of "crossing the beams" represented yet another feature of exactly-controllable output style in the ASP-based embedder. Figure 15.4 demonstrates driving the embedder in opposite stylistic directions.

When no style constraints are provided for the ASP-based embedder, the space of embeddings it might generate is identical to that of the DFS-based embedder running on the same inputs. Adding style constraints results in a generative space that is a strict subset of the unconstrained space. A similar situation applies to adding style constraints to the ASP-based puzzle solver.

While the original puzzle solver was primarily intended to produce a simple (yes or no) answer, the inclusion of style constraints in the input to our other tools naturally prompted our brainstorming of the potential alternate uses of an expressively constrainable puzzle solver mentioned previously. The ability to attach novel constrains to the input of our ASP-based tools, even when the existing definitions were not designed with these constraints in mind, represents a major qualitative difference between our two sets of tool implementations owing to ASP's architectural affordances.

15.8 Conclusion

I have described six examples of level design automation systems that make hard guarantees on key properties of their output. Covering three diverse level design automation challenges for *Refraction*, I have demonstrated that such guarantees can be made for the full complexity of a popular online game (further, one that was not designed around future design automation). In achieving this for an initial set of constraints, we made use of a constructive generator (which guarantees properties of its output by careful construction) and a familiar complete-search algorithm (DFS in a bounded space). To quickly produce generators for a wider va-

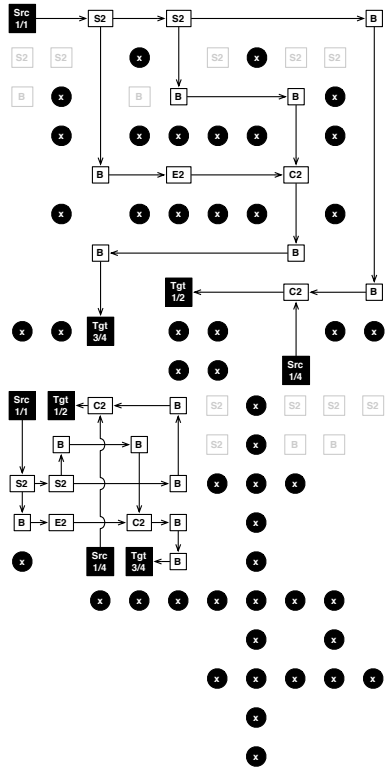

Figure 15.4: Two oppositely styled embeddings: the left exhibits compositional balance, blocker symmetry, beam spacing, and a lack of crossings while the right exhibits forced imbalance on both axes, asymmetry, piece abutting and multiple beam crossings.

riety of output guarantees, we explored emerging PCG results suggesting the use of answer set programming to declaratively capture exactly the design space we required.

These results suggest the developers of procedural content generators should not shy away from working with hard constraints. By applying a constraint-focused generator design perspective in depth, it is possible to not only produce reliably controlled generators with attractive performance measures, but to also come to better understand design automation problems through iterative exploration of constraints and generated output

Chapter 16

Rational Curiosity

The previous chapters have operated primarily from the perspective of design, giving designer-programmers access to new kinds of situational backtalk. However, exploratory game design can also be engaged in terms of creativity, of creating artifacts with novelty and value. In this chapter, I synthesize a knowledge-level description of creativity that puts the practices of exploratory game design to work in the service of artistic creativity.

Drawing on inspirations outside of traditional computational creativity domains, I describe a theoretical explanation of creativity in game design as a knowledge-seeking process. This process, based on the practices of human game designers and an extended analogy with creativity in science, is amenable to computational realization in the form of a discovery system. Further, the model of creativity it entails, creativity as the rational pursuit of curiosity, suggests a new perspective on existing artifact generation challenges and prompts a new mode of evaluation for creative agents (both human and machine).

16.1 Introduction

Paintings [116], melodies [42], and poems [90] are familiar domains for artifact generation in computational creativity (CC), and much established theory in the field is focused on evaluating such artifacts and the systems that produce them. In this chapter, I draw inspiration for a new understanding of creativity from the less familiar (but no less creative) domain of game design. In its full generality, game design overlaps visual art, music, and other areas where there are many existing results, but where it stands apart is in its unavoidably deep, active interaction with the audi-

ence: in gameplay. Crafting gameplay is the central focus of game design [73]. Play, however, is not an artifact to be generated directly. Instead, it is a result that emerges from the design of the formal rule system at the core of every game [176, chap. 12], a machine driven by external player actions.

Where, in visual art, we might judge the creativity (as *novelty* and *value*) of an artifact on the basis of the work's similarity to known pieces and its affective qualities [163], it is not so easy to make direct statements about the properties of the artifacts in game design. Desirable games are often celebrated for their *innovative gameplay* or the *fun* experiences they enable—these are properties of the artifact's interaction with the audience, not of the artifact itself. The focus on predominantly passive artifacts in CC, those which can be appreciated via direct inspection rather than through interactive execution, has masked what is obvious in game design: that, in any art form, the desirability of artifacts is in their relationship to their environment.

Armed with such an understanding, I seek a theoretical explanation of creativity in game design. Beyond the simpler application of established design knowledge, I want to understand the rarer experimentation that realizes wildly new forms of gameplay and deeply original player experiences for which there are yet no conventions or formulae. This theory should speak to both the artifacts and processes of game design, and do so in a way that meaningfully explains game design as done by humans as well as computational means. Towards capturing the richness of existing human design activity, I am most interested in a theory of *transformational* creativity [14] that explains how designers build new conceptual spaces of game designs and reshape them in response to feedback experiences observing play.

I introduce a new theoretical model that is amenable to computational realization that describes creative game design as a knowledge-seeking process (a kind of active learning). My broader contribution, creativity as the rational pursuit of curiosity, can provide an explanation of and suggest new questions for applications in traditional CC artifact generation domains.

In the following sections I review established game design practices, draw an analogy between game design and scientific discovery, review and apply Newell's concept of the knowledge level, and then introduce my model of creativity. Finally I conclude with a discussion of the implications of this theory for game design and the larger CC context.

16.2 Game Design Practices

In a standard text, Salen & Zimmerman [176, p. 168] introduce the "second-order" problem of game design bluntly:

> The goal of game design is meaningful play, but play is something that emerges from the functioning of the rules. As a game designer, you can never directly design play. You can only design the rules that give rise to it. Game designers create experience, but only indirectly.

Play includes the objective choices made by a player and the conditions achieved in the game, along with the player's subjective reactions and expectations. At this point, it is straightforward to adopt the first tenet of my theory of creative game design: **game designers are really designers of play**.

The idea of adopting an iterative, "playcentric" [73] design process, in which games are continually tested to better understand their emergent properties, is corroborated by others like Schell [179], who further describes the supreme importance of "listening" in the design process (being able to process feedback from the player's experience of candidate designs). Beyond initial conceptualization of a game idea and tuning and polish of the final product, the two most important practices of game design are prototyping and playtesting, both of which are intentionally focused on providing the designer with a better understanding of play.

Despite the ostensible purpose of game design being the production of complete, desirable games for play by end-users, the practices of playtesting and prototyping are centered on providing feedback to the designer. Through processes of exploratory game design, where several prototypes are created and playtested during a single project, the underlying goal is to build up sufficient skill and understanding to later produce the high-quality, final game artifact with the experienced gained. Such a self-affecting process is simultaneously an instance of Schön's "reflective practice" [182] and exactly the "iterative process of guesswork and evaluation" (often mistaken for generate-and-test) that McGraw and Hofstadter call the "central loop of creativity" [141].

Beneath the surface, the practices of game design are almost exclusively about the collection of design knowledge, knowledge regarding the relationship between the component elements of a game system and that game's potential execution in interaction with a player. Such design knowledge spans what design patterns to employ, how to assemble them, and why such an assembly will produce a certain play experience.

Existing design knowledge can be applied to realize familiar, well-understood play experiences, but creative game design demands a contin-

uous source of new design knowledge. Thus, the second tenet of my theory is this: **creative game design is about seeking design knowledge**.

16.3 An Analogy with Science

To expand on knowledge-seeking in game design, I want to draw an extended analogy between game design and science. Doing so will allow me to connect the creative activity in the game design process with the activity carried out by scientific discovery systems in CC.

For design generally, Dasgupta claims "design problem solving is a special instance of (and is indistinguishable from) the process of scientific discovery" [53, p. 353]. While Dasgupta focuses on explaining design activity specifically in terms of finding a confirmable theory that resolves a particular unexplained phenomena, my analogy is intentionally softer, to enable applications in a variety of CC domains that do not immediately appear as "design problem solving" domains.

16.3.1 Scientific Practices

Roughly, the scientific method is a closed loop with the following phases: A hypothesis is generated from a working theory, the hypothesis drives the design of an experiment (usually realized with a physical apparatus), data from executing this experiment is collected, and conclusions are drawn which can be integrated into the working theory. A scientist will design an experimental setup, despite already possessing a theory that makes prediction about the situation, precisely because there is some uncertainty about the result. This result, whether matching the prediction or not, should provide informative detail about the natural laws at play in the experiment's environment.

In my analogy, experiment design is prototyping; experiment execution and subsequent analysis is playtesting. The combination of the declarative knowledge of natural laws and the procedural knowledge of operating laboratory equipment is game design knowledge. Finally, the closed loop of the overall scientific method corresponds to iterative, exploratory game design.

Making the parallel clearer, Gingold and Hecker [84] talk about how gameplay prototypes should be informative, answer specific questions, and be falsifiable. In the philosophy of science, the notion of informative content (and its relation to falsifiability) guides the evaluation of theories and experimental designs.

In their capacity to design and execute informative experiments and produce coherent and illuminating explanations of the anomalous results,

scientists are clearly creative. In Colton's terms [37], we can easily *perceive* this creativity in the skill of precise experimental design, the appreciation of unexpected result in the context of a working theory, and the imagination of previously difficult to consider alternative theories and the invention of new instruments. By the analogy above, a scientist's kind of creativity can apply to the game designer as well, prompting our third tenet: **designers act as explorers in a science of play**, an "artificial science" in Simon's terms [190, chap. 5].

16.3.2 Automated Discovery

Though largely distinct from artifact generation, automating discovery in science and mathematics is an established CC tradition [120][124]. Within these systems it is common to find subprocesses that generate artifacts as part of the larger discovery process.

The GT system [69], an automated graph theorist, would periodically "doodle" random graphs within a specific design space as a means of generating relevant data which might spark a new conjecture about the desired area of focus. With my analogy in mind, such doodling is reminiscent of the exploratory, rapid prototyping process sometimes used in game design (in what Gingold and Hecker call the "discovery" phase of development). A heuristic to generate new concrete examples of abstract concepts was even present in the original automated mathematician, AM, working with number theory [124].

Beyond generating artifacts with only the indirect intention of knowledge gain, more recent discovery systems internally optimize expected knowledge gain when deciding which experimental setup to test next, realizing an active learning process [24]. Where artifact generation provides opportunistic benefits in mathematical domains (in which graphs and conjectures are non-interactive, static artifacts), discovery systems working in the physical sciences fundamentally cannot avoid artifact generation (as experimental design) during active exploration of their domain.

A notion of "interestingness" is the glue that binds the various subprocesses of automated discovery (artifact generation included) together into an overall control flow [33]. In many cases, interestingness measures the likelihood or quality of knowledge expected to be discovered by taking a particular action (e.g. searching for a counterexample) or focusing on a particular concept (e.g. looking for new examples of special graphs). A system's overall notion of interestingness can be used to induce a measure of value for artifacts generated in its artifact generation subprocesses, a measure related to potential for knowledge gain as opposed to aesthetic value.

Returning to game design, my fourth tenet holds that **automated**

discovery systems inspire a computational model of creative game design that explains the prototypes produced in exploratory game design as the doodles produced trying to flesh out design theories residing in the designer's head, motivated by an interest in designs that have the potential to reveal new patterns.

16.4 Newell's Knowledge Level

To complete the image of the creative game designer as a discoverer, I need some better vocabulary for talking about a designer's knowledge, around which the entire discovery process revolves.

Newell [158] describes the *knowledge level* as a systems level set above the symbolic, program level. At the knowledge level, intelligent *agents* have bodies of *knowledge* and can take *actions* in some environment to make progress towards *goals*. The actions taken by an agent are said to be governed by a *principle of rationality* that states, "If an agent has knowledge that one of its actions will lead to one of its goals, then the agent will select that action." Recall that this sense of rationality is distinct from decision theoretic rationality in that it does not necessarily imply that an agent must optimize anything.

While this radically underspecifies an agent's behavior from a computational perspective, the constellation of concepts at the knowledge level is useful for making statements about the game designer. The intention of knowledge-level modeling is to explain the behavior of knowledge-bearing agents (be they human or machine) without reference to how that knowledge is represented or observational access to an operational model of the agent's mode of processing.

Understanding the game designer, at the knowledge level (see Figure 16.1), starts with making an assumption about what is known and what is sought. But from my theory so far, I can safely assume an important body of knowledge possessed is that of tentative game design knowledge. This knowledge permits the designer the use of tools such as paper and trinkets for physical gameplay prototypes (often styled after board games) and programming languages and compilers for more detailed, computational prototypes. This same knowledge permits understanding to be gained from the observation of game artifacts in play, and suggests a tentative vocabulary for composition of those artifacts (i.e. knowledge of design patterns for game rules). The creative designer's goal, per my analogy with the automation of science, is clearly to gain more design knowledge.

Given this, I expect the designer to rationally (specifically in the knowledge-level sense) go about the practices of game design as part of

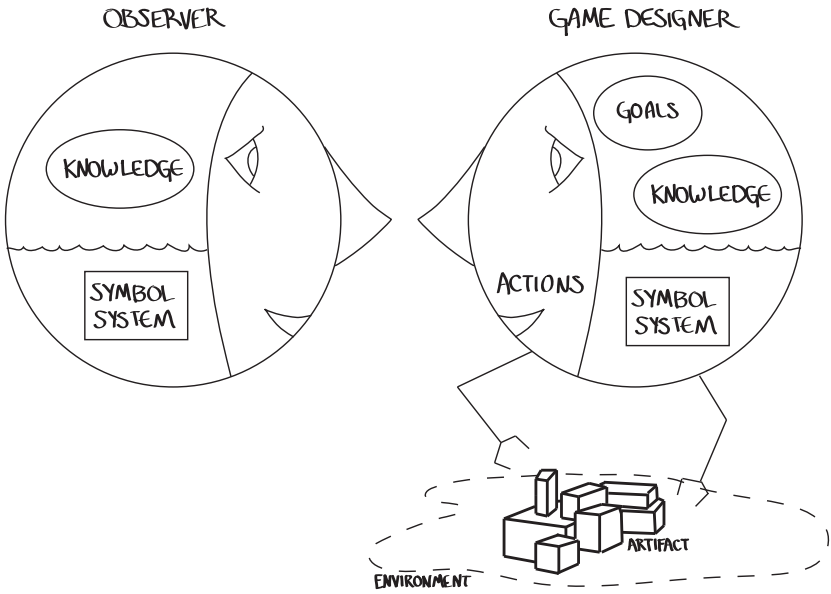

Figure 16.1: Knowledge-level view of the creative game designer. This illustration echoes the illustration Newell used in the article introducing the knowledge level [158]

taking actions that lead towards the gain of design knowledge. That is, **game design activity can be explained as the rational pursuit of design knowledge gain**.

