The Intelligent Game Designer: Game Design as a New Domain for Automated Discovery

Adam M. Smith
Department of Computer Science
University of California, Santa Cruz
amsmith@cs.ucsc.edu

July 25, 2009

Abstract

Designing video games is commonly understood to be a creative task, drawing on a designer's talent, inspiration, and personal experience. The last ten years have seen multiple calls from the design community to produce reusable knowledge about the structure of games and the design process itself. These designers would like to establish standardized languages and libraries of design patterns so that the next generation of designers can benefit from the best of past generations. The realization of such a move can be read as a transition from thinking about game design as a playable-artifact creation process to a *science of play* in which we might see the designer's goal as discovering new gameplay structures and their production of concrete games as a side effect of this process.

Thirty years ago, a similar-yet-disconnected thread of research in artificial intelligence was just being born. First marked by Doug Lenat's AM (an "automated mathematician"), *discovery systems* aim to automatically produce new and interesting knowledge. Such systems contrast sharply with the then-popular expert systems which applied *fixed* libraries of "expert" knowledge to various tasks. Discovery systems, which have commonly operated in the domains of natural science and mathematics, are now seen as distant ancestors of contemporary, statistical *machine learning* techniques which find extensive application in a wide array of industries. Contrary to the current emphasis on the optimal learning of statistical descriptions of data, some recent developments in machine learning, specifically combined abductive and inductive logic learning systems, are bringing the production and revision of structured, symbolic knowledge back into focus. Simultaneous research in computational creativity is making inroads into modeling the creative process and the production of creative artifacts.

This is the question I aim to answer: If we squint a bit to see game design as the science of play that some designers imagine it to be, can we build a discovery system that really works in the domain of game design? *Can we build an intelligent game designer?*

In my thesis proposal I lay out a plan to build an intelligent game designer that learns from the process of game design (including the observation of human players) and exports newly discovered design knowledge. This will require an operationalization of game design as an automatable, scientific process and a detailed re-synthesis of the creative design of expressive artifacts as a knowledge-seeking effort.

1 Introduction

Proving theorems and playing chess were once considered shining examples of activities so abstract and sublime that they would be forever off limits to the number-crunching brains of computing machines. When we observed a machine performing these tasks previously only approachable by humans, we were quick to reclassify them as routine and uncreative. This phenomenon is of no surprise for those working on computational creativity where it is a central tension in their research.

Consider the design of games (familiar, flashy video games): they elicit the subjective audience reactions of paintings in their display, the rich temporal structure of music in their simulated worlds, and the textual

structure of written language in their code. Like a natural science, the unobservable structure of the thought process of a human player must be reconstructed from elementary observations, and, like mathematics, the properties of intangible, imaginary objects must be cataloged and explained. On top of this, games have a sense of deep, live interaction that none of these other fields address. If there were a machine that could address all of these in one package, would we still write it off as uncreative?

In this thesis proposal I will describe my plan to develop a system that will simultaneously address as many of these issues as possible. I propose building a machine that can design games, one that will design games intelligently.

High aims are not enough. In Bruce Buchanan's AAAI-2000 presidential address [4], after reviewing the impressive output of the day's creative intelligent systems, he raised some serious criticisms: "(1) they do no accumulate experience, and thus, cannot reason about it; (2) they work within fixed frameworks including fixed assumptions, and criteria of success; and (3) they lack the means to transfer concepts and methods from one domain to another." The depth of my thesis comes in how I plan overcome these issues, admittedly, within the specific domain of game design. The primary function of my intelligent game designer, in fact, is to turn game design experience into communicable knowledge. Doing this will require an operationalization of game design as an automatable, scientific process and a detailed re-synthesis of the creative design of expressive artifacts as a knowledge-seeking effort.

While I hope to discover something new about creativity in general, this thesis will also have lessons for the various fields it touches: game design, game development, game studies, and several fields within artificial intelligence (namely expressive intelligence, automated discovery, and computational creativity). In the remainder of this introduction I will situate my project in the context of other research narratives, review the core research questions and proposed contributions of my thesis, and provide a road-map to the larger proposal.

1.1 Context

I see my work at a sort of intersection of two roads that are largely oblivious to one another. On the first, there is a progression of the practice of game design as an informal, even mystical art through a practice of machine-assisted design to fully automated game designing systems. On the second, there is a progression from sterile, domain-agnostic machine learning methods, through both discovery systems working in various fields and creative systems that exhibit interesting behavior or produce interesting artifacts to creative learning systems that produce theories as well as meaningful artifacts in specific domains. These are not chronological progressions, but spaces of perspectives from which work can be done.

Just outside of the informal extreme of the game design spectrum, there are perspectives that allow a designer to believe that there are patterns, visible to others, to which that they might want access. In this region, designers may swap stories about patterns in players, their games' rules, and design processes. Certainly, sharing such knowledge does not entail giving up any of one's expressive power. Learning from another's mistake or building on another's success provides the incentive to externalize design knowledge via informal means. Many game design textbooks, forum posts, and technical talks are situated in a perspective from this region.

In the middle of the progression we find elements like procedural content generation and drama management systems – processes which only work because human designers relinquish some of their creative control to the machine, entrusting it with a primitive understanding of small design tasks here and there. The lure of reducing combined development effort provides the incentive to take informal experiences and codify them as reusable, formal knowledge. Tool, middle-ware, and engine builders from industry as well as games research operate from perspectives in this region, sharing binaries.

Towards, but not actually at, the automated design extreme, is my proposed system. My system should be able to consume structured knowledge and contribute some back, all the while producing games. However, given any number of realistic limitations, I hesitate to call my system fully automatic. From an industry perspective, a system with no room for human input is largely uninteresting anyway. My system will be informed by the shared ideas of other game designers, and by the tools coming out of games research, but it will best be understood from a perspective at the intersection of both roads.

Turning to the sterile-to-domain-rich spectrum, we find a range of perspectives that largely share one goal: to produce interesting and relevant artifacts. Clearly there is some stretching here to put tuned statistical distributions, electronic paintings, and conjectures about integer sequences into one spectrum of artifacts, but the relationships become clear if we look a little deeper.

Even the most abstract machine learning algorithms take in some model of the environment (even if this is just the number of elements to expect in a fixed-size vector representation) and some data then produce a prediction or predictive model of that data. In such an abstract environment, there is hardly anything else that might be considered an interesting, relevant artifact for output.

Moving to discovery systems, which have commonly worked in natural science or mathematics, there is a much richer description of the environment and the data coming in may be only partial evidence of the workings of an external system (for which even the basic structure is unknown). In such a setting, raw predictions and predictive models are interesting and relevant artifacts. However, physical configurations that to not form any sort of prediction are still interesting for the patterns of data which they spawn. Through experimentation, discovery systems can focus their interest on data which is more likely to inform their internal theories (which form the basis for their predictions).

Creative art systems, which appear to have no relationship with machine learning systems, can be seen as performing, in isolation, something akin the the experimental design process in discovery systems. After all, we are impressed by a creative art system that is able to elicit strong reactions from its audience (which in turn feed back new data patterns to the system). There are creative art systems that accept and meaningfully integrate feedback on their "experiment"; however, in many cases, this feedback is little more than a heavily abstracted goodness-number, between zero and one. Even operating on a hard-coded description of the environment, the artistic artifacts produced by creative art systems are generally much more interesting than physical experiments produced in a scientific discovery systems. I claim this is because these art systems are much more in-tune with the mental models of their audience, even if this understanding is static.

Mashing together the best of discovery systems and creative art systems brings us into a realm of systems that incrementally come to understand their environment better, but apply a useful model of how their experiments interact with the environment (including the human audience). No scientific discovery system has yet considered "audience reaction" to its experimental designs, but with the existence of the fields of science ethics and science education, we can clearly see that there is indeed an opinionated audience for experiments in science. If there were a collection of systems that were designed from a perspective at this end of the spectrum, I would be happy to place my own system at an unnamed fringe of it.

In my review of related work later in this proposal, I will cover established practices on both the game design and machine learning roads described above. It is my hope to populate a new point in both spectra, taking advantage of unexpected synergies along the way.