16.5 Creativity as Rational Curiosity

The knowledge level lets me talk about a kind of rationality, one that gives an explanation for why game designers take the actions they do. But not all game design activity is creative, no more than all scientific activity is. So where does creativity come in?

The most creative parts of game design, I claim, are the ones where the designer's behavior is best explained by the direct intention to gain new knowledge, to satisfy *curiosity*. The bulk efforts of game production are a kind of engineering which applies the knowledge gained in the curiosity-driven creative mode.

As the motivation to reduce uncertainty and explore novel stimuli [10], curiosity has long been known to be intertwined with the judgment of aesthetics [11]. Saunders' "curious design agents" [178] generate aesthetic artifacts according to their potential to satisfy an internal measure of curiosity, doing so *in order to learn* about an outside environment. This framework has also been used to drive the behavior of a simulated society of curious visual artists and even a flock of curious sheep in a virtual world [142].

How curiosity-driven behavior can explain the various processes and artifacts of human creativity at a high level has been demonstrated at great length [181]. My unique claim, that *creativity is a knowledge-level phenomenon*, gains similar explanatory power without reference to algorithmic details (such as the use of reinforcement learning or optimization procedures) or human psychology, as in Loewenstein's comprehensive review of research on human curiosity [129].

Looking concretely at curiosity in the domain of game design, consider the example of speed-runs, gameplay traces that demonstrate a way of doing something in a game (completing a level or collecting certain items) much faster than a designer previously expected. Speed-runs are interesting to game designers, from a curiosity perspective, because they often represent a novel stimulus and quickly increase uncertainty about what is possible in a game, creating an urge to seek out related gameplay traces that would illustrate the general pattern by which the run was achieved. With additional experience, the designer can learn to either design out such speed-runs by adjusting the game's rules, or create new mechanics that entice players to master the skills speed-running requires.

Putting together curiosity about design knowledge with knowledge-

level rationality, I have the following new definition of creativity: **Creativity is the rational pursuit of curiosity**, a knowledge-level phenomenon.

This claim applies to human and machine design agents and gives a goal-oriented explanation to sequences of design activity that result in design knowledge gain (clearly including prototyping and playtesting). Creative game designers makes games, not because that is their function, but because they want to learn things about play that require experimentation with certain artifacts to illuminate.

I call this theory *rational curiosity* because it is a knowledge-level treatment of the concept of curiosity that focuses on how curiosity explains the selection of actions towards a known goal. It claims that curiosity, applied rationally and resulting in surprise, will realize behavior recognizable as creative design activity.

16.5.1 Transformational Creativity in Game Design

Consider the model creative game designer over time (imagined visually in Figure 16.2), producing increasingly complex and refined playable artifacts that are in line with the complexity of the designer's currently operating design theory. That playable artifacts are produced is just an externally visible byproduct of the more interesting process going on in the designer's thoughts: the growth and refinement of design knowledge.

Taking a snapshot at any one time, the designer's knowledge is fixed. The present knowledge describes a "conceptual space," in Boden's terms, of game designs and play possibilities. *Combinational* creativity within this space would entail the generation of artifacts from known structures and construction constraints or, perhaps, the enumeration of explanations of a player's behavior with respect to known patterns. Taking a series of steps in terms of the current design theory, producing a new game using a design pattern of interest, producing a prediction of player behavior, and then performing a playtest and comparing the results with the prediction is an example of *exploratory* creativity in this space. These activities are weakly creative in the rational curiosity view because, though they might indeed be motivated by potential knowledge gain, neither realizes an actual change to the designer's personal theory.

Transformational creativity in game design, then, is design activity that results in a redefinition of design theories. Iterative game design, in which many prototypes are produced in succession in response to feedback from playtesting, has the potential to be an intensely transformative process. Such transformations could include the definition of a new design pattern which simplifies the explanation of how another designer's

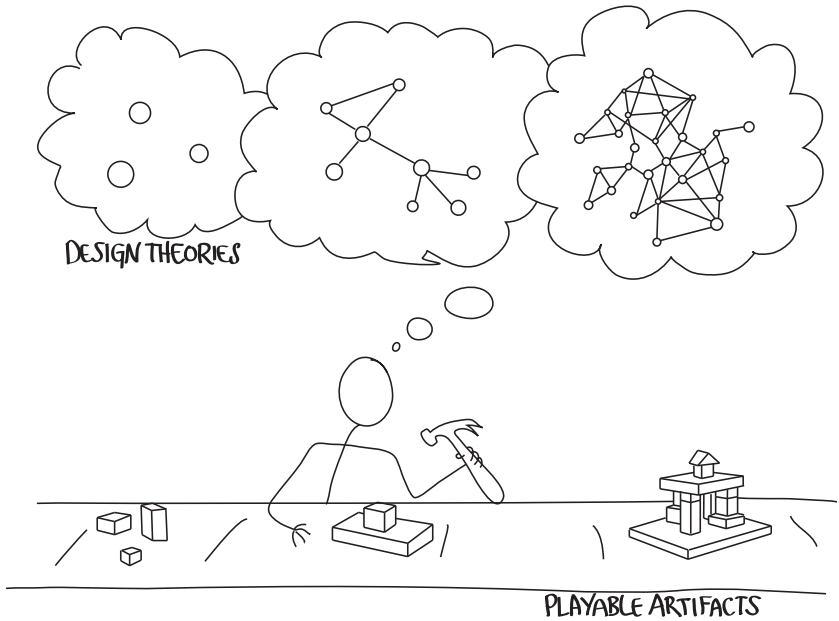

Figure 16.2: Rendition of transformationally creative game design, building to learn to build to learn.

game was constructed,[1] a constraint which limits the use of two patterns together, or a rule which predicts a certain kind of player's behavior when a certain combination of patterns are present.

Not all knowledge possessed and used by a game designer is focused on directly constructive activities. Partitioning a game designer's knowledge into three broad categories, I can distinguish which kind of knowledge is being transformed by different kinds of discovery processes. *Game level* knowledge deals with the structure of concrete games and how those structures shape the objective (mathematically definable) space of play possible in that game. *Play level* knowledge speaks to the linkage between games and particular individual players or classes of players—it captures the gap between the smaller spaces of play that are actually expected to be observed and the larger space of all structurally feasible play. Finally, *design level* knowledge captures the linkage between incremental design moves (adding and removing game elements or assumptions about target audiences) and the effects on observed play. Design level knowledge is what enables intentioned design activity, but it is founded on play level understandings that are in turn based on game level knowledge. These three categories are visualized in Figure 16.3. An interesting implication of this layering is that the capacity for a game design assistance tool to speak of *play*-level concerns will be gated by that tool's capacity to talk about interesting classes of players (i.e. it *must* support some form of player modeling).

Examining specific games (via direct inspection of rules and content or through basic machine playtesting) and self-testing one's own games are actions that a designer might carry out with the goal of refining game level knowledge. But examination and testing, on their own, do not immediately yield transformation. A designer must integrate the results of their actions in a way that changes how they design to be considered transformational, and this, in turn, requires that the results that inspire the knowledge-update to be surprising, to some degree. Likewise, machine playtesting with hypothetical player models or with other human players has the potential to result in refinements of play level knowledge. Finally, the steps taken across a series of prototypes and playtests are what contribute to transformations in design level knowledge.

[1] A tool that implemented the automated explanation of how a game was constructed given a library of design patterns would realize the idea of *design-pattern-aware decompilers* hinted at in Chapter 1.

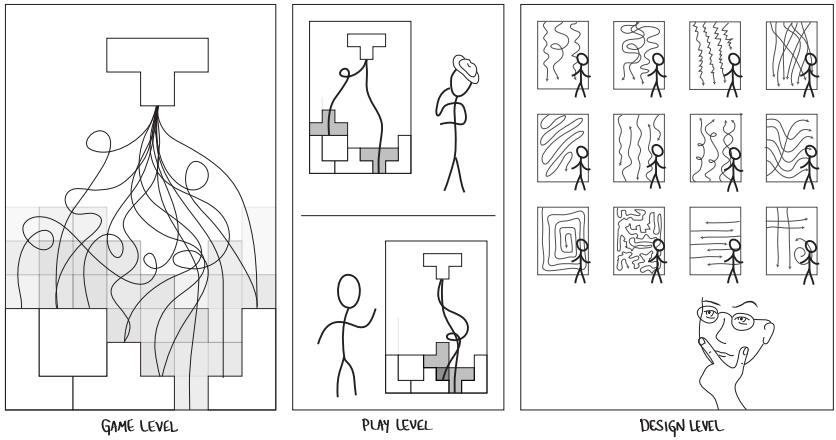

Figure 16.3: Three categories of knowledge a game designer may possess. Rational curiosity will drive a designer to carry out experiments that refine understanding at each level, broadening the space of games the designer can create with informed intention.

16.6 Discussion

Having proposed a theoretical explanation of creativity in game design, I now look at what it entails.

16.6.1 Computational Creativity in Game Design

The theory of rational curiosity in game design can be realized computationally along two major paths: the development of a game design discovery system, and new, knowledge-oriented creativity support tools. More generally, however, it suggests new elements that need to be modeled computationally in support of either path.

Game Design Discovery Systems

Recalling the analogy with scientific and mathematical discovery systems, I can imagine the design of a new kind of discovery system that would work in the domain of game design knowledge. This discovery system would produce games as part of its experiments in exploring play, but it would also allocate significant attention to decomposing games made by other designers and producing explanations of observed human player actions.

The notion of *interestingness* in this discovery system would correspond to a symbol-level realization of the agent's knowledge-level goal: the satisfaction of curiosity about design knowledge. By selecting actions (such as the construction of a prototype, the simulation of a playtest using a known player model, or the analysis of a previously produced game in light of a refined theory) according to their calculated prospects for improvement of the working design theory (a library of design patterns, predictive rules for player behavior, and constraints on play-model construction), the system's behavior would implement a rational pursuit of curiosity: creativity in game design.

Constructing such a system would require new research into adapting *symbol-level representations of design knowledge* for use in game design, the development of a *task decomposition* of creative game design into subgoals and actions (such a concrete design methodology which would be of interest to human designers as well, as an instance of computational caricature), and the identification of the relevant *external tools* of game design (certainly paper prototyping materials and programming environments are some of these tools, but there is ample room for more).

These avenues of research, thus far largely unexplored, are now much more feasible given the results presented in this dissertation. With examples of how to carry out machine playtesting (with and without specific player models), how to construct code-like content automatically, and how to mechanize other instances of appositional reasoning that might arise, the effort to build a game design discovery system much more clearly focuses precisely on creativity-relevant aspects: implementing a top-level rational pursuit of curiosity.

Knowledge-oriented Creativity Support Tools

With recognition that design knowledge gain is the designer's goal, creativity support tools in game design should focus on easing this process. In terms of Yeap's desiderata for creativity support tools, these tools should focus on *ideation* and *empowerment* [230]. In game design, these translate to the generation of candidate design knowledge for the designer to consider and then leaving the adoption of the new knowledge up to the designer (without undue interference). **Knowledge-oriented creativity support tools should attempt to *remove bottlenecks in the discovery process.***

The gameplay pattern language and corresponding search tool described in "Towards Knowledge-Oriented Creativity Support in Game Design" [197], which is intended to accelerate the extraction of feedback about game designs from pre-recorded traces of play, is an example of

this philosophy in action.[2] Neither the system implementing the pattern language nor the BIPED system (from Chapter 13 that it integrates with are themselves creative (in the sense of rational curiosity), but they are designed to provide new actions to the creative designer for rational selection in the service of knowledge gain.

Through the process of representing design spaces (in ASP or otherwise), I aim to encourage designer-programmers to produce their own support tools. Creativity support tools need not come only from the small population of researchers that are familiar with computational creativity. The development of project-specific tools can be a part of a designer's everyday reflective practice. The *Refraction* tools (from Chapter 15) are an example of the co-development of a complex game and design automation tools for that game.

16.6.2 New Perspective for Computational Creativity

I have mostly focused on game design, but rational curiosity is intentionally defined so as to apply to other CC domains. In fact, it should apply even to domains with apparently non-interactive artifacts. Where, in game design, we were concerned with the implications of game rule systems on player actions and reactions, in the domain of music we could explore the implications for sound patterns on audience anticipation and mood, in visual art the implications for perceptual details on where the viewer's eye lingers or flees, and in sculpture the implications of geometric arrangements on audience interest from particular viewpoints. Such domains are not as obviously interactive as game design, but they could be equally deep in the subtlety of how an audience reacts to an artifact—depth enough to keep the rationally curious artist busy producing experiments for quite some time.

The knowledge-level analysis of creativity suggests new questions to ask of CC systems: What does this system want to learn? How is knowl-

[2]This system was originally created as an in-house tool to address one of the bottlenecks that rose in my own discovery process when working with BIPED (digging through archived play traces for additional instances of an interesting pattern that had just re-emerged in a recent playtest). After the fact, it became clear that the system was also a reasonably self-contained example of how to build creativity-support tools that are centered around accelerating knowledge gain. The details of this system are not reviewed in this dissertation because the primary technical challenges that the project involved (developing a miniature Datalog implementation with a focus on interactive queries from a command-line shell) would distract from my primary emphasis on modeling design spaces. For more information on the late 20^{th} century art form of Prolog-based meta-interpreters for domain-specific programming languages, see *The Art of Prolog* [210, chap. 17] and *The Craft of Prolog* [159, chap. 7] (preferably in that order).

edge represented in this domain? Is the system experimenting with the affordances of the raw medium or focusing on nuances of audience reactions achievable through it? (Both are equally creative assuming the desired kind of knowledge is gained.)

Consider *NEvAr* [132], a creative system in the domain of visual art. Unlike the straightforward interactive genetic algorithm in *PicBreeder* [185], *NEvAr* does not ask its audience for feedback on every artifact it internally considers. The system summarizes sparse feedback from its human audience in the form of a neural network that becomes a proxy for their ratings in an internal evolutionary process. Rational curiosity would describe *NEvAr* as a creative system, not because it produces novel and valuable images, but because, at the knowledge level, the system appears to be rationally soliciting reactions on believed high-valued images and incorporating the responses in a way that transforms the space of images that the system will next produce. That is, it behaves consistently with the explanation that it is rationally pursuing its curiosity (albeit with limited design knowledge storage capabilities). If redesigned from scratch with rational curiosity in mind, the system might incorporate a more interpretable representation of learned knowledge that is easier to read as a design theory that improves over time. Alternatively, it might put more computation into experimental design, reasoning over the expected knowledge gain from enticing the human audience to provide feedback on a particular work rather than always trying to display the estimated-best available artifacts. Orienting the system around active learning, I predict, would improve the system's apparent creativity.

From my perspective, it is natural to ask what a system learns as it runs. Though established techniques in computational visual art such as design grammars and iterated function systems can, in some cases, produce very interesting images, the static nature of these techniques in isolation implies that, over time, our sense of novelty of the kinds of artifacts these techniques produce will necessarily wane because these techniques do not learn. While rational curiosity would deem a technique that merely samples a fixed generative space uncreative, these techniques are still valuable to us.[3] They encode very rich generative spaces that, upon gaining experience through experimentation, a creative agent can redefine as part of large-scale experiments.

[3]Coming out of my own artistic practice (and, to a large degree, procrastination in grad school), several members of my family have received artifacts derived from design grammars and iterated function systems as holiday gifts. So, *clearly*, these techniques can yield novelty and value.

16.7 Conclusion

I have followed the clues embedded in the practices of human game designers to a set of building blocks for a theory of creative game design. To recap:

1. Game designers are really designers of play.
2. Creative game design is about seeking knowledge.
3. Designers act as explorers in a science of play.
4. Automated discovery systems inspire a computational model of creative design.
5. Game design activity can be explained as the rational pursuit of design knowledge gain.

This has led me to a new statement about creativity that can apply to human or machine design agents in any artifact generation domain: **creativity is the rational pursuit of curiosity**, *a knowledge-level phenomenon.*

I hope this model of creativity will inspire the exploration of discovery system architectures for artifact generation systems and the development of a new space of knowledge-oriented creativity support tools.

Chapter 17

Evaluation

Following a design-of-design approach, the primary contribution of this dissertation is a designed artifact: a particular practice for mechanizing game design. Specifically, my contribution is the use of answer set programming and related symbolic AI techniques to represent design spaces of game rules and content, empowering the designer-programmer to develop their own powerfully automated synthesis and analysis tools for exploratory game design.

As a designed artifact, it is natural to ask just how appropriate the artifact is to its environment. My particular approach to mechanizing game design may fail by not being usable on realistic problems, not being usable by realistic teams, or not yielding any results that could not be directly had by some easier approach. Unpacking each of these potential failure modes in turn, I develop specific questions for evaluation. Answers to these questions come from the interactions between my practices and the realistic design problems to which they have been applied.

- Does ASP, as realized today and running on todays computing resources, cover realistic design automation problems of interest?
  - While ASP provides many representational benefits over lower-level representations (e.g. SAT), this alone does not imply that the critical facets of appropriateness in realistic exploratory game design problems will be natural to express in realistically authored design space models with ASP.
    - **Can ASP naturally express realistic conditions of appropriateness for design spaces?**
  - Nearly all of the computational problems that arise in the kind of design automation I address are immensely complex (at least

NP-hard). Assuming $P \neq NP$, there will never be algorithms with attractive (i.e. polynomial) scaling for even the simplest of these problems. This state of affairs does not immediately mean that automated synthesis and analysis in game design are *practically* intractable; what matters is whether the cost associated with waiting to get a response for the constant-sized problems that occur in practice is acceptable in exchange for the insight gained.

    * **Can informative tools derived from my practice be made to perform acceptably on realistic problems?**

- Can this practice be followed by the designer-programmers it targets?

  - Even if I can successfully use this practice, it does not immediately follow that others (particularly those designer-programmers with little to no previous symbolic AI experience) will meet with success. The amount of symbolic AI expertise may simply be unreasonable to require.

    * **Does unfamiliarity with symbolic AI traditions block usage?**

  - Much of my original research effort has been finding ways to logically express the concerns of design problems that have arisen in my own exploratory game design experiences. It might be the case that my approach is applicable only to those problems that I have specifically addressed previously and progress by others is blocked on my own development of ways of generalizing it.

    * **Are the followers of this practice generally empowered to solve their own project-specific design automation problems?**

- Do my practices unlock exploration of new design territory?

  - One reason that my practices might be tractable and usable (in the senses required above) is that they may be simply a renaming or repackaging of already proven practices in exploratory game design (PCG, in particular). In order to be able to explore new territory, my approach must offer something fundamentally different than what has come before.

    * **Are there fundamental differences between my proposed practices and those that are already being used by other designer-programmers?**

- Even if my practices are novel, to gain wider attention, they need to offer access to new territory. A reduction in required design effort becomes much more interesting when that reduction makes entirely new designs feasible.
  - **Are radically new game designs made feasible through this practice?**

17.1 ASP for PCG

The general practice of using ASP as a procedural content generation approach has grown beyond a pet practice of my own. So far, at least two graduate level university courses in PCG have adopted ASP-based modeling practices into their curriculum (for a single class or lab section each). Further, ASP now plays a central role in the ongoing design automation research for students outside of these classes.

In Julian Togelius' 2011 class on procedural content generation (at the IT University of Copenhagen), students were introduced to the ideas covered in my journal article [195] and given access to the example source code accompanying a blog post[1] I wrote for the benefit of this class. While I do not have access to the breadth of projects that students created for the mandatory ASP assignment, I do know of one student project that made it to wider publication.

In "Compositional Procedural Content Generation," Togelius, along with Tróndur Justinussen and Anders Hartzen [218] review the idea of "compositional" generators (where a larger generator is made from cooperative interaction between smaller generative components). Their example system, a dungeon map generator similar to the one given in the blog post, uses an evolutionary optimization system to tune numerical parameters exposed by an ASP-encoded design space model. For each of several artifacts sampled with the problems seventeen numerical parameters held fixed, they compute the fitness of the artifact by a custom evaluation function that measures proxies of challenge (a function of solution length) and skill differentiation (damage taken by informed vs. naïve simulated players). Although this system largely misses the chance to use ASP to automate appositional reasoning (to directly synthesize appropriate artifacts), it does provide evidence that students, with minimal backgrounds in symbolic AI, are able to work with AnsProlog well enough to create a generator for a new kind of game content.

In Jim Whitehead's 2012 class on procedural content generation (at the University of California, Santa Cruz; my home institution), I personally introduced students to ASP using the same materials prepared for

[1] `http://eis-blog.ucsc.edu/2011/10/map-generation-speedrun/`

Togelius' class along with live demonstrations of iterative program development. In this class, students were given more time to complete their projects, however they would only use ASP if they decided their content generation project required it. By the end of the class, two students presented original ASP-based content generation systems. Even though these projects were developed in the context of a content generation class, both systems involve modeling design spaces for the purpose of enabling new gameplay (as opposed to simply generating static artifacts).

John Grey [87] presented a system for role-playing game dialog tree generation that, quite unexpectedly, embedded dialog trees into a grid-based map representation. In the resulting spatial-conversational puzzles, moves on the grid are only allowed if there is a natural flow of conversation between two tiles (a "gossip" tile can be reached from an "ask" tile while the only tiles reachable from "condemn" are "goodbye" tiles). In traditional dialog trees, designers may accidentally (or even intentionally) encode maze-like puzzles where the player, presented only with a small number of dialog options, has no intuition for which options to pick to bring about any particular goal. In Grey's spatially embedded dialog trees, a player can employ their own maze-solving heuristics (such as trying to move in the general direction of an exit) to help themselves towards the conversational outcome they desire. Although Grey's prototype is rough, it shows the potential for an entirely new kind of mini-game design that would be unreasonable to consider if the same kind of spatial embedding logic used in the *Refraction* tools was not available for immediate and opportunistic reuse.

Kate Compton presented a prototype for a new game for which one of the core mechanics was directly enabled by the use of ASP to automate combinatorial search. Published as "Anza Island: Novel Gameplay Using ASP" [39], the prototype uses ASP to implement the intelligence of an adversarial non-player character. Anza, the games antagonist, must devise ways of altering connectivity between island regions that simultaneously thwart the player while also being absolutely consistent with any constraints the player has constructed by combining game objects in a mechanic called "logic crafting." Again, opportunistic reuse of combinatorial search infrastructure has allowed a student (and commercial game industry veteran) without previous logic programming experience to realize a game design that, while likely to be realizable via other means, would not have been conceived in exploratory game design otherwise.

Outside of Whitehead's class but still at UC Santa Cruz, Ph.D. candidate Sherol Chen is pursuing a program of doctoral research in interactive storytelling that involves reasoning over spaces of variation in the telling of a story skeleton. In an early prototype produced as part of the research, I assisted Chen in developing a story generator, RoleModel [29],

that blended the traditionally separated story generation paradigms of story modeling, character modeling, and author modeling. For a generated story to be appropriate, it had to satisfy constraints on story structure (that certain skeletal events always occur), constraints from character ability and volition (that, for example, dead character's cannot speak and non-aggressive characters will never perform actions for which simulated readers would impute malicious intent) and other author-specified constraints (that certain multi-event patterns are required or forbidden or that certain world-state will always be the result—e.g. someone dies in every episode).

At the University of Washington, other doctoral students involved in the *Refraction* project are continuing to build on the design automation tools that I developed. For example, Eric Butler has nested the puzzle generation and puzzle solving tools (described in Chapter 15) in a larger system that produces levels for which the only solutions are solutions involving the target mathematical concept (posing verified level design automation as an NP^{NP} problem, within range of disjunctive ASP, instead of the NP problem used in the original problem formulation). Meanwhile, Erik Andersen is investigating using ASP and related declarative approaches to combinatorial search to model the level progression design problem: deciding which requirements to place on a player's next puzzle given their performance on previous puzzles.

Finally, at the Georgia Institute of Technology, doctoral student Alex Zook has reported to have used my research as a starting point to replace a project-specific genetic algorithm in the player-adaptation component of in his training game scenario generator [233] with a more concise and controllable answer set program.

17.2 Applied Systems

In the following subsections, walk through each of my applied systems in turn, gathering evidence for their impact on the questions above.

17.2.1 Ludocore

The Ludocore system was originally created as a follow-up to Mark Nelson's program of modeling game mechanics with the event calculus. Nelson's early systems were based on *decreasoner*,[2] a dedicated event calculus planning system. Seeking a programming syntax closer to the standard Prolog syntax, I discovered that Mueller (proponent of the event calculus and developer of *decreasoner*) had also developed an AnsProlog

[2] `http://decreasoner.sourceforge.net/`

encoding of the essential axioms of the event calculus. When I later discovered that many uses of LUDOCORE's "structural queries" to generate game content did not depend on the event calculus axioms, it was natural to simply delete these axioms and use the raw affordances of ASP (as offered by the now-obsolete LPARSE/SMODELS[3] answer set solving tools) in my early, unpublished experiments in generating dungeon maps with an unstructured graph representation.

In many cases, the appropriateness of an artifact in a design space is deducible directly from its static form, and no framework for temporal causality is required to express appropriateness in AnsProlog. However, when the appropriateness of an artifact hinges on dynamic interaction, the range of this interaction needs to be expressed in logical rules. Because building logical encodings of the state of a system over time can be a difficult and subtle problem (this is exactly the problem the event calculus was invented to solve!), it is rather convenient that six[4] AnsProlog rules suffice to upgrade a working design space model with a rich model of state and events over time.

The general design of LUDOCORE and its extensions to the event calculus (including *possible* and *conflicts* conditions expressed independently of *initiates* and *terminates* rules) paved the way for the RoleModel system in which stories were represented, in effect, by particular gameplay traces. LUDOCORE's flexible use as a gameplay trace inference system transformed into RoleModel's multiple uses in story generation: "(1) a tabula rasa generator, which takes few or no constraints and autonomously generates varied narratives from the background theory, (2) a partially constrained generator, with which the author can specify additional story constraints on top of the background theory, such as constraints on role fillers, character traits, and even the appearance of specific events within the story, without locking down a specific linear sequence of events, and (3) a highly constrained generator, with which an author can specify a linear story that the system generates variations and explanations on" [29].

17.2.2 Biped

Recall that in the BIPED system, a single logic program drives both offline machine playtesting (in LUDOCORE) and online human playtesting (in BIPED's player-facing interactive interface). To understand whether authoring game sketches as logic programs was feasible for AI-inexperienced

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

[4] I have collected the most commonly used axioms from Mueller's more complex encoding of the event calculus in this small simple AnsProlog library: `http://adamsmith.as/typ0/ec.ans`

designer-programmers and whether BIPED's language could express game designs of interest, I engaged two undergraduate students in a quarter of independent study in 2010.

Vivian Wong (then sophomore) and Erica Woolley (then junior) were students in the Bachelor of Science game design degree program at UC Santa Cruz. They were competent programmers with videogame development (e.g. implementing collision detection) and videogame design (e.g. paper and computational prototyping and playtesting) experience. Their challenge was focused on the human side of BIPED, brainstorming new language features and idioms that might be useful, implementing them, and performing exploratory game design with them. Together, they identified and added a number of new language features as well as developed a number of game sketching idioms that made playtesting prototypes in BIPED more natural.

New Language Features

The original version of BIPED labeled visual tokens and spaces in a way that made their coherence with the abstract mechanics as explicit as possible. In the example game *DrillBot 6000*, the abstract location *e* and the abstract rock *dino_bones* were represented with a space labeled "e" and a token labeled "dino_bones" (the appearance of all spaces and tokens, modulo label text, was uniform). For BIPED to gather feedback from human play of richer game designs (for comparison against results from offline testing with LUDOCORE), BIPED's graphical interface needed to be able to express richer visualizations of the abstract game state.

Staying within the paper-inspired tokens, spaces, and lines paradigm on which I founded BIPED, Wong and Woolley greatly expanded the expressiveness of those tokens and spaces. By the final version, they had added support for a number of new representational rules that they themselves could optionally exploit to customize the appearance and functionality of their game sketch.

The new *ui_token_label(Token,Label)* (or *ui_space_label(Space,Label)*) predicate could be used to hide the underlying symbolic name of a token (or space) and replace it with a computed name at each logical timepoint. For example, the token identifier *monster(5)* (which exposes irrelevant information: the monster's serial number) could be replaced with the computed label *m(hp=2)* (which embeds otherwise hidden game state, the monster's health, in its displayed label). The most common use of this modeling primitive was simply to hide the distracting name of tokens and spaces that could already be understood by their spatial relationship to other tokens and spaces. For example, hiding the names of every location space in *DrillBot 6000* except for the *base* space was an obvious improvement

that reduced visual clutter.

To make certain kinds of tokens (or spaces) more recognizable (e.g. distinguishing the special token for the mining robot in *DrillBot 6000* from the plentiful rock tokens) the new *ui_token_color(Token,Color)* predicate specifies a non-default background color (an arbitrary RGBA value) that can vary at each logical timepoint in the game. Now, color conventions could be invented and followed to distinguish clickable spaces used to represent abstract buttons from those spaces intended to represent physical locations in the game world. Colors were often used to distinguish the type of a token in sufficient detail that its symbolic label could be hidden (e.g. any yellow token is collectable treasure; any red token is a monster to fight).

Finally, the *ui_token_image(Token,ImgUrl)* predicate specified an optional, time-varying mapping from a token (or space) to an arbitrary image on the web or the local file system. When testing with players who did not have patience for decoding an abstract depiction of the game world, this predicate allowed the inclusion of custom art assets.