1.2 Research Questions and Contributions

In building an intelligent game designer, there is certainly a primary question: How should such a system function? Before launching into a detailed discussion of research questions, I should quickly rule out an important non-answer. Building a computer program that successfully generates games is not enough to answer this. As a sufficiently interesting set of games could always be crammed into a fixed table, I should look for something more than game generation. Human game designers do more than *just* produce playable artifacts, and my system should as well.

I have two references for *real* intelligent game design: successful, professional designers, on one hand, and my own amateur game design experience on the other. Beyond generation, real game designers can meaningfully analyze games made by others, use experience from observing players to improve designs, use experience from the design process itself to inform future design choices, and leverage existing design knowledge to produce and share new knowledge. My system should address this extended space of activities.

- **Primary research question:** How does an intelligent game designer function?

Unpacking this a bit, I am looking for an explanation of how any intelligent game designer (one that performs the activities above) functions, and I would like this explanation to be a formal, computational explanation. Ideally, a functioning system will be a literal embodiment of such an explanation. A single system, however, is not enough. This explanation should describe how related systems function as well (a black-box system is not acceptable).

The sense of "games" that I am interested is one of single-player, real-time, mechanics-heavy games that are easily recognizable by experienced players as "video games" (though the level of graphical and other details need not be competitive with recent titles). I acknowledge that elements such as narrative and meta-gaming are pervasive in modern games; however, as a simplifying assumption, I choose not to consider them in this thesis.

The sense of "intelligent" that I am interested in is a qualifier that stands in for having the system not "cheat" in any obvious way such as using extensive lookup-tables, proxying to human assistants, or other subterfuge while producing effective design knowledge. The knowledge produced and exported should be learned from design experience and documented accordingly (a simple log of theory revision choices will suffice).

Finally, "game design" deserves some unpacking too. In industry, the term "designer" is used to describe a range of roles from non-technical writing through complete game development. In all cases, the role is one that involves making decisions about a game's core mechanics. As, stripped of all of their window dressings, a game's mechanics are embodied by its rule system, I will define "game design" in this project to mean the informed construction of rule sets and supporting logic needed to produce playable games. Though I am excited by the prospect of a system that performs end-to-end game development, I understand that some core designs may need human polish before they are complete. This is similar to how, in automatic music composition, it is enough for a system to compose MIDI data (which can be played directly, resulting in a mechanical sounding performance), while a human orchestra is often needed to expose the true nature of a piece to an uninitiated audience.

- **Secondary research question:** What does such a system imply for the relationship between discovery and expressive domains?

Concretely, this translates into a more direct question when looking to examine game design: How can game design be re-conceptualized to make it into a suitable domain for discovery? An answer to this question should also answer several related, lower-level questions: What are the discovery activities in game design? What do conjectures apply to? How are experiments carried out (in terms of environments, observations, and instruments)? How are conjectures verified? And is there a sense of proof in game design that mirrors that of mathematics?

Looking to reexamine discovery, there is the reverse question: How can discovery be re-conceptualized to make it applicable to game design? Again, answers to lower-level questions should fall out of this: What sort of discovery role do activities like prototyping and play testing fill? What does it mean to publish a game? Where does fun fit in? And what ensures that "games" are produced instead of abstract state-progression systems?

From answers to these questions, I should be able to say more about the relation between discovery and expressive domains. Considering music, narrative, visual art, or poetry, how are these like a science or like mathematics? Do the notions of author and audience gain a clear meaning in science? What role do affordances play in discovery domains?

While neither of my big research questions explicitly mention creativity, I claim these are still good questions to ask as part of a computational creativity research program. In the context of discovery applied to expressive domains, I am interested in how much creative activity can be understood as a rational pursuit of knowledge.

1.3 Proposal Overview

In the remainder of my proposal I will provide a detailed tour through related research (split between the game design and discovery/creativity system lines), lay out my prior work towards the goal of this thesis

(from preliminary experience formalizing an expressive domain through performing elementary discovery in game design myself using the same tools as my imagined systems), and finally propose a plan for the work that must be done to complete it (including a snapshot of my personal background theory, a space of possible systems, and concrete experiments designed to validate my work, concluding with a two year time line).

2 Related Work

In the discussion of related work below, I aim to sketch out a basis for the two roads I described in my introduction. While I usually imagine these two areas to be largely disjoint, it is interesting to see that artificial intelligence becomes tangled in both sides.

2.1 Game Design and the Science Thereof

Turning to game design first, I should admit that I intend to take the reader on a bit of a ride in this section. In the interest of clarity, I will spoil the story up front. First, I would like to paint a picture of game design as a fairly clear-cut, textbook-documented activity. Then, I will pop this bubble with some big questions, and find evidence that game designers have been asking these questions for long time. Next, I will show how game studies has responded to these questions by producing a detailed (but not yet executable) understanding of games. Finally, I explain how research in artificial intelligence has addressed similar questions, though with little awareness of progress in game studies.

2.1.1 By The Book

Game design is no mystery; it is even taught in schools (in my own department no less[1]). Recent textbooks such as *Game Design Workshop* [10] or *Rules of Play* [32] aim to prepare students to produce the same kind of video games produced in industry; though there is often a bias towards preparing students to take the creative leap to produce innovative gameplay (with a strong emphasis on understanding play itself) instead of simply reproducing past successes. Most game design education programs acknowledge that game design involves much more than programming and that while it builds strongly on a computer science background, it also draws on elements of art, music, narrative, and digital media.

I will briefly review some of the core topics in game design as it documented so that I can reference these items as a develop a means of automating game design. At a very high level, students of game design are introduced to various roles they may fill in industry, a set of constructive elements they may produce, and a set of practices they should follow when designing games.

Amongst a sea of roles such as artist, writer, designer, programer, publisher, producer, director, and tester, I have selected the role of "designer" to provide the basis for my system. Designers are the individuals responsible for constructing or coordinating the a game's rules and other structures that impact gameplay. In industry, often the job of turning the high level rules of a game into rigid software is pushed to dedicated programmers; however, there are many instances of designer-programers who are capable of designing all the way to the code level. The title of "developer" is usually reserved for for individuals or teams that cover all of the roles in producing complete, marketable games (which I do not intend to address in this thesis).

Games are dynamic systems with state that evolves over time in response to player input. The dynamics of these systems are shaped by a game's rules. For table-top games, rules written in natural language may be subject to player interpretation (much the same way that contract language or "legalese" is open haggling). In contrast, the rules for computer games are absolute and rigid: the executable binary encoding of the rules is dispassionately enforced by the machine's hardware. The rich interaction between elements of a game's rule set are what create the detailed state spaces that players explore in their interaction with a game. Towards shaping gameplay, designers pay very close attention to a game's rules.

In isolation, a game rule may not tell us much about how a game might be played. One rule might read (when translated into conversational language) "being hit by a level-k fireball causes a fighter's health to

[1] `http://www.cs.ucsc.edu/game-design`

be reduced by k units if no damage reduction items are possessed." To interpret this rule we have to read the rules governing the movement of fireballs, the association of health to fighters, and classes of objects which might count towards the reduction of damage. Groups of related rules form coherent *game mechanics* that are recognizable during play. The set of mechanics that describe the most pervasive elements of play in a game are called the *core* mechanics of that game. Every game has mechanics, and often describing the core mechanics of a game is a good way to communicate a game's design without detailing its complete implementation. Furthermore, many games are classified into genres by their core mechanics (though this is not the only way to classify games).

Outside of a game's mechanics, designers make decisions about other structures in the game that help players come to understand the mechanics over time. In traditional computer games, graphics and audio expose (parts of) the state of a game to players while devices like the keyboard and mouse collect player input to modify that state. Within the realization of this interface, designers are free to add elements which to not impact the game as a mathematically-defined dynamic system, but nonetheless deeply shape player experience. Two important elements in this class are characters and narrative. Games may exist without them, but it seems that at least some element of story (even if just a brief mention of setting) is critical to the player's understanding of a rule set as a game instead of an abstract logic puzzle (or an unattractive piece of productivity software).

If game design is about producing collections of mechanics with associated player-assisting glue, designers must have some practices to bring these into existence. Three practices stand out clearly in the literature: prototyping, play testing, and tuning.