In *GridSlasher*, a grid-based puzzle game sketch with simplified roguelike mechanics, the undergrads added a mechanic the required that certain monsters could only be attacked from the side, not head-on. Unsatisfied with trying to represent the current facing-direction of a monster and the player character with custom text labels, custom images allowed the recycling of a sprite sheet from another game that used a different sprite for each direction. The resulting visualization was direct and required no conscious decoding to interpret. Another use of custom images, pioneered in *GridSlasher*, was the declarative creation of a linked grid of spaces, all depicted with repeating tile art. Abstractly, navigation on a regular grid is no different than navigating on an arbitrary graph, but presenting the grid as a uniform field of tiled background art conveys the intended concept to the human player much more succinctly.

In one language feature reaching out of the tokens/spaces/lines paradigm, Wong and Woolley improved the detail text generation system. Previously, I had exposed a way for designers to express a list of logical symbols that should be dumped into a movable text box on screen (hard-coding a blob of text, for example, could be used to display on-screen instructions). As the demand to express richer game state grew, the undergrads found it necessary to add logic that could automatically flatten nested lists of game state text into a single paragraph. This same logic could have been expressed in the definition of the original *ui_details(Symbols)* predicate, but adding pretty-printing to this feature by default greatly reduced the tedium associated with displaying debugging data in the text panel.

Finally, in a refinement to the event triggering logic, the undergrads added support for rules like *ui_triggers(ui_keypress(Key),PlayerEvent)* for

intercepting physical key press events. For games like *GridSlasher*, keyboard input provided a natural means for navigating the rectilinear mesh of linked spaces.

New Sketching Idioms

Along with adding new features to the game sketching language, the undergrads also developed a number of interesting new patterns for using BIPED's representational primitives.

In the *DrillBot 6000* example, clicks on the *base* space were interpreted in two ways: if the mining robot was in a nearby space, the click was interpreted as a trigger for the event of moving into the base; if the mining robot was already in the base, the click was interpreted as a trigger for the *trade* and *refuel* events. This use (or abuse) of a single space as both a depiction of a physical location and an abstract button for players to press was untangled and clarified in the idiom of adding on-screen controls. In this idiom, an arrangement of extra spaces was coded into a game sketch with the explicit intent for use as clickable buttons. In variants of *GridSlasher*, *on-screen controls* (for triggering north, south, east, and west movement events and attacking events) were first placed on one side of the game board (in an arrangement similar to a D-pad), later recolored to be distinguished as buttons, and then re-labeled to signify which keyboard keys could be used to trigger the same events.

A variant on the on-screen controls idiom was the *meta-controls* idiom where additional abstract mechanics and visual representations are added to the game sketch to facilitate exploratory testing. These extra actions, such as toggling "god mode," resetting monster positions, or resurrecting slain monsters, were not intended to be used by players. Instead, these actions were intended for use by the designer acting as a game master, steering the trajectory of a playtest in directions that, while inconsistent with the formal mechanics of the game, were more informative to the designer or made better use of the playtester. If events associated with these actions are never tagged as player or natural events in the game definition, the machine playtesting side of BIPED would know not to attempt to use these cheats when inventing gameplay traces that satisfy specified constraints.

Finally, even though no answer set programming is involved in the human playtesting side of BIPED, the presence of a general purpose Prolog interpreter enabled a basic form of procedural content generation. In *DrillBot 6000*, a significant fraction of code defining the game sketch is dedicated to declaring the abstract properties of the game's map (explicitly describing the abstract properties for each underground cavern, the properties of the rocks initially in it, and where, with floating point num-

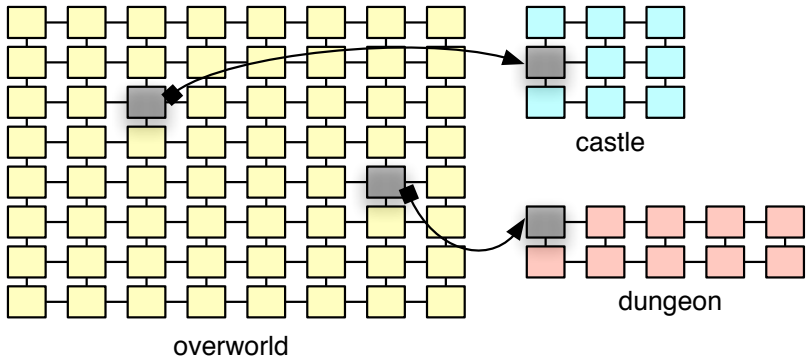

Figure 17.1: Map for a game sketch involving several interacting grid maps linked by directional portals. This is an example of simple content generation using deductive rules understandable by both BIPED's machine and human playtesting sides.

bers, to place the space that represents that cavern). In *GridSlasher*, the abstract and visual properties of the large grid map (an n-by-m network) could be derived from a single assertion, *grid_map(N,M)*, by adding brief rules that deduced *ui_space(Space)* and *ui_layout(Space,FloatX,FloatY)* from it. Because the map generation idiom existed in the code of a game sketch and not as a fixed feature of the BIPED system, it was a small change to support multiple grid spaces linked by portals using assertions like the following (which describes a multi-grid map analogous to the one shown in Figure 17.1):

```
grid_map(overworld,8,8).
grid_map(castle,3,3).
grid_map(dungeon,5,2).
portal(cell(overworld,3,3),cell(castle,1,2)).
portal(cell(overworld,7,5),cell(dungeon,1,1)).
```

Because this map is declaratively expressed by deductive rules understandable by both sides of BIPED, it is suitable for use in both human and machine playtesting. As BIPED's language does not allow choice rules (recall that it is really just Prolog), this mode of content generation is limited to simple, deterministic computation (where a single, regularly-structured artifact is expanded from a compressed and, ideally, more natural to author representation).

Summary

Wong and Woolley's experience carrying out exploratory game design did not uncover game designs that would not be reachable otherwise—all variations of *GridSlasher* could have been realized with other videogame development tools. However, it did demonstrate that authoring logical rules was not unrealistic for these designer-programmers. Between proposing and implementing new language features and inventing new idioms, they were empowered to build their own support for the kind of games they wanted to make.

In their exploration, the undergrads did not encounter the need to break out of the event calculus (*initiates* and *terminates*) formalism for representing game mechanics. Even though some of their game variants became, in effect, real-time simulations (using a `ui_ticker` that fired several times per second instead of the once-per-musical-beat ticker in *DrillBot 6000*) capable of testing human twitch reactions to wandering monsters (using keyboard controls, in particular), the core mechanics of the game retained a representation that was fully amenable to formal analysis in LUDOCORE. Whereas real-time games with human hesitation and reaction delays lead to empirical gameplay traces spanning thousands of timepoints, LUDOCORE's analytic gameplay traces could usually demonstrate similar behavior in a much smaller number of logical timepoints.

17.2.3 Variations Forever

As *Variations Forever*, the game project and the research project, was primarily developed and tested by myself, I report on its function as a computational caricature of AI in the game design process (as opposed to, say, a deployed game). Towards my evaluation questions, I review VF's encounters with the conditions of appropriateness, performance concerns, and its ability to uncover new territory in game design and design automation.

Appropriateness

At the ruleset level of detail, the VF prototype was able to capture all of the requirements of interest that arose in exploratory sampling. What VF most importantly highlighted, however, is that many of the flaws in the generated mini-games were actually flaws in the fixed game engine—something out of reach from the ruleset generator. A common example of this is the situation where the engine decides to spawn key characters (such as the player character) inside of walls, making instantiations of otherwise reasonable rulesets impossible to play.

The ongoing Game-o-Matic project [222] uses a similar level of abstraction to represent the rules of the games that it generates to the one used in VF. That is, the game generator can ensure certain mechanics are in play in a game, but it cannot (as of the time of this writing) express the requirements of sane initial conditions for spatial arrangement or reason about how the game unfolds over time. Thus, at the time of writing, Game-o-Matic suffers from some of the same play-breaking bugs that VF does.

Because the location walls in VF was only dictated by the rules at the level of using or not using the "random obstacles" mechanic, the information necessary to place characters outside of these walls was not available to use in constraints. A relatively small engineering change (allowing the ruleset generator to also place walls and character starting locations) could resolve this situation in isolation, but negotiating the precise level of abstraction in use between generator and engine is not an easy problem, nor is it one that is particularly stable with respect to shifting requirements on how generated games should play. Without having solved automatic programming (in the sense of generally replacing human programming effort with machines), there will always be some level below which the generator cannot be responsible for design choices.

ASP seems ready to generate a much wider array of game-describing content (LUDOCORE shows it can be used to reason about game dynamics under the influence of player models), but new game engines that allow much more of a game's mechanics to be specified at runtime are needed first. In traditional game development cycles, the core mechanics of a game are worked out well in advance of selecting a game engine. In expressive game generation, the fact that the core mechanics may vary at play-time, in response to player input, turns this process on its head. As of the moment, game ruleset generation is not currently blocked by raw generation infrastructure, but it is ensnared by integration concerns that may, in time, place new requirements on generators.

Performance

As formalized, the ruleset generation problem was quite easy for the answer set solver to handle. VF's ruleset generator could easily generate artifacts faster than they could be delivered to players over the network. This should not be surprising, as rulesets are relatively small and simple artifacts (when compared with Ludocore's play traces or *Refraction*'s puzzle specifications). It seems likely that many of the problems that will arise in future play-time design automation problems will follow this pattern: they are nearly trivial to solve, but would be completely unreasonable to consider involving a human game master to support otherwise.

The performance of VF's generator says something about the design space model used: its conditions for appropriateness were too shallow (a limit imposed by game engine technology, as described above). Ruleset generation is far from an inherently easy problem (if anything, generating interactive artifacts *should* be harder than generating non-interactive artifacts), and future formalizations of the ruleset generation problem may tax and even exceed the reasonable limits for acceptable use of ASP. Currently, it is unreasonable to consider using ASP to model rulesets at a per-pixel-per-frame level of detail. If the number of interesting time-points and pixel boundaries can be bounded by small constants, future developments in ASP (such as Clingcon's experimental use of a numerical constraint solver) may allow this.

New Territory

Compared with the game generation system [219] to which VF was designed in response, VF was primarily about defining a space of appropriate mini-games, not about choosing the best mini-game according to a well-defined metric. Recognizing the nature of design problems (in particular, the wickedness of general game design), and knowing that a trusted metric simply cannot be had, it seems that the use of ASP (particularly in iterative development) was distinctly more appropriate than past game and puzzle generation system (e.g. Browne's Ludi [22] or Oranchak's Shinro puzzle generator [160]). The ability to invent and inject speculative assumptions and constraints into a generator without designing custom algorithms is the key to flexibly exploring a large and expressive space of game rulesets.

Follow-ups to VF beyond the reported prototype have not been pursued. Although VF was headed in the direction of new kinds of gameplay experiences, progress was halted by lack of suitable game development infrastructure. Nonetheless, interesting and important results changing this situation may yet emerge from the Game-o-Matic project.

Examining VF as an instance of general, online PCG, it is easy to see how the ideas first developed in VF paved the way for Compton's *Anza Island* and the future versions of *Refraction* that will be adapted to individual players. VF was the first system to deploy ASP-based content generation in a live, interactive, play experience.

17.2.4 *Refraction* Tools

Although the development of design automation tools for *Refraction* is still an ongoing project, I can report on the preliminary interactions between my practice of modeling design spaces in ASP and the challenges

of a complex, pre-existing game design project that could not be altered to suit the capabilities of ASP.

Appropriateness

The first interaction between *Refraction* and ASP was in developing a preliminary spatial embedding solver. Although I was able to quickly (in one day) produce a functioning embedder, reflective practice with this first problem eventually pointed me at a better understanding of the sensitivity of grounding and solving times to problem encoding. I produced a series of alternate encodings of the embedding problem that were increasingly less sensitive to the resolution of the game's grid. One encoding, which focused exclusively on building consistent patterns of relative piece positions, was, in fact, independent of grid resolution and would have been suitable for use on a continuous play space. The best (i.e. fastest for grounding and solving) encoding for the game's fixed 10-by-10 grid was one that mixed relative and absolute positioning concerns. This encoding, later, was particularly convenient for reuse in the puzzle-solving problem were both absolute and relative position details are required to express many properties of interest on solution style (e.g. crossing and abutting).

In interaction with *Refraction's* mission generation problem, my ASP-based practice encountered challenges associated with modeling arithmetic (including the logic of fraction simplification). After inventing new idioms for Lua-based metaprogramming and bulk table generation, this challenge was eventually overcome in a way that yielded new insight into the game's design: for a given set of pieces, only a relatively small space of fractions can be constructed.

Combining the mission generator and spatial embedder into a monolithic puzzle generator highlighted a problem with the initial overall problem formulation that underlay the project's original hand-rolled search tools: the existing tools were focused on generating puzzles that *permitted* an educationally appropriate solution to a puzzle without *requiring* that property of all of the puzzles solutions. Designing interactive artifacts (such as a puzzle with many solutions) is naturally a problem with a higher complexity. Although any NP^{NP} problem is theoretically expressible in ASP, so far, only those problems in NP seem natural to encode in AnsProlog. New language features need to be added to AnsProlog to assist in naturally expressing these elevated complexity problems that the underlying solvers can already handle.

So far, we have not attempted a direct encoding in disjunctive ASP via the mind-boggling "saturation" idiom. The ongoing effort to generate puzzles with verified properties of all solutions, currently lead by

Eric Butler while I complete this dissertation, is based on a guess-and-check (i.e. generate-and-test) architecture that misses opportunities for the "guesser" solver to learn from failures repeatedly highlighted by the "checker" solver. As described in § 18.5.1, a key piece of my planned future work is to address natural encodings of this kind of problem.

Performance

The new, ASP-based tools are distinctly faster than the original hand-rolled search tools, producing solutions in seconds even for problems on the difficult end of realistic for expected *Refraction* puzzles (e.g. those with enough active pieces and blockers so that the beams flowing between pieces cover almost all available spaces on the 10-by-10 grid). However, when the puzzle solver is used in the inner loop of the prototype guess-and-check system, even one second of computation per solution is an unacceptable delay for use in a future game system that invents a new level for a player not long after they have completed the previous one. Although figuring out how to harness disjunctive ASP will undoubtedly outperform the current verified generation prototype, it remains to be seen whether this result performs acceptably well to be used online. If it is revealed that even disjunctive ASP does not provide acceptable performance, either the game's grid resolution must be reduced, the variety of potential pieces must be narrowed, or online puzzle selection must be implemented by selection from a smaller pool of pre-verified puzzles (missing many opportunities for player adaptation).

Accessibility by Designer-Programmers

I did most of the original ASP modeling work for the *Refraction* tools, developing a number of new and advanced ASP techniques, but others are continuing the work I started. In particular, Eric Butler was able to quickly add constraints on blocker symmetry when he found the style of unconstrained puzzles unacceptable. He has also gained sufficient fluency with AnsProlog to orchestrate the current guess-and-check prototype of the verified puzzle generator (that permits no shortcut solutions for puzzles it emits).