Prototyping is the process of building smaller or simpler versions of an artifact of interest with the aim of getting feedback on the larger design without the work of producing the complete artifact. In the design of computer games, a particularly low-effort way of prototyping games is to produce a paper prototype. Paper prototypes look like board games and may have soft, incomplete rule sets (new rules may even be invented as the game is played). By recycling physical objects already in hand (such as graph paper, dice, paper clips, etc.) a game design idea can even be turned into a playable paper prototype on the fly. Lessons learned from paper prototypes should not always be trusted, though, as players of the prototype often have far more leeway and access to game state than they would in a computer version of the same game. Because most games have a somewhat meaningful paper version, I have adopted a board-game-like interface metaphor in a tool that I will discuss later in coverage of my prior work.

Computational prototypes are complete computer programs. These programs usually implement the rigid rules of a game's core mechanic; however, they may elide auxiliary rules (including victory conditions) or employ austere player interfaces (including the use of "programer art"). Computational prototypes can be iteratively refined and elaborated to the point where they approximate complete games, even including tentative narrative elements in some cases. Along the way, however, prototypes need not be, by various definitions, games at all. They may simply be partially specified dynamic systems in which the designer aims to externalize some mechanic in an executable manner, possibly even running without any user interaction. The games I imagine my system producing will be most recognizable as computational prototypes.

Although simply the act of building a prototype can give a designer some form of feedback (in that the rules they imagine may be particularly easy or hard to codify), the primary source of feedback a designer gets comes from play testing. Play testing involves literally playing through the a game (or prototype) and observing player actions and reactions during this experience. There are roughly three forms of play testing: self-testing, testing with friends, and testing with the target audience. In self-testing, a designer (possibly in concert with their designer peers) plays a game himself. The rules of a game may be vastly underspecified as the testers are assumed to use their own design knowledge to fill in plausible rule extensions. Testing with friends, though it may have a social cost, gives much better feedback as friends will approach a game from a player perspective instead of a designer perspective. Friends tend to be relatively tolerant of the minimal, temporary interfaces used in early prototypes, however they may not think the same way as the target audience, making different decisions during play. Finally, testing with the target audience is often a very expensive practice. For feedback from the target audience to be accurate, the game under test must be nearly finished (which usually implies it is in a difficult-to-modify state). In some of my prior work I have

built tools to assist play testing with early-stage computational prototypes.

The final stage of the game design process is tuning. Tuning focuses on making small, quantitative changes to games (such as tweaking numerical constants or adding and removing bulk content, but not rules). Tuning involves integrating feedback from play testing at a stage where the core mechanics of a game have long been fixed. I do not expect to address game tuning in my thesis.

Given the above background, a student of game design can understand a progression from coarse prototypes to fully polished games, and that feedback from their designer peers, friends, and others will help guide the process down an attractive path. But where do these prototypes come from in the first place? How does one make changes to them in response to feedback without breaking them? Software engineering practice can help control the design process only at a software level, but at the level of rules and mechanics above, some other kind of knowledge is needed. The official word from textbooks of game design is that this knowledge comes from experience.

2.1.2 A Call for Structure

Saying that game design knowledge comes from experience is not enough for some game designers. In the introduction to his *400 Project*[2], a project to collect a large number of unexpressed, informal rules for game design, Hal Barwood writes the following.

> "Game design is an uncertain and murky endeavor. In this it resembles art, architecture, writing, moviemaking, engineering, 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 simpler approach?"

The rules collected so far, while indeed quite informal, do suggest that there is a wealth of design knowledge that can be shared. Unfortunately, towards automating game design, most of the rules in the *400 Project* are difficult to imagine mapping onto machine. One of the more promising rules reads "Make Subgames: Players want to participate in the course they take through your game – so give them plenty of opportunities to voluntarily take up ancillary challenges."

Barwood is not alone, nor is his bias towards informal knowledge universal. In *Formal Abstract Design Tools*[3], 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 that 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 computer games.

Later, in *The Case for Game Design Patterns*[4] 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 patterns in game design. He goes on to define a schema to which such patterns should conform and several examples. While Kreimeier's patterns are only modestly closer to code than Barwood's, it is interesting that they are structured in the form of problem-and-solution pairs, suggesting a sort of proto-formalism for case-based reasoning in game design.

From these calls we can see that designers are deeply interested in obtaining design rules, patterns, and other knowledge. However, neither producing such knowledge in a sharable form nor applying knowledge produced by others is a core practice in textbook game design. These proposals call for game designers to perform this knowledge engineering task themselves (and part of my thesis is a proposal for how this might be done at the code level), but, in the remainder of this section, I will show how researchers in game studies and artificial intelligence have stepped in to preform similar tasks.

[2] `http://www.finitearts.com/Pages/400page.html`
[3] `http://www.gamasutra.com/features/19990716/design_tools_01.htm`
[4] `http://www.gamasutra.com/features/20020313/kreimeier_01.htm`

2.1.3 Game Studies

The field of game studies works from an interdisciplinary perspective, considering combinations of background theory from social science, humanities, engineering, and others. One game studies researcher, Jesper Juul (a self-proclaimed "ludologist"), adopted a perspective similar to the ones designers had been calling for when he produced a structured, historical account of the development of match-three games[5]. His analysis focused on the addition and mutation of game mechanics over time. By naming the mechanics he was able to distinguish a sparse set of informative relations instead of arguing wild differences between all pairs of games. This analysis represents an example application of the kind of knowledge I am interesting in having my system discover.

Following up on Kreimeier's call for design patterns, Staffan Bjork published an entire book [1] of structured design patterns with detailed relations between them. The book proposes a language of patterns (quite similar to design patterns in software engineering). He admits that his pattern library cannot help with the "interestingness" of a game, but it can help express ideas to others in a concise, structured way (just what designers were looking for). His discussion ignores the goodness of patterns, focusing instead of describing common patterns that arise in real games (developing a kind of positive theory of of game structures instead of a normative theory). Design patterns associated with bad gameplay experience are useful, he claims, for analyzing faults as well as allowing novel uses of these patterns in in a "good" way.

In the *Game Ontology Project* [40], structured design patterns are hierarchically organized into an ontology with prototypical examples (and the result is visible as an open-access wiki[6]). In an example from the "gameworld rules" category reads "Savepoints are specific (non-random and predetermined) places or moments in a game where a players progress in that game is stored for the purpose of allowing the player to resume the game from that point at a later occasion." While they are written crystal clear English, these design patterns do not have a clear mapping to the kind of code-level knowledge an automated design system would work with. This is not surprising, however, as doing so is not part of the goal of any of the projects considered so far in game studies.

The knowledge brought to light in the field of games studies, especially when it is of an engineering nature, can bring insight to human game designers, but something more formal is needed before a designer's tools can understand and apply this knowledge on their behalf.

2.1.4 Artificial Intelligence for Games

In artificial intelligence, where the symbolic manipulation of knowledge by machines is the norm, bits of game design knowledge are being used in the production of game generators and general game players. Until recently, with growing interest in game design support tools, artificial intelligence had only minimal alignment with knowledge production and application efforts by game designers. I should note that in this section I am not interested in the applications of artificial intelligence to controlling game characters within the realm of well-known, fixed game rules.

Looking first at game generation, there are several systems that, in one sense or another, applied fixed elements of design knowledge to the automatic design of playable games. One key example is the

Extensible Graphical Game Generator (EGGG) [27]. EGGG is able to take minimal specifications for a game and transform these into complete, playable games with rich graphical interfaces and natural language documentation. The system accomplishes this by apply expert-system-like rules to expand a single input definition into a single output game.

While EGGG is only capable of producing a single game at a time, other game generators encode a generative space of games that can be sampled from in bulk. One interesting example is an application of the HR discovery system to puzzle generation [5]. In this application, the system is given a model what makes a good puzzle in terms of a generation template and several heuristics for identifying bad puzzles that might arise. The system is capable of producing a large array word and number puzzles.

[5] `http://www.jesperjuul.net/text/swapadjacent/`

[6] `http://www.gameontology.org`

Towards automating content generation in popular video games, other research at the Expressive Intelligence Studio have looked at generating levels for 2D platformer games. In a recent paper, a rhythm-based conception of level design is coupled with designer style profiles to produce a level generator that is parametrized by a designer's preferences [37].

In *Towards Automated Game Design* [24], Mark Nelson defined a factoring of game design into four areas: abstract mechanics, concrete representation, thematic (real-world) references, and player input. His paper illustrates a system that works in the realm of thematic references, leveraging a common sense knowledge database. Applying this factoring to other projects, the puzzle generation project focuses on thematic references as well, while the platform level generation is primarily a matter of concrete representation. In contrast, my thesis focuses on abstract mechanics.

In a bold experiment in automatic game design, Raph Koster's *Theory of Fun* [16] was applied to the design of simple games [38]. Koster's theory links a player's experience of fun in games to their process of learning that game, and Togelius' system links learning to game rules using genetic algorithms and reinforcement learning. While the invocation of the theory of fun is quite impressive, the genetic representation used in the system locked it into designing only elementary Pacman-like games. The author admits that the project is just a proof-of-concept in automated game design, and that it primarily exists to point at an unexplored area of research. My thesis aims to pick up this thread, however I feel there is much to be learned about how designer manipulate a games rules beyond genetic representations before we can return to the theory of fun in a principled manner.

Outside of game generation, but still assuming a game's rules are not known in advance, general game playing has grown out of the longer tradition of AI-chess. General game players (GGPs) are conventionally software agents which, given the rules to a completely unfamiliar game and a small amount of time to practice playing, are expected to perform well in competition against other agents. GGPs must efficiently reason about novel game mechanics in order to gain control over the dynamic system they define. In this sense, GPPs are likely to be as hungry for design knowledge as designers are themselves.

The most common way of describing the rules of a game to a GGP is to encode the rules in the Game Description Language (GDL)[7]. GDL requires the game designer to represent the game as a literal state transition system. Fortunately, no industry game designers have been asked to use GDL, as it is a language far more friendly to machine players than it is to human designers. The language defines a basic vocabulary for describing the rules of a game in terms of a set of logical predicates that include a means to name the (player) roles of a game, the set of possible moves and when these move are legal, and under which conditions a state is judged to be a terminal or goal state (and what the value is for that state, such a victory or loss). GGPs can use standard logical reasoning tools to reason backwards from high-valued goals, through legal moves, to particular actions to be taken at each step, and the goal of general game playing research is to improve this process. Historically, there has been an emphasis on symmetric chess-like games, however GDL has been applied to games with many players that involve a mixture of competition and collaboration between players. In my prior work I have proposed a game modeling language which is comparable to GDL in may ways.

Motivated by the need to find representations for state spaces that are efficiently searchable, very recent work with GDL has looked at being able to factor large games into smaller, largely independent subgames that can be searched quickly [11]. This task involves recognizing familiar games in new ones, which is one of the same tasks designers inherently do when they analyze a game made by a peer.

Finally, artificial intelligence has been applied to the development of direct game design assistance tools. In more work at the Expressive Intelligence Studio, professional game designers have been the subject of a requirements analysis for game design workbench [25]. In this requirements analysis, designers were exposed to various tools which could provide them with detailed objective answers to a certain set of questions about their designs. In particular, these tools, which assumed a mechanics-only representation of games, used inference over a temporal logic called the event calculus (discussed later in my prior work) to find abstract gameplay traces which meet a designers constraints or prove that none exist. I see my thesis in line with this work; however, I plan to contribute by to focus building a solid theory for how designers operate using

[7] `http://games.stanford.edu/language/spec/gdl_spec_2008_03.pdf`

these tools as external elements.

2.2 Discovery Systems and Computational Creativity

In another region of artificial intelligence, quite distant from general game players and the like, there are the fields of discovery systems and computational creativity. Each area has distinct goals – discovery aims to carry out real science, while computational creativity aims reveal secrets of human creativity and share them with machines. These areas have intersections in the form of projects and people who are interested in the creativity practiced by scientists and mathematicians, and what this might say about creativity and discovery in a larger context.

In this section I treat the two fields as largely separate. However, it is my hope that by carrying out my proposed work that I can bring these fields into intimate interaction. Here, I first look at a few models of discovery to establish a space of options for implementing discover systems. Then I review several discovery systems produced in the past. Stepping to creativity, I will review a few general models before reviewing several generative art systems for which the output, at one time or another, was considered evidence of machine creativity.

2.2.1 Models of Discovery

There are several models of discovery as it might be carried out by automated systems. The motivations behind some have been to explain historical discoveries and, behind others, to make interesting and new discoveries beyond the state of our current knowledge. In game design, the lack of clear documentation for historical discoveries (whatever these might be) implies that a forward looking perspective is in order. In the traditional fields for automated discovery, namely natural science and mathematics, there has been a long tradition of keeping detailed records. Accordingly, past discovery systems have had clear points of reference – something that I must forge in game design going forward.

In a landmark book for the field, *Computational Models of Scientific Discovery and Theory Formation*, several different models are proposed – some backed up by historical precedent and others by functioning systems. In the first pages of the book the editors review progress up until the present (which, for them, was 1990) and conjecture a sweeping vocabulary of scientific discovery which aims to put the various systems discussed into a common picture [34]. This vocabulary consists of several scientific knowledge structures and scientific processes or activities that transform them. I review this vocabulary in detail and map it into the field of game design when I discuss my proposed work later.

As AI systems, discovery systems are bound to do quite a bit of searching. The Kulkarni and Simon model of discovery posits that the overall discovery process can be seen as a dual-search of hypothesis and instance (experiment) spaces, guided by surprise [17]. Nearly all models of discovery have similar ideas of a space of hypotheses (theories, more generally) that a discoverer may consider. Likewise, in systems that actively perform experiments, there is a related notion of a vocabulary for describing all possible experiments, from which experiments of interest can be attempted and the results examined. The essence of the Kulkarni and Simon model is anomaly-driven discovery in which experiments are chosen to actively pursue anomalous data and hypotheses are chosen to account for anomalies as soon as possible. While I have not decided on the top-level algorithm for my own discoverer yet, this model seems intriguing.

In an interesting design-related twist, Karp proposed that discovery itself was a design process [14], where a hypothesis that explains the environment is an artifact to be synthesized according to the constraints of predictive accuracy. His discussion introduces design operators that modify a theory towards the satisfaction of these constraints. The result is an impressively detailed formalism for manipulating structured theories in the light of theory-repair goals which also generalizes to an model for goal-oriented design in general. While I have other plans for theory revision in my system, I imagine that the elements of a predictive theory interact in a similar way to how game rules interact to to form mechanics – Karp's design operators inspire a means of manipulating games as software constructs with intention.

Several years layer, Zytkow proposed a literal list of actions to be taken when building a discoverer in a new domain [41]. Initially I hoped this discussion would be directly applicable to my work. However the

list becomes too abstract to apply when the author attempts to simultaneously unify discovery in science, discovery in mathematics, and discovery in databases. In game design, I hope to avoid this problem by borrowing heavily from the grounded practices of game designers, and, only when a system is functioning, attempt to generalize to other domains.

2.2.2 Discovery Systems

Doug Lenat produced the AM (an "automated mathematician") in 1977, arguably the first discovery system [20]. AM is a large Lisp program that worked by searching according to the advice of an impressive 242 heuristics. A frame-based knowledge representation for mathematical concepts is used to initially describe a few basic mathematical notions: sets, bags, and a some common functions. As the system applied its various heuristics, it creates new frames and fill the slots by following more heuristic advice, all the while managing a very complex agenda system. AM is reported to have rediscovered interesting mathematical concepts such as subsets, disjoint sets, prime numbers, and highly composite numbers.

BACON was a very early system (or more accurately a succession of systems) that worked in natural science [18] . BACON uses a much more conservative set of heuristics to guide the explanation of scientific data in terms of numerical laws. In this way, BACON is very much like contemporary, statistical machine learning systems. To avoid searching the infinite spaces of possible numerical laws, the system observed simple regularities in the data to conjecture laws via templates – if two variables increase in parallel, they may be linearly related, so the system then applies heuristics for evaluating the constancy of their ratio.

Over the years, these projects inspired new generations of discovery systems that were able to take more refined approaches to the problem of automating discovery. In noticing that AM was quick to "run out of steam" given a fixed set of heuristics, Lenat created a follow up system called EURISKO that worked in discovering heuristics [21]. Frustrated with the fact that AM and EURISKO were never fully described in the literature (their functioning seemed to hinge on the contents of Lenat's ad-hoc frame representations), Ken Haase created CYRANO, a rational reconstruction that used a deterministic control flow and clean partitioning between the nested loops of exploration and invention [12]. Anchoring mathematical discovery to a well known domain and seeking new knowledge, the GT system (a "graph theorist") focused on a generative space of artifacts (graphs) and would often "doodle" random graphs to get new ideas for conjectures to prove [8]. Finally, in response to the fact that AM never proved anything (it only made conjectures on the basis of regularities), Simon Colton created the HR system which integrated a theorem prover and counterexample generator to formally anchor deduction into the discovery process while simultaneously clarifying and simplifying the control flow down to a simple set of (interestingly domain-independent) production rules [6].