New Territory

The new (ASP-based) tools do not commit to the design of a particular algorithm and are both future-proof and able to exploit a number of advanced techniques that we did not have the resources to back port into the previous tools (aside from the geometric restart policy). With current and future answer set solving systems supporting a variety of parallel and

distributed computation configurations, this implies the potential for new ground to be broken in the development of refined design automation tools for *Refraction*. Despite the availability of the current design automation tools within the *Refraction* development team, the deployed version of the game still uses hand-authored content because this static content in a static progression can be trusted. If the current design automation prototypes prove effective, they will play a central role in a subsequent game, internally called *Infinite Refraction*, in which both level progressions and the specific levels are generated, with generated levels having their educational constraints automatically verified (a first for procedural content generation, let alone educational games).

17.3 Recap

I now review my evaluation questions as a whole, looking across applications. Does ASP, as realized today and running on today's computing resources, cover realistic design automation problems of interest? *Can ASP naturally express realistic conditions of appropriateness? Can informative tools derived from my practice be made to perform acceptably on realistic problems?*

These two questions are best answered by my experience with the *Refraction* tools. For a complex and pre-existing game, the entire game, without approximation or simplification, could be modeled in AnsProlog and constraints even beyond those we thought needed to be expressed could be expressed. The solution times were acceptable for the purposes of the original tools. Where *Refraction* does point out a weakness of ASP is in elegantly encoding NP^{NP} problems. The "saturation" idiom required in this situation was simply too awkward to be immediately applied when the need for it arose. Future work remains to be done to examine alternate encodings of this problem that, at least from a theoretical perspective, should not be unreasonable to tackle.

Can this practice be followed by the designer-programmers it targets? *Does unfamiliarity with symbolic AI traditions block usage? Are the followers of this practice generally empowered to solve their own project-specific design automation problems?*

Undergraduate experience with Biped, and, to some extent, my graduate student colleagues experiences with the *Refraction* tools provide a response. No, the requisite symbolic AI experience is not unreasonable to learn, and, yes, people other than myself are empowered to solve their own problems. To the degree that the *Refraction* tools are one of the larger applications of ASP of which I am aware, it is not surprising that a large body of new techniques had to be devised to make these tools possible.

Once cataloged as reference material, such leaps will not be required in the future.

Do my practices unlock exploration of new design territory? *Are there fundamental differences between my proposed practices and those already being used by other designer-programmers? Are radically new game designs made feasible through this practice?*

VF and the future *Infinite Refraction* provide answers to these questions. Algorithms coming from search experts will generally trump best-effort hand-rolled combinatorial search by videogame developers who are not experts in combinatorial search (with the gap rapidly widening as problem complexity increases). Exploiting the latest results in search technology while being insulated from future advancements offers designer-programmers something qualitatively different than the often custom-designed algorithms common in procedural content generation. The focus on a space of appropriate artifacts over the more common approximate optimization afforded by metaheuristic search solutions makes ASP-based generators (or, more generally, *declarative, solver-based generators*) capable of unlocking entirely new applications that require performant and trustable play-time design automation.

The practice of modeling design spaces with ASP, it seems, is quite appropriate to the problems that arise when designer-programmers look to mechanize their own exploratory game design process. Where ASP is found lacking, particularly in being able to directly model problems with complexity beyond NP, is a well-known problem that others working in symbolic AI are interested in solving for their own purposes outside of game design.

Chapter 18

Conclusion

The overarching challenge taken on in this dissertation has been that of accelerating exploratory design practices with the machine-supported reasoning of artificial intelligence. Motivated by a desire to expand human culture-creating ability and an observation that exploration is driven by curiosity, this dissertation has described a practice that designer-programmers can follow to build tools that give them powerful new opportunities to pursue that curiosity.

Acknowledging that game design has an inherent indirectness (a designer crafts games, but really wishes to shape the experiences of play), I have shown how to produce mechanized design space models at two different levels. Design spaces for game rules and game content speak directly to the artifacts that a game designer manipulates. Meanwhile, design spaces of play (encoded as sequences of actions) permit exploration of the interactive possibilities of a given game design and provide an objective reference against which to compare the empirical observation of playtesting with human players.

One of the key observations made and exploited in the foregoing chapters is that symbolic AI offers infrastructure for mechanizing logical abduction that can be repurposed for mechanizing appositional reasoning, a key facet of design thinking. Answer set programming, in particular, offers many desirable properties (e.g. guaranteed termination and the use of advanced combinatorial search and optimization techniques) as a possible knowledge representation substrate for modeling design spaces.

In the remainder of this chapter, I review my impact on the three specific goals of this dissertation and new insights for the fields in my interdisciplinary research context. I then reiterate my primary and secondary contributions, both of which focus on the development of ASP-encoded design space models. Next, I highlight the impact of the five crosscutting

strategies that I introduced in § 1.4. Finally, I briefly describe a program of future work and conclude with my hopes for the broader impact of this dissertation.

18.1 Goals

Recall, from the first chapter, that I have had three specific goals: amplifying game designer creativity, the development of support for new kinds of games employing play-time design automation, and the demonstration of tools that respect the nature of design problems.

18.1.1 Amplify creativity of human game designers

Having revealed what creativity in exploratory game design is after (satisfying the thirst for design knowledge that enables intentioned game design in new territories), I have shown how to produce project-specific design automation tools that ease bottlenecks in the discovery process. By empowering designer-programmers to offload parts of their symbolic reasoning burden to AI-supported design tools, these designers may explore territory for which there are not abundantly available examples for how to craft games or for how players will interact with those games. Pieces of games (rules and content) and traces of their play can be synthesized, and these synthesized artifacts can be compared against sparser and more expensive hand-authored designs and human playtests to yield new design knowledge. Although automating the generation of appropriate artifacts addresses only one facet of creativity (a designer may still struggle with integrating recent feedback into a working theory of the domain, for example), it is one that has deep links to design as it is understood beyond the design of games.

18.1.2 Support deep, play-time design automation

Some game designs require the involvement of a designer in the loop to guide play along interesting trajectories and to produce original content that will be consumed in these player-specific experiences. As involving a dedicated human game designer in every instances of play for these games is infeasible (for availability, cost, expertise, or reaction time concerns), it is natural to look to automated replacements that can carry out the limited design responsibilities required by a particular game design. The mechanized design space models that I have shown how to develop provide what is missing for some of these designs requiring play-time design automation. The *Refraction* tools (from Chapter 15), in particular, are

an important example of the breadth of design processes that can be fully automated with relatively simple design space models.

18.1.3 Demonstrate tools that respect design problems

Acknowledging the inherently *ill-defined* (and potentially *wicked*) nature of design problems, I have shown how to develop design-automation tools on a foundation specifically selected for its ability to support iterative, exploratory development and to ease *problem-framing* activities. Although mechanized design spaces offer automation of only the inner loop of a designer's *double-loop learning*, a reliable oracle for this inner loop allows a designer to focus their own learning and discovery in the outer loop. Noting, first, that not all of design boils down to optimization, and, further, that, even where optimization is relevant, satisficing is what is really desired, I have chosen to found my design space models on substrates that can handle hard constraints: crisp definitions of what counts as "good enough" (otherwise described by *"appropriateness"*). The tools emerging from my practice of modeling design spaces are not simply high-performance solutions to a stated generation problem; they are pointers into a space of *problem formulations* that can be *reflectively* inspected and refined.

18.2 Research Context

In building my dissertation in the interdisciplinary space between four diverse fields, I have gained new perspective and insight that advances the goals of each of these fields.

18.2.1 Game Design

By linking game design with the vocabulary of design studies, I have found general precedents for the textbook practices of game design and broadened our power to describe how games are designed and how they might be designed in the future. The strategy of developing project-specific tools to accelerate one's own exploration is already a standard practice, at least in the form of building prototypes. Using the practices described in this dissertation, designer-programmers are now able to build a broad array of prototype content generators and machine playtesting tools on a common foundation that does not ask them to become experts in combinatorial search algorithm design (something that would distract

them from the project at hand). The availability of new tools at design-time and technologies for use during play-time both alters how games can be designed and alters the bounds of the space of designs that can be realistically realized.

18.2.2 Design Studies

In a significant contribution to design studies, I have provided a definition of appositional reasoning in sufficiently formal terms for it to be mechanized by systems that support the required features I described in Chapter 8. I have shown several examples of doing this using ASP as one such representational substrate. These ASP-encoded design space models are examples of a new, computational realization of Darke's concept of imposing a *primary generator* [52].

Broadly applying the vocabulary of design has resulted in new observations about the nature of several artifact generation systems for videogames and other computational creativity contexts left unexplored by design studies so far.

18.2.3 Computational Creativity

Advancing the CC goal of realizing artifact generators in a domain traditionally dominated by human creators, I have demonstrated several game content (and game play) generators under a unified programming paradigm without the invention of any new algorithms. The concepts of appropriateness used in these systems provide important and practical examples of what is sought in artifacts outside of those notions of value more often discussed in CC such as gallery-ready visual appeal or emotional impact. For creativity in game design, these examples demonstrate that, in many cases for realistic design problems, something other than player-judged "fun" is sought, and, importantly, we should not overlook the designer as a judge of novelty and value.

All of my example systems have been applications to game design, but the techniques advanced in this dissertation are amenable to producing symbolically describable content in other domains (as evidenced by the music composition system Anton [15] and the machine code superoptimizer Toast [45]).

Examining CC's other goals of understanding how machines can be creative and how to discuss human creativity on computational terms, my theory of *rational curiosity* is the first to define creativity as a *knowledge level* phenomenon. This knowledge-level description of creativity *rationalizes* the activity of physical symbol systems that appear to pursue the gain of design knowledge and suggests specific architectures for future

systems that desire to be read as *mechanizations* of the overall creative process. None of the design automation tools developed in this dissertation are an example of how to mechanize this *top-level* process. However, each provides examples of how to mechanize some of the individual actions that naturally occur as parts of this process (e.g. the development of experimental designs or the simulation of their results according to a working model of an environment).

18.2.4 Symbolic AI

The most practical result of this dissertation for the field of symbolic AI is a number of new applications that stretch the performance and representational affordances of existing symbolic AI tools (namely answer set solvers). This work articulates new requirements on future symbolic AI tools that would better support a mechanization of appositional reasoning than do the tools available today.

Although AI seeks to understand human intelligence in computational terms, my example systems that computationally realize facets of design cognition are not meant as accurate depictions of how humans reason in game design. Instead, these systems should be interpreted as intentionally-distorted caricatures of design cognition that are knowingly built on an alternate, non-human physical symbol system. Nonetheless, the link between automated logical abduction and a designer's often-informal appositional reasoning provides an interesting story for artificially intelligent design thinkers: they may mentally compute in incommensurate methods, but their goal of synthesizing the description of an artifact that is well fit, or *appropriate*, to a working definition of a design problem remains the same.

18.3 Contributions

In Chapter 1, I introduced the primary contribution of my dissertation as the development of technical methods for carrying out several exploratory game design processes with a machine. As these methods, in the form of developing various forms of design space models, have been well covered at this point, I should note the applicability of these methods. It is not enough that I can use these methods to build design automation tools for myself; it is necessary that designer-programmers are empowered to build their own tools for practical problems. As Chapter 17 details, when placed in the hands of designer-programmers with little to no symbolic AI background, my methods yield systems for a variety of problems that are usable, performant, and productive of design insight.

The secondary contribution of this dissertation has been situating these methods within a larger research context. By linking design space models back to the concerns of game design, design studies, computational creativity, and symbolic AI, I have been able to develop the perspectives and insights described in the previous section. That is, building design spaces is not simply a way to accelerate one's personal exploratory game design process; it is a way to investigate larger questions in a variety of fields. For example, the development of computational caricatures (whether built around a design space model or not) can be used to probe the ways in which AI can assist game designers, yielding insight for all four of the fields named above.

18.4 Approaches

Chapter 1 named five crosscutting strategies or approaches that I have adopted in the research supporting the ideas advanced in this dissertation. In this section, I look at how each approach fared.

18.4.1 Design Spaces

My emphasis on design spaces (over specific artifacts) has led to performant content generators that are able to use constraint learning to automate learning in the inner loop of double-loop learning. This kind of learning is left untouched in strict generate-and-test architectures for generative processes and is only approximately implemented in contingent generation architectures (such as in genetic algorithms) that maintain a constant-sized archive of promising candidate solutions. Given that constraint learning is a relatively advanced combinatorial search technique, it would normally be out of the reach of most designer-programmers who are not experts in search algorithm design. However, because the practice of developing design space models produces a declarative (vs. procedural) definition of a generation problem, advanced algorithms can be brought to bear on that problem without the programmer's understanding or even explicit knowledge.

Another impact of focusing on declarative definitions of design spaces has been explicitly raising the question of "what makes an artifact appropriate?" In constructive content generators that produce artifacts that are (to some approximation) appropriate *by construction*, what makes an artifact appropriate may easily go unconsidered. Part of reflective practice, however, is iterating on one's understanding of what a design problem demands. When a design space is only implicitly defined through a procedural definition, a designer must keep track of this design space

mentally as they perform exploratory modifications to a generative procedure. When the design space is reified with an explicit and declarative representation, this level of indirection is removed.

18.4.2 Computational Caricature

In the design of computational systems that model some natural or cultural process, some degree of exaggeration and oversimplification is unavoidable. The practice of computational caricature gives a name and a purpose (specifically, improved recognizability) to the intentional uses of these distortions in making concrete systems. In the context of game design, computational caricature points out an interesting and independent use of prototypes: per tradition, they answer specific questions about a specific game design project; and, as caricatures, they make statements about the nature of game design and how future design practices might be structured. In these terms, computational caricature is an effective practice in a design-of-design approach to design studies (i.e. it is a design research practice).

18.4.3 Target Audience: Procedurally Literate Designer-Programmers

Having a clear target audience has placed requirements on my choice of mechanization techniques, but it has also allowed me to make simplifying assumptions. Targeting practicing designer-programmers means that the modes of mechanization I suggest must not require someone to be an expert in combinatorial search algorithm design (or the willingness to become one). However, it is safe for me to assume that these individuals can read and write symbolic program code and perform the necessary tasks to integrate the output of an answer set solver into a live game (if play-time design automation is required). As Chapter 17 describes, these requirements and assumptions have fared well in use by individuals other than myself.

The strategy of targeting designer-programmers leaves out not only the large population of non-programmer designers but also non-designers who nonetheless wish to generate game content and playtest tentative game designs. Empowering novice designers should serve to expand our creative reach as a culture, however this strategy has not been pursued in this dissertation. As systems like Diorama (the highly-parameterized ASP-based real-time strategy game map generator) demonstrate, it seems possible that the techniques described in this dissertation could fruitfully serve as an efficient and maintainable foundation for high-level tools that

can be operated by individuals who neither are nor perhaps aspire to be expert designer-programmers.

18.4.4 Automating Logical Reasoning

My emphasis on logical reasoning has allowed me to focus on just those potential bottlenecks in the game design process that might become limited by tedious inference: namely twiddling the combinatorial details of a piece of game content/rules to bring about a well-defined outcome or selecting sequences of hypothetical player actions that bring about interesting and edge-scale behaviors of a previously defined interactive artifact. Even within the space of logical reasoning, I have only hinted at the possibilities (e.g. in § 6.1.2) of automating *inductive* inference in game design: automating the learning of patterns that fit or explain a bulk of empirical data.

Although using design space models to automate a portion of the cycle between designers and quality-assurance testers in a traditional game development process has the potential to be very valuable, this is a result of exploiting just one type of abundance offered by today's computational resources. The design-assisting possibilities of exploiting, for example, mass storage or high-speed communication remains to be explored.

18.4.5 Artifacts as Communication

Finally, the artifacts-as-communication strategy has been a foundation of my example systems: all are machines (potentially including humans in the loop) that produce artifacts. This strategy provides a very wide channel of information flow from a design scenario back to the designer (as opposed to offering a smaller, pre-summarized report). Acknowledging learning in the outer loop of double-loop learning as a fundamental and unavoidable process of design thinking, I have shown this strategy regularly provides the backtalk that inspires new formulations of a design problem that account for experience gained in inspection of those artifacts.

Iteratively re-framing a designer's best understanding of a design scenario in the form of a design space model, a designer creates a sequence of well-defined search and optimization problems that can be automatically solved by today's available infrastructure for single-loop problem solving. That is, in a reflective conversation with a design scenario, a designer speaks with tentative problem formulations and listens for artifacts. Successful examples of this outer loop have been given again and again in the multiple programming and modeling tutorials of Chapter 10 and the example systems of Chapters 12 through 15.

18.5 Future Work

Having introduced the practice of modeling design spaces in ASP and situated this practice with respect to the concerns of several related fields, my program of research is not exhausted: the mechanization of exploratory game design is far from complete. In this section, I outline four specific directions for future work that will continue to advance the goals of this dissertation: development of *two-level* design space models, improved infrastructure for using ASP-encoded design spaces in larger projects, building a broader base of example design space models, and exploring a mechanization of the outer loop of double loop learning in game design via the development of a game design discovery system.

18.5.1 Two-level Design Spaces

In terms of the projects reviewed in this dissertation, I would like it to be possible for a ruleset generator, such as the one in *Variations Forever*, to reject rulesets that admit gameplay traces demonstrating exploits, such as those detectable using LUDOCORE. That is, I want to combine design space descriptions at two different levels in a way that lets the appropriateness of artifacts at the upper level (say, of rules and content) depend on both the existence or the *non-existence* of certain appropriate artifacts at the lower level (that of potential interaction with the rules and content under consideration).

Recall from Chapter 8 that one of the requirements of a satisfying representational substrate for design spaces was that it should support the expression of universal constraints. This kind of constraint expresses the idea that *all* solutions to a puzzle solve it in an interesting way, not just the one or small number of example solutions generated at the same time as the puzzle itself. The need for this type of constraint was highlighted in *Refraction* where it is not enough that a player can practice a mathematical concept of interest, we demand that, in certain generated puzzles, the player *must* practice that concept.

This kind of constraint is an instance of a general pattern in the design of interactive artifacts. We want to be able to express that "we want an artifact X so that for all interaction traces Y, p(X,Y) is true. Reasoning over *all* interaction traces can be partially addressed with basic ASP via encodings like that used in the maze generation examples (where a predicate `player_at(X,Y,T)` recorded the potential end state for every possible interaction trace). However, in general (particularly for almost anything but mazes), the space of possible interaction traces is vastly larger than the space of basic artifact descriptions (immensely so when the artifact is a ruleset). Thus, some more efficient (in terms of designer attention and

computational resources) encoding is required.

Naturally encoding the concerns of universal constraints gives rise to a kind of two-level design space model for which generating artifacts is a problem in the complexity class NP^{NP} (problems in NP for a Turing machine with access to an oracle for problems in NP). This kind of problem is equivalent to a certain kind of two-player game played for one turn. The first player, representing the designer, makes a move that represents the construction of an artifact (perhaps a puzzle). The second player, representing a malicious playtester, attempts a response (perhaps a sequence of actions that form a solution to that puzzle) that results in designer regret (evidence of inappropriateness in the artifact's design). An appropriate artifact is one that allows a designer to force a win for the first player in this game—one for which there are no possible interaction traces that they would regret seeing realized with that artifact.

Although *disjunctive* answer set solvers are quite capable of solving problems at this elevated level of complexity [60], how best to encode this kind of problem in AnsProlog is an area of active research in which I am already engaged. In future work, I will refine the mechanisms for specifying this kind of problem (which naturally arises in the concerns of almost any interactive artifact design problem) and produce example design space models. These examples will communicate the relevant representation idioms to other designer-programmers whose criteria for artifact appropriateness hinge on similar concerns.

18.5.2 Integration infrastructure

Current answer set solvers are primarily designed around a batch computation workflow: a complete AnsProlog program is read from static files and the resulting answer sets are printed to standard output. To both better support play-time design automation and enable a new generation of tools that respect design problems, I will develop reusable infrastructure for integrating *interactive* ASP-based artifact generators with the larger games and tools that require those artifacts. The web-service I created to support *Variations Forever* and the Java-based template used by Chen's *RoleModel* [29] and Compton's *Anza Island* [39] are two examples of rudimentary integration infrastructure. In the future, I will investigate the use of reactive ASP systems [74] (such as *oclingo*[1]) and the potential need for languages which declaratively script the interaction between answer set solvers and the larger systems with which they must communicate.

[1] `http://www.cs.uni-potsdam.de/wv/oclingo/`

18.5.3 Broader base of modeling examples

In addition to the two-level design spaces mentioned above, I plan to develop design space examples that demonstrate how to make use of emerging developments in ASP as they become available. In § 10.3 I offered a single example of how to work with numerical constraints for the problem of generating Golomb rulers. As these tools move out of the experimental stage, I would like to create examples of how to model design spaces that are insensitive to the scale of key parameters (such as in the experimental grid embedders for *Refraction* that were insensitive to the resolution of the game grid). Incremental encodings that, following an iterative-deepening strategy, permit solutions whose time and space requirements scale with problem difficulty (as opposed to the scale of the largest possible solution) also deserve attention.

My future modeling examples will also convey important debugging information. While questions like "why is this program slow?" or "why did/didn't you generate this artifact?" can be answered by detailed investigation of a particular answer set solver's internals, this kind of answer is inappropriate and undesirable for the kind of designer-programmers I have been targeting. I plan to develop a clear set of examples that strive to provide insight that helps programmers avoid egregious slowdowns while still abstracting over the details of solver-specific algorithms (which are subject to change). Further, for explaining the generation (or non-generation) of specific artifacts, the ANTON music composition system has demonstrated that, backed by ASP, this kind of diagnosis can be surfaced as a user-facing feature of a content generator. In the future, I will further investigate the use of automated abduction (logical explanation forming) over design space models in a way that yields insight to designer-programmers and offers them building blocks for exposing these explanations to end users.

Finally, in the same vein as the graph visualization system used in several of my example systems, I intend to develop a broad array of flexible visualizations that can be used and refined along with a main design space model. Instead of the universal visualization strategy pursued in ASPViz,[2] I will make a variety of small, scope-limited visualization support tools (following the Unix philosophy: *do one thing and do it well*).

[2] `http://www.cs.bath.ac.uk/~occ/aspviz/`

18.5.4 Automating discovery of design knowledge

Returning to the ideas that originally inspired this body of research,[3] I would like to explore the development of a game design discovery system. This system would not seek to replace human game designers at an industrial scale. Instead, like many of the systems presented in this dissertation, it would function as a computational caricature of the top-level processes in game design, pointing out what they might be, sketching a means for their automation, and (perhaps most importantly) highlighting our misconceptions about these top-level processes.

Where my strategy so far has been to automate efforts in the inner loop of double-loop learning in an effort to ease exploration on the outer loop, a game design discovery system would intentionally perform these outer loop responsibilities itself. This requires the proposal of tentative formalizations of the *problem framing* problem, the process of *reflective practice*, and a means of assembling *primary generators* in the face of incomplete problem definitions. Importantly, however, this project does not require a formalization of the aesthetics of game design, as my theory of rational curiosity suggests a sufficient (though perhaps skewed) sense of value will arise from the thirst for new design knowledge alone.

Unlike my other directions of future work that target the practicing designer-programmer, this direction intends to advance the work of design studies following the design-of-design paradigm. It would further explicate the nature of exploratory game design and sketch what it might become with deeper mechanization.

18.6 Epilogue

It is my hope that attaching my work to the four widely spaced fields linked together in Chapter 7 brings useful structure and insight to exploratory game design, an otherwise highly informal practice. I intend to offer not another prescriptive formula for how to explore, but an array of newly available actions that game designers may opt to select as part of exploring in a different way than they have in the past. By supporting deeper and wider exploration in game design, I hope to materially expand our ability to create human culture.

The focus of this dissertation has been on game design, an inherently interactive art form. However I wish that the technology and practices introduced in this dissertation afford a deep reevaluation of other art forms, operating from an *all art is interactive* perspective. While it is possible

[3]Reading about the rare successes of early discovery systems like *AM* and *EURISKO* [123] are part of what drew me to artificial intelligence research in the first place.

to address automation in game design in a way that sweeps interactivity under the rug (e.g. evaluating rulesets and content via functions over a small number of interaction traces), acknowledging that the space of play is, first, unavoidably existent and, second, almost inconceivably vast prompts a fundamental shift in perspective. This shift should have rippling implications for how we look at mechanization of exploratory design for other forms of art that do not expose their interactive aspects so freely as does game design.

Bibliography

[1] Ernest Adams. *Fundamentals of Game Design.* New Riders, 2010.

[2] C. Ansótegui, C. P Gomes, and B. Selman. The achilles' heel of QBF. In *Proceedings of the National Conference on Artificial Intelligence*, volume 20, page 275, 2005.

[3] K. R. Apt and M. H. Van Emden. Contributions to the theory of logic programming. *Journal of the ACM (JACM)*, 29(3):841–862, 1982.

[4] C. Argyris and D. A. Schön. *Organizational Learning: A Theory of Action Perspective.* Addison-Wesley, 1978.

[5] Daniel A. Ashlock. Automatic generation of game elements via evolution. In *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG)*, 2010.

[6] C. Baier and J. P Katoen. *Principles of Model Checking.* MIT Press, 2008.

[7] Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving.* Cambridge University Press, 2003.

[8] Marco Benedetti. sKizzo: a suite to evaluate and certify QBFs. *Automated Deduction–CADE-20*, pages 736–736, 2005.

[9] Jon Bentley. Programming pearls: Little languages. *Communications of the ACM*, 29(8):711–721, 1986.

[10] D. E. Berlyne. *Conflict, Arousal, and Curiosity.* McGraw-Hill, New York, 1960.

[11] D. E. Berlyne. *Aesthetics and Psychobiology.* Appleton-Century-Crofts, New York, 1971.

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

[13] Jonathan Blow. Game development: Harder than you think. *ACM Queue*, 1(10):28, 2004.

[14] Margaret A. Boden. *The Creative Mind: Myths and Mechanisms*. Psychology Press, 2004.

[15] G. Boenn, M. Brain, M. De Vos, and J. ffitch. Automatic music composition using answer set programming. *Theory and Practice of Logic Programming*, 11(2-3):397–427, 2011.

[16] Georg Boenn, Martin Brain, Marina De Vos, and John ffitch. Computational music theory. In *Musical Metacreation: Papers from the 2012 AIIDE Workshop AAAI Technical Report WS-12-16*, 2012.

[17] M. Brain, O. Cliffe, and M. De Vos. A pragmatic programmer's guide to answer set programming. In *Software Engineering for Answer Set Programming (SEA09)*, 2009.

[18] M. Brain, M. Gebser, J. Pührer, T. Schaub, H. Tompits, and S. Woltran. Debugging ASP programs by means of ASP. *Logic Programming and Nonmonotonic Reasoning*, pages 31–43, 2007.

[19] Martin Brain. ASP galore! (email), July 2012.

[20] Frederick P. Brooks. *The Design of Design: Essays from a Computer Scientist*. Addison-Wesley Professional, April 2010.

[21] C. Browne and F. Maire. Evolutionary game design. *Computational Intelligence and AI in Games, IEEE Transactions on*, 2(1):1–16, 2010.

[22] Cameron Browne. *Automatic generation and evaluation of recombination games*. PhD thesis, Queensland University of Technology, 2008.

[23] R. Brummayer and A. Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 174–177, 2009.

[24] C. H Bryant, S. H. Muggleton, D. B. Kell, P. Reiser, R. D. King, and S. G. Oliver. Combining inductive logic programming, active learning and robotics to discover the function of genes. *Electronic Transactions in Artificial Intelligence*, 5(B):1–36, 2001.

[25] Bruce G. Buchanan. Creativity at the metalevel: AAAI-2000 presidential address. *AI magazine*, 22(3):13, 2001.

[26] Pedro Cabalar. Answer set; programming? *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning*, pages 334–343, 2011.

[27] Jonathan Cagan. *Engineering shape grammars: where we have been and where we are going, Formal engineering design synthesis*. Cambridge University Press, New York, NY, 2001.

[28] Jenova Chen. Flow in games. Master's thesis, University for Southern California, 2006.

[29] S. Chen, A. M. Smith, A. Jhala, N. Wardrip-Fruin, and M. Mateas. RoleModel: towards a formal model of dramatic roles for story generation. In *Proceedings of the Intelligent Narrative Technologies III Workshop*, page 17, 2010.

[30] Choco Team. `choco`: an open source java constraint programming library. Technical Report 10-02-INFO, École de Mines de Nantes, 2010.

[31] Doug Church. Formal abstract design tools. *Gamasutra*, July 1999.

[32] Harold Cohen. Colouring without seeing: a problem in machine creativity. *AISB Quarterly*, 102:26–35, 1999.

[33] S. Colton, A. Bundy, and T. Walsh. On the notion of interestingness in automated mathematical discovery. *International Journal of Human-Computer Studies*, 53(3):351–375, 2000.

[34] S. Colton, R. Lopez de Mantaras, O. Stock, et al. Computational creativity: Coming of age. *AI Magazine*, 30(3), 2009.

[35] Simon Colton. Automated puzzle generation. In *Proceedings of the AISB'02 Symposium on AI and Creativity in the Arts and Sciences*, 2002.

[36] Simon Colton. The HR program for theorem generation. *Automated Deduction—CADE-18*, pages 37–61, 2002.

[37] Simon Colton. Creativity versus the perception of creativity in computational systems. In *Creative Intelligent Systems: Papers from the AAAI Spring Symposium*, pages 14–20, 2008.

[38] K. Compton and M. Mateas. Procedural level design for platform games. In *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment International Conference (AIIDE)*, 2006.

[39] K. Compton, A. M. Smith, and M. Mateas. Anza island: Novel gameplay using ASP. In *Proceedings of the FDG Workshop on Procedural Content Generation (PCGames'2012)*, 2012.