Outside of the flashy role for discovery systems that AM and EURISKO forged, the reasoning that these system have carried out has been refined over several years and separated from the "discovery" label. Logic learning, specifically recent advances in combined abductive and inductive reasoning provides a stable platform for induction of laws from data, revision of theories in the face of new data, and the generation of explanations of new scenarios on the basis of older experiences [31]. Inductive process modeling brings BACON-like induction to bear on real ecological applications where domains-specific knowledge enables improved language biases for discovering a differential equation model of some ecosystem. In graph theory, the process of automated conjecture making (leading to open-loop discovery) has been greatly clarified [19]. Innumerable advances in statistical modeling and general machine learning have also greatly improved in recent years, reaching far beyond the traditions of natural science or mathematics.

Finally, the end-to-end automation of functional genomics is taking place in the Robot Scientist project [15]. While this project depends heavily on very recent advances in genetics and robotics, the robot scientist does have a very significant AI component which, amongst other duties, is entrusted with producing hypotheses, selecting and executing experiments (while considering consumable supplies and expenses), analyzing data and writing reports. I find this project quite inspiring, and, while I would like to call my own project a robot scientist for games, figuring out what the robotic tools of game design are is a first-class research question of its own.

2.2.3 Computational Creativity

While the robot scientist does provide strong evidence that machines can do real science, it is tempting to write off the area of functional genomics as quite routine (after all, a machine can do it). The question of whether such a robot could do the "really creative" part of science, or carry out the creative process in another domain is question of computational creativity. While many researchers in computational creativity place their philosophical home with Margaret Boden there are many alternatives.

Boden's understanding of creativity focuses on conceptual spaces (possible sets of concepts, actions, etc.) [2]. As individuals explore conceptual spaces, their discoveries, if at all, may be p-creative (personal or psychological) or h-creative (historical, if it is new to the world). She distinguishes three types of creativity: "combinatorial" wherein familiar ideas are combined, "exploratory" wherein ideas are continually modified to reach new points in a conceptual space, and "transformational" wherein whole new conceptual spaces are defined that were unreachable by exploration within the old. While these types may seem reasonable in the abstract, it is often difficult to map Boden's ideas onto existing software systems without introducing too much ambiguity.

Another model of creativity that bears mention is the DIFI model (for domain, individual, field interaction) [9]. In this model, creativity cannot be reduced to a property of individual designers, products, or processes. It must be described in terms of a dynamic and socially distributed network. While the DIFI model explicitly rules out a role for individual "creative designers," it does highlight the role of knowledge, as it accumulates in a domain, as a basis for recognizing creativity.

Yet another model, and one that I find quite interesting, is the curious agents agenda explored by Rob Saunders [33]. He proposes an intimate connection between creativity and the satisfaction of curiosity. I will say more about this in a later review of his generative art system.

In the more positive notes of the presidential address that I mentioned in my introduction, Buchanan summarizes models of creativity he had come across in AI, psychology, and cognitive science into four classes: "(1) combinatorial = generate and test; (2) heuristic search = push local test criteria into the generator, add additional global test criteria; (3) transformational = add analogies and other transformations to the space of hypotheses; and (4) layered search = include bias space search at the metalevel" [4]. It is my intention to eventually address each of these levels in my system as they appear needed.

2.2.4 Generative Art

AARON is a fine example of a practical generative art system working in the domain of visual art[8]. AARON applies a collection of heuristic rules assembled over several years of development to produce highly detailed paintings. Interaction of the author and the system slowly lead to the development of an executable theory of painting derived from experiences, good and bad. Though images are the primary product of the system, the man-machine combination of system-and-author produced an impressive collection of design knowledge as well.

By contrast, NEvAr, another visual art system, is hard to imagine as a knowledge production system, even in a larger context. NEvAr, uses genetic programming to evolves functions that, when evaluated, describe how to paint a virtual canvas [13]. Whereas AARON applies a large library of design knowledge, the code behind each of the images coming out of NEvAr is relatively tiny. However, NEvAr is directly in contact with its human audience, from whom it solicits feedback on images as a means to guide the system towards generation of more interesting images. Such a setup is normally known as an interactive genetic algorithm, but NEvAr goes one step further. Instead of asking for human evaluation on every candidate, NEvAr applies a neural network trained on past human image-and-feedback pairs to provide a high-speed proxy for that human feedback, enabling dramatically faster search. Unfortunately, the opaque representation of the neural network prohibits reading the state of the network a satisfying understanding of audience reaction with the art pieces.

In Saunders' *Digital Clockwork Muse* a simulated society of genetic-art agents create and exchange works. As each agent is exposed to new designs, it updates an (opaque) internal model of images it has seen [33].

[8] `http://crca.ucsd.edu/~hcohen/cohenpdf/colouringwithoutseeing.pdf`

New works, whether produced by the agent or one of its peers, may be compared against this model to appraise the work's novelty. From novelty, curiosity is formulated, and with curiosity as a reward, the agent's search for "better" images continues. While each agent's knowledge was largely uninterpretable in this system, emergent social phenomena such as clique-formation suggested the formation of genres withing the genetic art medium *in silico*.

David Cope's EMI system composed music [7]. In may ways EMI was like AARON, both in its large collection of knowledge and slow improvement at the hand of its designer. However, unlike AARON, EMI was capable of some amount of learning on *her* own. EMI could learn (or more formally induce) the structural constraints apparent in high-quality inputs, and, recombining known elements within the confines of these learned constraints, EMI was able produce compositions that were indistinguishable from forgotten works from the original composers that inspired the system. EMI carried out a kind of creative re-synthesis process, turning known-interesting inputs into possibly-interesting-but-assured-novel outputs.

Finally, Turner's MINSTREL system generated textual narratives (and, interesting, designed simple mechanical devices as well) – it was even intended to be a model of creativity itself [39]. Contrary to all of the other systems I have mentioned, MINSTREL actually had (designer) author-level goals that helped produce artifacts that were not just structurally valid, but served some distinct purpose. The system took a small amount of input data (in the form of well-annotated example stories) and, in another re-synthesis process, was able to produce some outputs that appeared to come from a similar space. MINSTREL's re-synthesis process was vastly more flexible (and correspondingly difficult to control) than EMI's; some of the resulting stories were rather grotesque mutations of those from the input. Turner's experiment was meant to test the idea that artistic ability could be explained in terms of problem solving; however, the wide range of results leaves the study inconclusive. The introduction of author-level goals to the system may foreshadow a similar move on my part to ensure that the games my system produces are "real games" and not just abstract state-progression systems.

3 Prior Work

My familiarity with the related work above flows from my previous, personal research experiences. In this section I will cover a set of projects that have given me experience in formalizing expressive domains, representing games in first-order logic, bundling conventions from these logical games into a reusable representation, packaging this representation and associated reasoning systems as a practical tool for game designers, creating a generative space of game-like rule sets, and performing initial discovery in same manner I intend for my systems.

3.1 *Tableau Machine*

Tableau Machine, created in joint work with HCI researchers at Georgia Tech, is an interactive, generative art installation designed for deployment shared living spaces such as homes or offices [29]. My contribution to this system is the core logic for interpreting the environment as sensed by several overhead cameras and expressing the that interpretation via a series of abstract, geometric compositions. In 2008, I published an interpretation of the interactive system as a "creative alien presence" in which I drew on the creativity literature to identify an instantiation of the basic generate-and-test loop for creative systems, randomness as a source of novelty, and situational relevance as a measure of value [35].

The visual art that our system produces is sampled from multiple, distinct generative spaces of images (an example is shown in Figure 3.1). These spaces were designed to have a distinct flavors within a common visual motif. To accomplish this, I formalized the design rules for each space of images using context-free design grammars. While the rules in the design grammars were sufficient for encoding local constraints on the shapes in the images produced, an image-based analysis of the images was required to catalog the emergent, gestalt properties so that both local and global properties could be prescribed by the generation component of the system at run-time.