[40] Kate Compton. Using procedural generation for human-computer collaboration, October 2009. `http://eis-blog.ucsc.edu/2009/10/eis-hosts-the-procedural-content-generation-symposium/`.

[41] Daniel Cook. Creating a system of game play notation, January 2006. `http://www.lostgarden.com/2006/01/creating-system-of-game-play-notation.html`.

[42] David Cope. *Computer Models of Musical Creativity*. MIT Press, 2005.

[43] Domenico Corapi, Alessandra Russo, and Emil Lupu. Inductive logic programming in answer set programming. In *Proceedings of the 21st international conference on Inductive Logic Programming*, ILP'11, pages 91–97, Berlin, Heidelberg, 2012. Springer-Verlag.

[44] P. Coussy and A. Morawiec. *High-level synthesis: from algorithm to digital circuit*. Springer Verlag, 2008.

[45] T. Crick, M. Brain, M. De Vos, and J. Fitch. Generating optimal code using answer set programming. *Logic Programming and Nonmonotonic Reasoning*, pages 554–559, 2009.

[46] Nigel Cross. *Simulation of computer aided design*. University of Manchester Institute of Science and Technology (UMIST), 1967.

[47] Nigel Cross. *Designerly Ways of Knowing*. Springer, Goldaming, 2006.

[48] Nigel Cross. *Design Thinking: Understanding How Designers Think and Work*. Berg, April 2011.

[49] Mihály Csíkszentmihályi. *Creative thinking in art students: An exploratory study*. PhD thesis, University of Chicago, 1964.

[50] Mihály Csíkszentmihályi. *Flow: The psychology of optimal experience*. Harper Perennial, 1991.

[51] George B. Dantzig. *Linear programming and extensions*. Princeton University Press, 1998.

[52] Jane Darke. The primary generator and the design process. *Design Studies*, 1(1):36–44, July 1979.

[53] Subrata Dasgupta. *Design Theory and Computer Science*. Cambridge University Press, 1991.

[54] L. De Raedt and N. Lavrač. Multiple predicate learning in two inductive logic programming settings. *Logic Journal of IGPL*, 4(2):227–254, 1996.

[55] Daniel C. Dennett. *Brainstorms: Philosophical Essays on Mind and Psychology*. MIT Press, 1981.

[56] E. Denti, A. Omicini, and A. Ricci. tuProlog: a light-weight Prolog for internet applications and infrastructures. *Practical Aspects of Declarative Languages*, pages 184–198, 2001.

[57] Joris Dormans. Level design as model transformation: A strategy for automated content generation. In *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*, page 2, 2011.

[58] Joris Dormans. *Engineering emergence: applied theory for game design*. PhD thesis, University of Amsterdam, 2012.

[59] Scott Draves. The electric sheep screen-saver: A case study in aesthetic evolution. *Applications of Evolutionary Computing*, pages 458–467, 2005.

[60] C. Drescher, M. Gebser, T. Grote, B. Kaufmann, A. König, M. Ostrowski, and T. Schaub. Conflict-driven disjunctive answer set solving. In *Proceedings of the 11th International Conference on Principles of Knowledge Representation and Reasoning (KR'08)*, pages 422–432, 2008.

[61] S. Dzeroski and L. Todorovski. Discovering dynamics: from inductive logic programming to machine discovery. *Journal of Intelligent Information Systems*, 4(1):89–108, 1995.

[62] T. Eiter and G. Gottlob. On the computational cost of disjunctive logic programming: Propositional case. *Annals of Mathematics and Artificial Intelligence*, 15(3):289–323, 1995.

[63] T. Eiter and A. Polleres. Towards automated integration of guess and check programs in answer set programming: a meta-interpreter and applications. *Theory and Practice of Logic Programming*, 6(1-2):23–60, 2006.

[64] M. P. Eladhari and M. Mateas. Semi-autonomous avatars in world of minds: A case study of AI-based game design. In *Proceedings of the 2008 International Conference on Advances in Computer Entertainment Technology*, pages 201–208, 2008.

[65] M. P. Eladhari, A. Sullivan, G. Smith, and J. McCoy. AI-Based game design: Enabling new playable experiences. Technical Report UCSC-SOE-11-27, University of California, Santa Cruz, 2011.

[66] J. Ellson, E. Gansner, L. Koutsofios, S. North, and G. Woodhull. Graphviz—open source graph drawing tools. In *Graph Drawing*, pages 594–597, 2002.

[67] D. Elson, J. Rowe, A.M. Smith, G. Smith, and E. Tomai. Reports on the 2011 AAAI fourth artificial intelligence for interactive digital entertainment conference workshops. *AI Magazine*, 33(1):55, 2012.

[68] Douglas C. Engelbart. Augmenting human intellect–a conceptual framework. (Summary Report AFOSR-3223 under Contract AF 49 (638)), 1962.

[69] S. L Epstein. Learning and discovery: One system's search for mathematical knowledge. *Computational Intelligence*, 4(1):42–53, 1988.

[70] D. H. Feldman, M. Csíkszentmihályi, and H. Gardner. *Changing the world: A framework for the study of creativity.* Praeger Publishers / Greenwood Publishing Group, 1994.

[71] A. S. Fraenkel and D. Lichtenstein. Computing a perfect strategy for $n \times n$ chess requires time exponential in n. *Journal of Combinatorial Theory, Series A*, 31(2):199–214, 1981.

[72] R. Buckminster Fuller. World design science decade, July 1961. `http://bfi.org/about-bucky/resources/world-design-science-decade-documents`.

[73] T. Fullerton, C. Swain, and S. Hoffman. *Game design workshop: a playcentric approach to creating innovative games.* Morgan Kaufmann, 2008.

[74] M. Gebser, T. Grote, R. Kaminski, and T. Schaub. Reactive answer set programming. *Logic Programming and Nonmonotonic Reasoning*, pages 54–66, 2011.

[75] M. Gebser, C. Guziolowski, M. Ivanchev, T. Schaub, A. Siegel, S. Thiele, and P. Veber. Repair and prediction (under inconsistency) in large biological networks with answer set programming. In *Proceedings of the Twelfth International Conference on Principles of Knowledge Representation and Reasoning (KR'10), F. Lin and U. Sattler, Eds. AAAI Press*, pages 497–507, 2010.

[76] M. Gebser, R. Kaminski, A. König, and T. Schaub. Advances in gringo series 3. *Logic Programming and Nonmonotonic Reasoning*, pages 345–351, 2011.

[77] M. Gebser, R. Kaminski, and T. Schaub. Complex optimization in answer set programming. *Theory and Practice of Logic Programming*, 11(4-5):821–839, 2011.

[78] M. Gebser, B. Kaufmann, R. Kaminski, M. Ostrowski, T. Schaub, and M. T Schneider. Potassco: The potsdam answer set solving collection. *AI Communications*, 24(2):107–124, 2011.

[79] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 386–392, 2007.

[80] M. Gebser, M. Ostrowski, and T. Schaub. Constraint answer set solving. In *Proceedings of the 25th International Conference on Logic Programming (ICLP'09)*, pages 235–249, 2009.

[81] Martin Gebser. gen_sudoku.gringo, 2010. `http://asparagus.cs.uni-potsdam.de/encoding/show/id/12739`.

[82] M. Genesereth, N. Love, and B. Pell. General game playing: Overview of the AAAI competition. *AI Magazine*, 26(2):62, 2005.

[83] J. S Gero. Design prototypes: a knowledge representation schema for design. *AI magazine*, 11(4):26, 1990.

[84] C. Gingold and C. Hecker. Advanced prototyping, 2006. `http://levitylab.com/cog/presentations/gdc06-AdvancedPrototyping.ppt`.

[85] C. P Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proceedings of the 15th National Conference on Artificial Intelligence*, pages 431–437, 1998.

[86] S. A. Gregory. Design science. In *The Design Method*, volume 323, page 330. Butterworth, 1966.

[87] John Grey. The paper as it is (email), May 2012.

[88] G. Gunzelmann, R. Moore Jr, D. D. Salvucci, K. A. Gluck, et al. Sleep loss and driver performance: Quantitative predictions with zero free parameters. *Cognitive Systems Research*, 12(2):154–163, 2011.

[89] M. Hanus, H. Kuchen, and J. J Moreno-Navarro. Curry: A truly functional logic language. In *Proceedings of ILPS'95 Workshop on Visions for the Future of Logic Programming*, pages 95–107, 1995.

[90] Charles O. Hartman. *Virtual Muse: Experiments in Computer Poetry*. Wesleyan University Press, 1996.

[91] Jörg Hoffmann. FF: the fast-forward planning system. *AI magazine*, 22(3):57, 2001.

[92] D. R Hofstadter. *Fluid concepts and creative analogies: Computer models of the fundamental mechanisms of thought*. Basic Books, 1995.

[93] C. Holzbaur. OFAI clp (q, r) manual. Technical report, edition 1.3. 3. Technical Report TR-95-09, Austrian Research Institute for Artificial Intelligence, Vienna, 1995.

[94] V. Hom and J. Marks. Automatic design of balanced board games. In *Proceedings 3rd Artificial Intelligence and Interactive Digital Entertainment Conference*, pages 25–30, 2007.

[95] Ken Hullett. *The Science of Level Design: Design Patterns and Analysis of Player Behavior in FPS Levels*. PhD thesis, University of California, Santa Cruz, 2012.

[96] R. Hunicke, M. LeBlanc, and R. Zubek. MDA: a formal approach to game design and game research. In *Proceedings of the AAAI Workshop on Challenges in Game AI*, pages 1–5, 2004.

[97] R. Ierusalimschy, L. H De Figueiredo, and W. C Filho. Lua—an extensible extension language. *Software Practice and Experience*, 26(6):635–652, 1996.

[98] A. Jaffe, A. Miller, E. Andersen, Y.E. Liu, A. Karlin, , and Z. Popović. Evaluating competitive game balance with restricted play. In *Proceedings 8th Artificial Intelligence and Interactive Digital Entertainment Conference*, pages 26–31, 2012.

[99] John C. Jones. Designing designing. *Design Studies*, 1(1):31–35, July 1979.

[100] J. R Josephson. Abduction-prediction model of scientific inference reflected in a prototype system for model-based diagnosis. *Philosophica*, pages 13–18, 1998.

[101] Jesper Juul. The game, the player, the world: Looking for a heart of gameness. In *Level Up: Digital Games Research Conference Proceedings*, 2003.

[102] Jesper Juul. *Half-real: video games between real rules and fictional worlds*. MIT Press, December 2005.

[103] A. C Kakas, R. A Kowalski, and F. Toni. Abductive logic programming. *Journal of logic and computation*, 2(6):719–770, 1992.

[104] R. M Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum, 1972.

[105] H. Kautz, E. Horvitz, Y. Ruan, C. Gomes, and B. Selman. Dynamic restart policies. In *Proceedings of the National Conference on Artificial Intelligence*, pages 674–681, 2002.

[106] H. Kautz, B. Selman, and J. Hoffmann. SatPlan: Planning as satisfiability. *Booklet of the 2006 International Planning Competition*, 2006.

[107] John F. Kelley. *Natural Language and computers: Six empirical steps for writing an easy-to-use computer application*. PhD thesis, The Johns Hopkins University, 1983.

[108] Kristian Kersting. An inductive logic programming approach to statistical relational learning. In *Proceeding of the 2005 conference on An Inductive Logic Programming Approach to Statistical Relational Learning*, pages 1–228, 2005.

[109] T. W Kim, J. Lee, and R. Palla. Circumscriptive event calculus as answer set programming. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 823–829, 2009.

[110] S. O. Kimbrough, G. J. Koehler, M. Lu, and D. H. Wood. On a feasible-infeasible two-population (FI-2Pop) genetic algorithm for constrained optimization: Distance tracing and no free lunch. *European Journal of Operational Research*, 190(2):310–327, October 2008.

[111] R. D. King, K. E. Whelan, F. M. Jones, P. G.K. Reiser, C. H. Bryant, S. H. Muggleton, D. B. Kell, and S. G. Oliver. Functional genomic hypothesis generation and experimentation by a robot scientist. *Nature*, 427(6971):247–252, 2004.

[112] D. Kline and L. McHugh. The AI of Darkspore, October 2011.

[113] Raph Koster. A grammar of gameplay, 2005.

[114] Raph Koster. *A theory of fun for game design*. Paraglyph Press, Scottsdale, AZ, 2005.

[115] Bernd Kreimeier. The case for game design patterns. *Gamasutra*, March 2002.

[116] A. Krzeczkowska, J. El-Hage, S. Colton, and S. Clark. Automated collage generation-with intent. In *Proceedings of the 1st International Conference on Computational Creativity*, 2010.

[117] Thomas S. Kuhn. *The structure of scientific revolutions*. University of Chicago press, 1996.

[118] J. Kuittinen and J. Holopainen. Some notes on the nature of game design. In *Breaking New Ground: Innovation in Games, Play, Practice and Theory: Proceedings of the 2009 Digital Games Research Association Conference.*, 2009.

[119] Imre Lakatos. *Criticism and the Growth of Knowledge*, volume 4. Cambridge Univ Pr, 1970.

[120] P. Langley, G. L Bradshaw, and H. A Simon. Rediscovering chemistry with the BACON system. *Machine Learning: an artificial intelligence approach*, 1:307–329, 1983.

[121] Brenda Laurel. *Design research: Methods and perspectives*. MIT Press, 2003.

[122] Le Corbusier. *Towards a New Architecture*. Courier Dover Publications, 1931.

[123] D. B. Lenat and J. S. Brown. Why AM and EURISKO appear to work. *Artificial intelligence*, 23(3):269–294, 1984.

[124] Douglas B. Lenat. *AM: an artificial intelligence application to discovery in mathematics*. PhD thesis, Stanford Univeristy, July 1976.

[125] A. Liapis, G. N. Yannakakis, and J. Togelius. Computational caricatures: Probing the game design process with AI. In *Workshops at the Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2012.

[126] Vladimir Lifschitz. What is answer set programming. In *Proceedings of the 23rd National Conference on Artificial intelligence*, pages 1594–1597, 2008.

[127] H. Liu and P. Singh. ConceptNet practical commonsense reasoning tool-kit. *BT technology journal*, 22(4):211–226, 2004.

[128] Yu-Tung Liu. Creativity or novelty?: Cognitive-computational versus social-cultural. *Design Studies*, 21(3):261–276, 2000.

[129] George Loewenstein. The psychology of curiosity: A review and reinterpretation. *Psychological bulletin*, 116(1):75, 1994.

[130] N. Love, T. Hinrichs, D. Haley, E. Schkufza, and M. Genesereth. *General game playing: Game description language specification.* Stanford Logic Group Computer Science Department Stanford University, Technical Report LG-2006-01, 2008.

[131] Ada Lovelace. Sketch of the analytical engine invented by charles babbage. *Scientific Memoirs*, 3, 1842.

[132] P. Machado and A. Cardoso. All the truth about NEvAr. *Applied Intelligence*, 16(2):101–118, 2002.