Figure 1: An example composition by the *Tableau Machine* system

```
% domain theory
fluent(alive(A)) :- agent(A).
event(attack(A1,A2)) :- agent(A1), agent(A2).
terminates(attack(A1,A2),alive(A2)) :- agent(A1), agent(A2).

% initial conditions
holds_at(alive(A),0) :- agent(A).

% trace constraints
happens(attack(a0,a5),20).
```

Figure 2: A snippet of game rules encoded directly in the event calculus defining a trivial combat mechanic

As *Tableau Machine* is intended to engage its audience over a long period of time (two to three months) it is critical that the system notice and adapt to long-term processes in its environment and communicate the distinctions it notices in its input with equally visible distinctions in its output. Consequently, the system's interpretation component was designed to build up a detailed dynamic model of its environment so that it could tell the difference between everyday activities and special occurrences. While the system's knowledge of the environment was stored only in the positions of certain cluster-centers in high-dimensional spaces, it did, in some sense, come to understand its environment better over time.

Towards my current aims, *Tableau Machine* gave me initial experience with coming up with a set of formal rules behind generative spaces of expressive artifacts. Beyond this, the need to *learn* a model of the environment to continue to notice patterns at longer and longer time scales suggested a more general link between discovery and beyond-superficial creativity the design of artifacts.

3.2 Simple Event-Calculus Games

Turning my attention towards games (another class of expressive artifacts) and adopting the event-calculus formalism emerging from Mark Nelson's research, I created several games in first-order logic. A brief example of a game mechanic is shown in Figure 3.2.

To provide a bit of background, the event calculus (detailed in [22]) is a non-monotonic, temporal logic similar to McCarthy's situation calculus. The core elements of the event calculus are *fluents* (conditions which can change over time) and *events* (instantaneous actions which can modify fluents). The truth of fluents over time and the occurrence of events are reified in the `holds_at` and `happens` predicate (adopting Prolog syntax) yielding expressions such as `holds_at(at(hero,vault),t1)` and `happens(move(hero,pit),t2)`. Events are linked to fluents using the `initiates` and `terminates` predicates. By supplying a small set of domain-independent axioms, standard logic programming tools can be used to calculate fluent traces and (event) narratives for abstract logical worlds given initial conditions and known event occurrences.

While other logical formalisms may be capable of representing the state of a game over time, the event calculus seems to be the simplest. Furthermore, within the various axiomatizations of the event calculus, the variant known as the simple, discrete event calculus proved sufficient for my initial experiments. Beyond simply reasoning about time, the event calculus provides us with an *elaboration tolerant* representation for a game's rules. That is, it is relatively safe to modify relations in event calculus by adding and removing individual rules rather than making distributed edits to rules. This property is property suggests that the event calculus may provide a robust basis for a representation of games.

Building directly from the event-calculus axioms, I implemented several games: Minesweeper (the ever-popular single-player puzzle game bundled with Microsoft Windows), tic-tac-toe (a reference game in the game-description literature), a world with character skill progression (a test of embedding an "achievements" system in a game), a multi-player shooter (a test of simultaneous interaction with partial-observability), a purposely broken racing game (created to test diagnosis practices), and a dungeon crawl game (focused on

testing my emerging software engineering practices). Each game was a stand-alone, answer-set program of about 150 lines of Prolog-like syntax. Running each program produced a set of logical facts as the only visible output.

While developing each game taught me lessons which deeply shaped my current project proposal, I will only cover the highlights here. The experiences from my logical game programming experiments fall into three areas, patterns, practices, and tools.

Starting with Minesweeper, I had no experience modeling games in first-order logic. However, by the time I reached the dungeon crawl, I was able to identify several patterns in the games I was creating. One kind of pattern was simply the textual layout of the code in my logic programs: separating game state and events from the trigger logic relating them, separating level-design issues from core mechanics, organizing entities in the game world into a class hierarchy, and saving display logic for the end of the program. Another kind of pattern emerged in the types of predicates I was defining. Certain idioms and design patterns specific to modeling games appeared that would not have been of interest from a general logic programming perspective (such as idioms for modeling default choice by agents in the game worlds). Finally, patterns in how I systematically improved a game's model in response to observing traces (listings of fluent state over time) began to emerge as well.

These patterns gave rise to certain practices which hinted at a possible automation of game design. In particular, the practice of "asserting that the victory event happens at a final time step, given standard initial conditions" was a means of "seeing how someone might beat the game" (effectively running the game's rules backwards). Glaring problems in traces resulting from this practice would often suggest mutations to the game's rules. When most paths to victory seemed reasonable, asking for a random trace in the game world with no particular goals was another way to dig for potential design issues. Fundamentally these practices were about spotting problems in the *semantics* of the game rules when they appeared otherwise fine from the *syntactic* perspective of the text editor. One last practice, involving annotating suspicious rules in the game with a "do I really believe you" flag allowed the same algorithms which found gameplay traces to suggest primitive design actions (by showing that traces resulting in certain conditions are only possible if they include disabling certain combinations of rules).

These patterns and practices emerged in the context of experiments involving a set of logic programming tools. Specifically, I used the answer-set programming system provided by the Lparse and Smodels tools[9] to perform the deductive (simulation) and abductive (explanatory) reasoning tasks required for extracting traces from game models. In further experiments using the inductive logic programming system Progol [23], I was able to summarize emergent statistical patterns in traces as additional logical rules (building a primitive player model from observations of play).

The collection of deductive, abductive, and inductive logic is often discussed in the context of the knowledge manipulation carried out by discovery systems. Though I was using these tools in the context of the design of particular, isolated games, these same tools may someday be used again for automating manipulation of design knowledge in my intelligent game designer.

3.3 Logical Game Engine

Attempting to turn my experience with designing event-calculus games into something usable by others, I modified my last game, the dungeon crawl, by incrementally moving all patterns that were not specific to the game at hand into a separate file, effectively separating the game-specific logic from the game-independent logic commonly found in a game engine. While the label of "game engine" usually suggests support for real-time physical simulation, advanced animation, and network communication, my "logical game engine" provides none of these. Nonetheless, it greatly simplifies the process of creating a new game using the event-calculus formalism. It does this by allowing the author to stay focused on the game by utilizing a set of predefined hooks to build a the dynamic system of a game. I have documented my game engine on my project blog[10].

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

[10] `https://sites.google.com/a/adamsmith.as/game-design-metatheory/ideas/thegameengine`

The game engine primarily works by enforcing a factoring of event-calculus games into several components: the event-calculus axioms, the game's core rules and structural definitions, the setting, the nature model, and the player model. The event-calculus axioms are never changed, so clearly no game need redefine them. The core rules of a game lay out the game state and game events (which directly translate to low-level fluents and events) and link them with trigger logic. Additionally, the core rules define how elements of the setting affect the game's rules. The setting defines what is true (in a timeless sense) in a particular game world, such as assertions describing the design of a particular level or the setting of modifiable constants. A game (generally identified by its core rules) can easily be combined with different setting components to model instantiations of that game for different level-design configurations or difficulty settings. Another easily-swappable component, the nature model, allows the easy redefinitions of policies that control natural-yet spontaneous events in a game world (such as how monsters decide where to wander in a dungeon, given fixed rules for the semantics of movement actions). The game engine's implementation makes a distinction between game events that must always happen in response to other events, natural events that may happen spontaneously, and player events which never occur without the player's direct input. Finally, an optional player model can augment the game's rules with a set of expectations for how reasonable players play, effectively disallowing possible-yet-unreasonable traces from being shown turn in the trace-finding process.

The idea behind the logical game engine was to lift the logic programming to the level of game programming without ruling out the creation of any particular games. While it is certainly possible to build nonsense abstract, dynamic systems using the logical game engine, the definitions in the game engine guarantee that there is a reading of the rules of that system as a games. Essentially, with the presence of the logical game engine, a game programmer (who happens to be using these tools) no longer needs to understand raw fluents and events, only the "game state" and "game events" they define themselves. Beyond this, I have taken steps to ease certain kinds of mutations to a game's definition and to enable, for example, logical level-generation and the incremental induction of player models from observations with minimal complications from the core rules of a game.