[133] Lorenzo Magnani. *Abduction, reason, and science: Processes of discovery and explanation.* Springer, 2001.

[134] Lionel March. *Architecture of Form.* MIT Press, 1977.

[135] M. Mateas and A. Stern. A behavior language for story-based believable agents. *IEEE Intelligent Systems*, 17(4):39–47, July 2002.

[136] M. Mateas and A. Stern. Build it to understand it: Ludology meets narratology in game design space. In *Proceedings of the 2005 Digital Games Research Association Conference (DiGRA)*, Vancouver, Canada, June 2005.

[137] M. Mateas and N. Wardrip-Fruin. Defining operational logics. *Digital Games Research Association (DiGRA)*, 2009.

[138] Michael Mateas. Expressive AI. In *SIGGRAPH 2000 Electronic Art and Animation Catalog*, 2000.

[139] Michael Mateas. Expressive AI: a semiotic analysis of machinic affordances. In *3rd Conference on Computational Semiotics for Games and New Media*, page 58, 2003.

[140] John McCarthy. Elaboration tolerance. In *Common Sense*, volume 98, 1998.

[141] G. McGraw and D. Hofstadter. Perception and creation of diverse alphabetic styles. *AISB QUARTERLY*, pages 42–42, 1993.

[142] K. E Merrick and M. L. Maher. *Motivated reinforcement learning: curious characters for multiuser games*. Springer-Verlag New York Inc, 2009.

[143] C. Meyer and P. A. Papakonstantinou. On the complexity of constructing golomb rulers. *Discrete Applied Mathematics*, 157(4):738–748, 2009.

[144] Gavin S.P. Miller. The definition and rendering of terrain maps. *ACM SIGGRAPH Computer Graphics*, 20(4):39–48, 1986.

[145] George A. Miller. WordNet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

[146] M. Möller, M. Schneider, M. Wegner, and T. Schaub. Centurio, a general game player: Parallel, java-and asp-based. *Künstliche Intelligenz*, 25(1):17–24, 2011.

[147] R. Morris and L. Cherry. DC- an interactive desk calculator. *Bell Laboratories*, 1978.

[148] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535, 2001.

[149] S. Moyle and S. Muggleton. Learning programs in the event calculus. *Inductive Logic Programming*, pages 205–212, 1997.

[150] Erik T. Mueller. Event calculus reasoning through satisfiability. *Journal of Logic and Computation*, 14(5):703–730, 2004.

[151] Erik T. Mueller. *Commonsense reasoning*. Morgan Kaufmann, 2006.

[152] S. Muggleton, J. Santos, and A. Tamaddoni-Nezhad. Toplog: ILP using a logic program declarative bias. *Logic Programming*, pages 687–692, 2008.

[153] Stephen Muggleton. Inverse entailment and progol. *New generation computing*, 13(3):245–286, 1995.

[154] M. J. Nelson and M. Mateas. Towards automated game design. *AI*IA 2007: Artificial Intelligence and Human-Oriented Computing*, pages 626–637, 2007.

[155] M. J. Nelson and M. Mateas. Recombinable game mechanics for automated design support. In *Proceedings 4th Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*, pages 84–89, 2008.

[156] M. J. Nelson and M. Mateas. A requirements analysis for videogame design support tools. In *Proceedings of the 4th International Conference on Foundations of Digital Games*, pages 137–144, 2009.

[157] A. Newell, J. C. Shaw, and H. A Simon. Chess-playing programs and the problem of complexity. *IBM Journal of Research and Development*, 2(4):320–335, 1958.

[158] Allen Newell. The knowledge level: presidential address. *AI magazine*, 2(2):1, 1981.

[159] Richard A. O'Keefe. *The craft of Prolog*. MIT Press, 1990.

[160] David Oranchak. Evolutionary algorithm for generation of entertaining shinro logic puzzles. In *EvoApplications 2010*, pages 181–190, 2010.

[161] Jon Orwant. EGGG: Automated programming for game generation. *IBM Systems Journal*, 39(3.4):782–794, 2000.

[162] Aarati Parmar. *Formalizing elaboration tolerance*. PhD thesis, stanford university, 2003.

[163] A. Pease, D. Winterstein, and S. Colton. Evaluating machine creativity. In *Workshop on Creative Systems, 4th International Conference on Case Based Reasoning*, pages 129–137, 2001.

[164] Charles S. Peirce. On the logic of drawing history from ancient documents, especially from testimonies. In *Collected Papers of Charles Sanders Peirce*. Harvard Univeristy Press, 1931.

[165] Barney Pell. METAGAME: A new challenge for games and learning. *Heuristic Programming in Artificial Intelligence*, 3:237–251, 1992.

[166] D. P Ranjan, D. Tang, and S. Malik. A comparative study of 2QBF algorithms. In *The Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, 2004.

[167] O. Ray and A. Kakas. ProLogICA: a practical system for abductive logic programming. In *Proceedings of the 11th International Workshop on Non-monotonic Reasoning*, pages 304–312, 2006.

[168] Chris Remo. MIGS: far cry 2's guay on the importance of procedural content. *Gamasutra*, November 2008.

[169] M.J. Rentmeesters, W.K. Tsai, and K.J. Lin. A theory of lexicographic multi-criteria optimization. In *Engineering of Complex Computer Systems, 1996. Proceedings., Second IEEE International Conference on*, pages 76–79. IEEE, 1996.

[170] Graeme Ritchie. Assessing creativity. In *Proceedings of the AISB*, volume 1, 2001.

[171] H. W.J. Rittel and M. M. Webber. Dilemmas in a general theory of planning. *Policy sciences*, 4(2):155–169, 1973.

[172] J. M. Robson. The complexity of go. *Information Processing*, 83(413-417):38, 1983.

[173] N. F.M. Roozenburg. On the pattern of reasoning in innovative design. *Design Studies*, 14(1):4–18, 1993.

[174] R. Rouse and S. Ogden. *Game design: theory & practice*. Jones & Bartlett Learning, 2005.

[175] Chiaki Sakama. Induction from answer sets in nonmonotonic logic programs. *ACM Transactions on Computational Logic (TOCL)*, 6(2):203–231, 2005.

[176] K. Salen and E. Zimmerman. *Rules of play: Game design fundamentals*. MIT Press, 2004.

[177] C. Salge, C. Lipski, T. Mahlmann, and B. Mathiak. Using genetically optimized artificial intelligence to improve gameplaying fun for strategical games. In *Proceedings of the 2008 ACM SIGGRAPH symposium on Video games*, pages 7–14, 2008.

[178] Rob Saunders. *Curious design agents and artificial creativity*. PhD thesis, University of Sydney, 2002.

[179] Jesse Schell. *The Art of Game Design: A book of lenses*. Morgan Kaufmann, 2008.

[180] S. Schiffel and M. Thielscher. Fluxplayer: A successful general game player. In *Proceedings of the National Conference on Artificial Intelligence*, volume 22, page 1191, 2007.

[181] Jürgen Schmidhuber. Formal theory of creativity, fun, and intrinsic motivation (1990–2010). *Autonomous Mental Development, IEEE Transactions on*, 2(3):230–247, 2010.

[182] Donald A. Schön. *The reflective practitioner: How professionals think in action*. Basic books, 1983.

[183] Donald A. Schön. Designing: rules, types, and worlds. *Design Studies*, 9:181–190, July 1998.

[184] C. Schulte, M. Lagerkvist, and G. Tack. GECODE: Generic constraint development environment, 2006. `http://www.gecode.org/`.

[185] J. Secretan, N. Beato, D. B. D'Ambrosio, A. Rodriguez, A. Campbell, and K. O. Stanley. Picbreeder: evolving pictures collaboratively online. In *Proceedings of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, CHI '08, pages 1759–1768, New York, NY, USA, 2008. ACM.

[186] M. Shanahan and M. Witkowski. Event calculus planning through satisfiability. *Journal of Logic and Computation*, 14(5):731–745, 2004.

[187] Murray Shanahan. *Solving the frame problem: a mathematical investigation of the common sense law of inertia*. MIT press, 1997.

[188] Murray Shanahan. The ramification problem in the event calculus. In *International Joint Conference on Artificial Intelligence*, volume 16, pages 140–146, 1999.

[189] Tyler Sigman. The siren song of the paper cutter: Tips and tricks from the trenches of paper prototyping. *Gamasutra*, 2005.

[190] Herbert A. Simon. *The Sciences of the Artificial*. MIT Press, first edition, 1969.

[191] Ruben Smelik. *A Declarative Approach to Procedural Generation of Virtual Worlds*. PhD thesis, TU Delft, 2011.

[192] A. M. Smith, E. Andersen, M. Mateas, and Z. Popović. A case study of expressively constrainable level design automation tools for a puzzle game. In *Proceedings of the International Conference on the Foundations of Digital Games*, 2012.

[193] A. M. Smith, C. Lewis, K. Hullett, G. Smith, and A. Sullivan. An inclusive taxonomy of player modeling. Technical Report UCSC-SOE-11-13, University of California, Santa Cruz, 2011.

[194] A. M. Smith and M. Mateas. Variations Forever: Flexibly generating rulesets from a sculptable design space of mini-games. In *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG)*, 2010.

[195] A. M. Smith and M. Mateas. Answer set programming for procedural content generation: A design space approach. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(3):187–200, 2011.

[196] A. M. Smith and M. Mateas. Knowledge-level creativity in game design. In *Proceedings of the 2nd International Conference in Computational Creativity (ICCC) 2011)*, 2011.

[197] A. M. Smith and M. Mateas. Towards knowledge-oriented creativity support in game design. In *Proceedings of the 2nd International Conference in Computational Creativity (ICCC 2011)*, 2011.

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

[199] A. M. Smith, M. J. Nelson, and M. Mateas. Ludocore: A logical game engine for modeling videogames. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, pages 91–98, 2010.

[200] A. M. Smith, M. Romero, Z. Pousman, and M. Mateas. Tableau Machine: A creative alien presence. In *AAAI Spring Symposium on Creative Intelligent Systems*, pages 82–89, 2008.

[201] G. Smith, A. Othenin-Girard, and J. Whitehead. PCG-Based game design: Creating endless web. In *Proceedings of the International Conference on the Foundations of Digital Games*, 2012.

[202] G. Smith, J. Whitehead, and M. Mateas. Tanagra: Reactive planning and constraint solving for mixed-initiative level design. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(3):201–215, 2011.

[203] G. Smith, J. Whitehead, M. Mateas, M. Treanor, J. March, and M. Cha. Launchpad: A rhythm-based level generator for 2D platformers. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(1):1–16, 2011.

[204] N. Sorenson and P. Pasquier. Towards a challenge-based model of pleasure in video games. In *ICCC-X: First International Conference on Computational Creativity*, 2010.

[205] N. Sorenson and P. Pasquier. Towards a generic framework for automated video game level creation. *Applications of Evolutionary Computation*, pages 131–140, 2010.

[206] N. Sörensson and N. Eén. MiniSat 2.1 and MiniSat++ 1.0 — SAT race 2008 editions. *SAT 2009 competitive events booklet: preliminary version*, page 31, 2009.

[207] Ashwin Srinivasan. The aleph manual. *Machine Learning at the Computing Laboratory, Oxford University*, 2001.

[208] R. M. Stallman, R. McGrath, and P. Smith. *GNU make*. Free Software Foundation, Boston, 1988.

[209] Eskil Steenberg. The pivot model, January 2012.

[210] L. Sterling, E. Shapiro, and M. Eytan. *The art of Prolog*, volume 94. Wiley Online Library, 1986.

[211] P.R. Suaris and G. Kedem. An algorithm for quadrisection and its application to standard cell placement. *IEEE Transactions on Circuits and Systems*, 35(3):294–303, 1988.

[212] Ivan Sutherland. *Sketchpad: A Man-Machine Graphical Interface*. PhD thesis, MIT, 1963.

[213] S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *Transactions on Computational Intelligence and AI in Games*, 2012.

[214] Michael Thielscher. Answer set programming for single-player games in general game playing. *Logic Programming*, pages 327–341, 2009.

[215] Michael Thielscher. A general game description language for incomplete information games. In *Proceedings of AAAI*, pages 994–999, 2010.

[216] J. C. Thomas and J. M. Carroll. The psychological study of design. *Design Studies*, 1(1):5–11, 1979.

[217] J. Togelius, R. De Nardi, and S. M. Lucas. Towards automatic personalised content creation for racing games. In *Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on*, pages 252–259, 2007.

[218] J. Togelius, T. Justinussen, and A. Hartzen. Compositional procedural content generation. In *Proceedings of the FDG Workshop on Procedural Content Generation (PCGames'2012)*, 2012.

[219] J. Togelius and J. Schmidhuber. An experiment in automatic game design. In *Computational Intelligence and Games, 2008. CIG'08. IEEE Symposium On*, pages 111–118, 2008.

[220] J. Togelius, G. Yannakakis, K. Stanley, and C. Browne. Search-based procedural content generation: A taxonomy and survey. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(3):172–186, 2011.

[221] Julian Togelius. Multiobjective exploration of the StarCraft map space. In *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG)*, 2010.

[222] M. Treanor, B. Blackford, M. Mateas, and I. Bogost. Game-O-Matic: Generating videogames that represent ideas. In *Procedural Content Generation Workshop at the Foundations of Digital Games Conference*, 2012.

[223] Alan M. Turing. Computing machinery and intelligence. *Mind*, 59(236):433–460, 1950.

[224] Sherry Turkle. *Life on the Screen: Identity in the Age of the Internet*. Simon and Schuster, 1995.

[225] T. Tutenel, R. Bidarra, R. M. Smelik, and K. J. de Kreker. Rule-based layout solving and its application to procedural interior generation. In *Proceedings of the CASA Workshop on 3D Advanced Media in Gaming and Simulation*, 2009.

[226] N. Wardrip-Fruin. *Expressive Processing: Digital fictions, computer games, and software studies*. MIT Press, 2009.

[227] Joseph Weizenbaum. *Computer Power and Human Reason*. Penguin Books Ltd, January 1984.

[228] Jan Wielemaker. An overview of the SWI-Prolog programming environment. In *Proceedings of the 13th International Workshop on Logic Programming Environments*, pages 1–16, 2003.

[229] Xin-She Yang. *Nature-Inspired Metaheuristic Algorithms*. Luniver Press, 2008.

[230] W. K Yeap, T. Opas, and N. Mahyar. On two desiderata for creativity support tools. In *Proceedings of the International Conference on Computational Creativity (ICCC)*, pages 180–189, 2010.

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

[232] L. Zhang and S. Malik. Towards a symmetric treatment of satisfaction and conflicts in quantified boolean formula evaluation. In *Principles and Practice of Constraint Programming-CP 2002*, pages 313–318, 2006.

[233] A. Zook, S. Lee-Urban, M.O. Riedl, H.K. Holden, R.A. Sottilare, and K.W. Brawner. Automated scenario generation: Toward tailored and optimized military training in virtual environments. In *Proceedings of the International Conference on the Foundations of Digital Games*, pages 164–171. ACM, 2012.