At this time, programming with the logical game engine still seems to be a practice only accessible to my peers with extensive logic programming backgrounds. In particular, assembling games at the level of predicates (even if they are game-specific) is not quite the level of detail I desire. Structures that allow easier assembling of interesting game cores are still needed. Luckily, making these logical tools more usable is directly in line with the goals of Mark Nelson's parallel research [25].

3.4 BIPED

One of the major bottlenecks to the adoption of my logical tools to assist in game design is the requirement that the designer recreate game-world situations in their head on the basis of reading Prolog-style facts in a text editor. To address this, Mark Nelson and I created the BIPED system [36] which aims to provide equal support for machine play testing, such as the trace-finding practice described above, and human play testing, which most certainly involves exposing the game to player via a graphical interface.

The basic idea behind BIPED was for a designer to encode the rules of their game in our declarative game modeling language and use our tools to get feedback on their design. Figure 3.4 shows an overview of our system. The modeling language was a slight upgrade to the one used by my logical game engine (with certain usability improvements such as removing the need for explicit T variables in all rules). The single definition of a game's rules is paired with two game engines. The first is the logical game engine discussed previously, and the second was a new game engine (this one actually supported graphics, sound, and input) that enabled the playing of the game by untrained human play testers. In its user interface, BIPED's human playable prototypes enforced a basic board-game-like representation metaphor (with tokens, spaces, connections, and an instruction card).

In an evaluation of the system, we were able to produce the majority of *DrillBot 6000* 3.4 (a board-game-like approximation to the popular Motherload[11]) in only one hour. Human play testing confirmed players' ability to read the abstract world of a mining robot through the board-game-like interface. When we looked

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

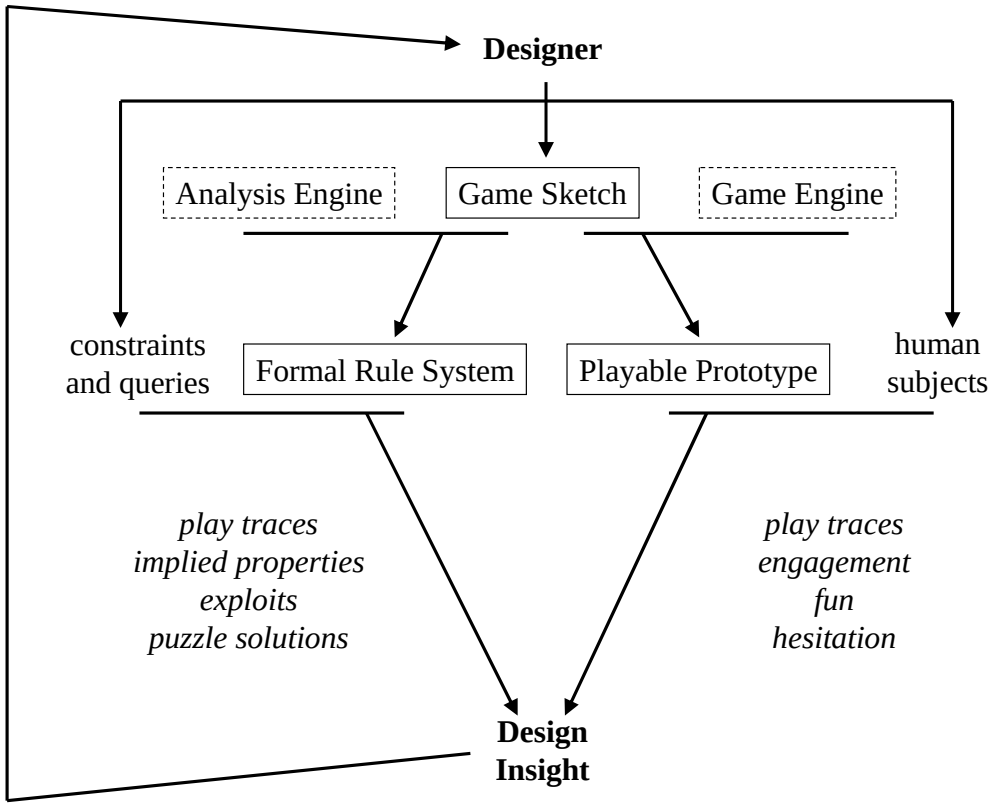

Figure 3: Architecture for the BIPED system

(a) DrillBot 6000 (b) Motherload

Figure 4: *DrillBot 6000* is a game created with BIPED, an approximation to the commercial game *Motherload*

into the maximum number of scoring game actions that could be performed in a given time window, machine play testing revealed several aggressive solutions. It was not until seeing these results from or logical analysis engine that we noticed that our human players had been rather cautious, refilling their mining robot's energy quite often. That is, machine play testing revealed to us a property of our human players that had not been apparent in direct observation.

By suggesting the use of BIPED as a first computational prototype, we aimed to help game designers gain deep insight into their designs as they make the first step of game design that requires them to apply rigid rules to their game. As the title of our paper suggested, we aimed to provide computational support for play testing game sketches (both in the familiar human mode of play testing and the machine mode we introduced).

Our system leveraged the fact that the event-calculus encoding of the game's rules could just as easily be used for incremental forward simulation in a playable game (as a "scripting language") as it could be for the reverse or otherwise non-linear computation required for trace-finding. While this clearly has some benefits in terms of reducing a designer's programming effort, it does have an another benefit. That is to say, from the perspective of extracting plausible play traces from a set of formal game rules, BIPED is ignorant as to who, or what, is driving it.

One final element of BIPED, even an afterthought it its original design, is a detail relevant to my current project. The component of the system that turned designer-specified rules into the fully-specified logic programs had to perform certain syntactic checks (as any compiler would). However, we got somewhat carried away and extended these syntactic checks into a basic design validation system which was capable of detecting not only when a game's definition was incomplete, but if there were unused or impossible events, unchangeable game state, missing elements for the visual interface, or interface trigger events that were not handled. In a sense, our compiler was able to raise design-level issues with the game and was only one step away from suggesting concrete design actions in response to them (and two steps away from performing those actions itself!).

3.5 Propositional Game Generation

With BIPED in hand, a fresh new tool for evaluating the semantics of a set of game rules through play, a clear question emerged: Where would these rule sets come from in an automated system? By analogy to *Tableau Machine*, human and machine play testing are the image-based analysis, revealing global, gestalt

```
add_initiates(E,F) -->
    $? game_event(E),
    $? game_state(F),
    invent_preconditions(Conds),
    $+ (initiates(E,F) :- Conds).
```

Figure 5: A design operator in the game generator language

```
% a switch that can be flipped by triggering an event
game_event(event0).
game_state(fluent0).
initially(fluent0).
possible(event0) :- true.
initiates(event0, fluent0) :- not(holds(fluent0)).
terminates(event0, fluent0) :- holds(fluent0)).
```

Figure 6: An example generated "propositional game"

properties, but the generative design grammar was still missing. My next experiment was building a basic game generation system that used representations very similar to those used in BIPED's language.

As the event-calculus axioms are usually cast in first-order logic with function symbols, events and fluents are commonly parametrized by domain objects. However, I focused on "propositional" rule sets for which events and fluents were represented by ground atoms. While this restriction does make for rather uninteresting games, several technical challenges could be addressed in this limited setting including incremental construction and predicate invention.

The game generator used a primitive operationalization of the game rule construction process. For a given partial rule set, several operators were available: add game state, add game event, add new derived state predicate (a named conjunction of other state predicates), add trigger logic (an elementary clause for either the initiates or terminates predicates), and add a precondition to existing trigger logic. All of these operators were implemented in an domain specific language for game design moves that I created. The generator could either start from an empty rule set, or continue construction on a user specified rule set. Figure 3.5 shows a design operator and Figure 3.5 shows a generated example.

I experimented briefly with different strategies for searching the space of propositional games. To avoid regeneration of previously seen rule sets, I created a means of identifying games modulo renaming of constant symbols. Interestingly, this operation could be abused for performing partial unification of games, effectively explaining one game by partial analogy to another.

With a proof-of-concept for the bulk generation of logical rule sets, I looked to larger issues, leaving the generation of first-order rules and integration with my game engine for future work. At this point, it seemed quite realistic to assume that my formally-represented games could be syntactically generated and semantically evaluated, what remained to be realized was some means of allowing my system to use bigger building blocks to be give it access to the efficient generation of complex and interesting designs.

3.6 Elementary Discovery and Game Design

While it is possible to explain any particular rule set as a detailed construction in terms of those primitive operators described above, this would not yield a compelling explanation. Game designers think in terms of game mechanics, adding and removing coordinated chunks of rules from games as they perform iterative design. To prove to myself that this was possible in the context of my logical representation, I looked through the several event-calculus games I had already created, gathering patterns that I could name and might like to reuse again.

It appeared that the interesting mechanics had components that were often split between the "core rules" and "setting" elements. That is, a hierarchy of objects defined in the core rules would be populated in the game's setting, but these elements of the setting would be used to parametrize other elements of the core rules (including other object hierarchies).

Looking at a concrete example from my dungeon crawl game, I identified the "per-entity bit" design pattern. In the dungeon crawl, there is a rule stating that all monsters are agents (other elements of the core rules described the movement and combat mechanics for general agents in the dungeon). In the setting, there were two named monsters. Back in the core rules, there was a game state declaration that allowed tracking whether a monster was angry or not at a given time. Abstracting this, the "per-entity bit" pattern, formally, is instantiated whenever there is a type of domain objects described in the core rules, that is instantiated one or more times in the setting, and elements of this type are used as the only argument to a fluent. Compared to existing game design patterns, it is much more realistic to imagine making a fully automated detector for this kind of design pattern (in terms of the partial analogy facility) and a corresponding rule construction move that would instantiate the pattern in a game that lacked it.

While I have yet to record these in a logical representation, I have notes on several more predicate-level patterns like this. Though my games did employ higher level game mechanics, it becomes unrealistic to assume that these high-level patterns always have a concise predicate-level definition. If, however, when design patterns were instantiated, a game was also tagged with additional assertions describing how the pattern was instantiated, it becomes reasonable to describe high-level patterns in terms of those as lower levels. Concretely, a particular combat resolution mechanic might be defined in terms of a `per_entity_bit(monster,angry)` assertion. If a rule set is stripped of its pattern assertions (or was constructed using as-yet unknown pattens) it is a straight-forward abductive inference task to recover sets of missing pattern assertions and high-level patterns from the raw predicates in a game's definition given a library of declarative design patterns.

In finding these design patterns and conjecturing a formalism for them, I have, in a sense, performed some of the same game design discovery tasks I hope my future system will perform. In spotting these design patterns, I have paid careful attention to the particular reasoning tasks I am carrying out and under what conditions these tasks make sense.

4 Proposed Work

My proposed work is essentially the simultaneous development of related systems and theory. The concrete systems I will build are intended to act as inspiration and partial evaluation for the theories, while the theories will serve as inspiration for future system components, structure, and interpretation.

In this section, I propose a space of systems (all more or less "intelligent game designers") ranging from a simplistic game generator which merely applies design knowledge to a fully introspective system that uses experience from designing *play* (defined later) to propose conjectures about the discovery process itself. First, however, I will describe some fledgling theories that will enable principled design of these systems. In particular, I will describe a knowledge-level account of game design set in the language of a "game design meta-theory" and an understanding of expressive artifact design as a discovery process motivated by more generic theory of "reflexive creativity".

After covering the space of systems, I next map my research questions into a concrete set of experimental plans that will allow to me to ensure that the system and theories I produce conform to the requirements I set out in the introduction of this proposal.

Finally, I lay out a two-year plan to answer my research questions. The first year fleshes out how to do discovery in game design via extrapolating from my own discovery experience using the same tools I intend my systems to use. Then, the second year looks to answer larger questions with experimental evaluation and to generalize my findings via theoretical interpretation of the larger context.

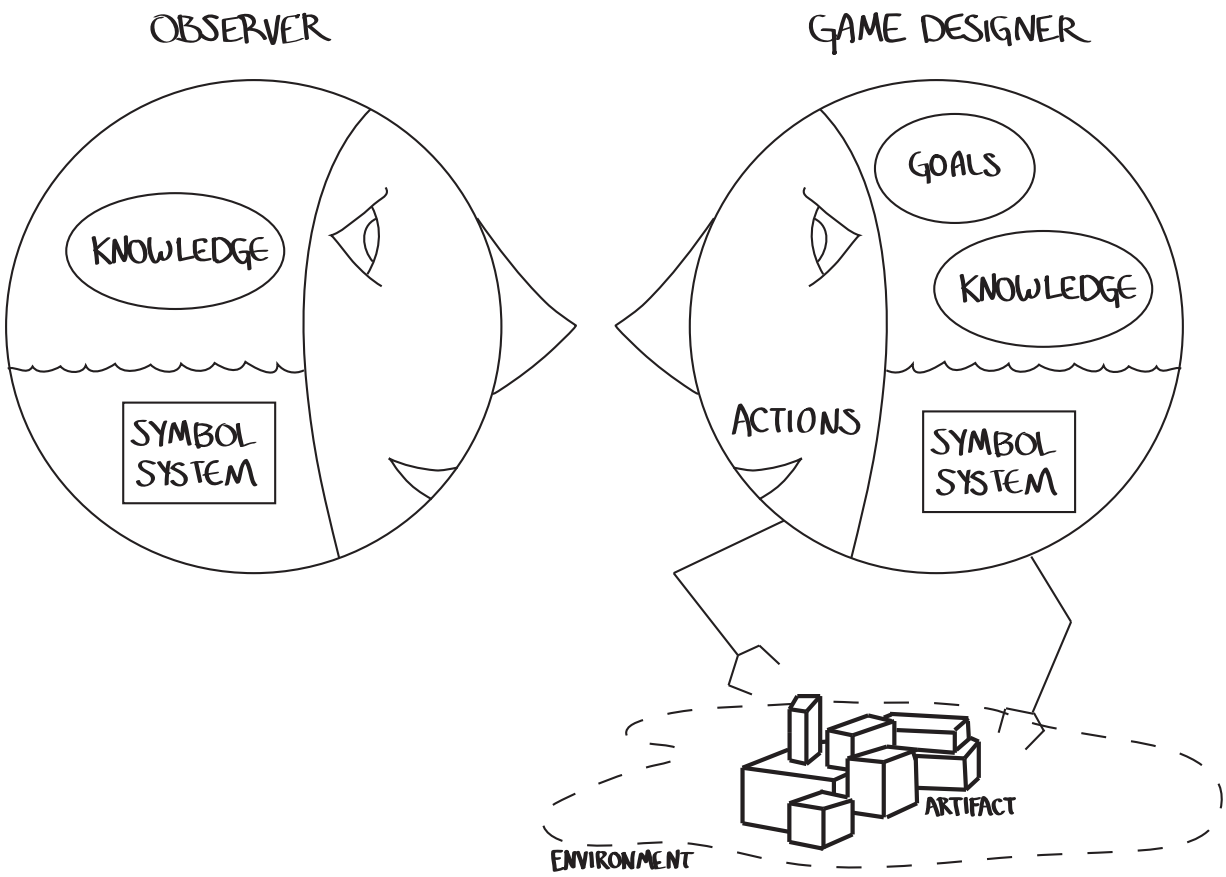

Figure 7: Newell's knowledge level in the context of game design

4.1 Theory

In the following discussion of my working theories, my aim is introducing a stable vocabulary that should be applicable to the description of game designers (human and machine alike) as well as designers of expressive artifacts in other domains. To anchor this abstraction in established terms, I build directly on Newell's "knowledge level" [26].

Recall that the knowledge level is a systems level set above the symbol (or program) level. At the knowledge level, systems are *agents* composed of *goals*, *actions*, and *bodies of knowledge*. A behavioral law called the *principle of rationality* defines an agent's behavior at this level. The principle of rationality states that "if an agent has knowledge that one of its actions will lead to one of its goals, it will select that action" (where some extended principle of rationality is needed to say more about how actions are taken out of candidates currently in the selected set). The essence of the knowledge level proposal is a perspective that allows us to make predictions about a system from an intentional stance, without reference to symbol level concerns (such as inference rules, representations or search processes).

In the context of game design we can extend this picture a bit (see Figure 4.1) by acknowledging that one such agent we can imagine is a game designer. This game designer manipulates some external, structured artifacts that are known to interact directly with a surrounding environment (independent of the designer). My working theories aim to give us a way to partition the designer's knowledge, describe these mysterious

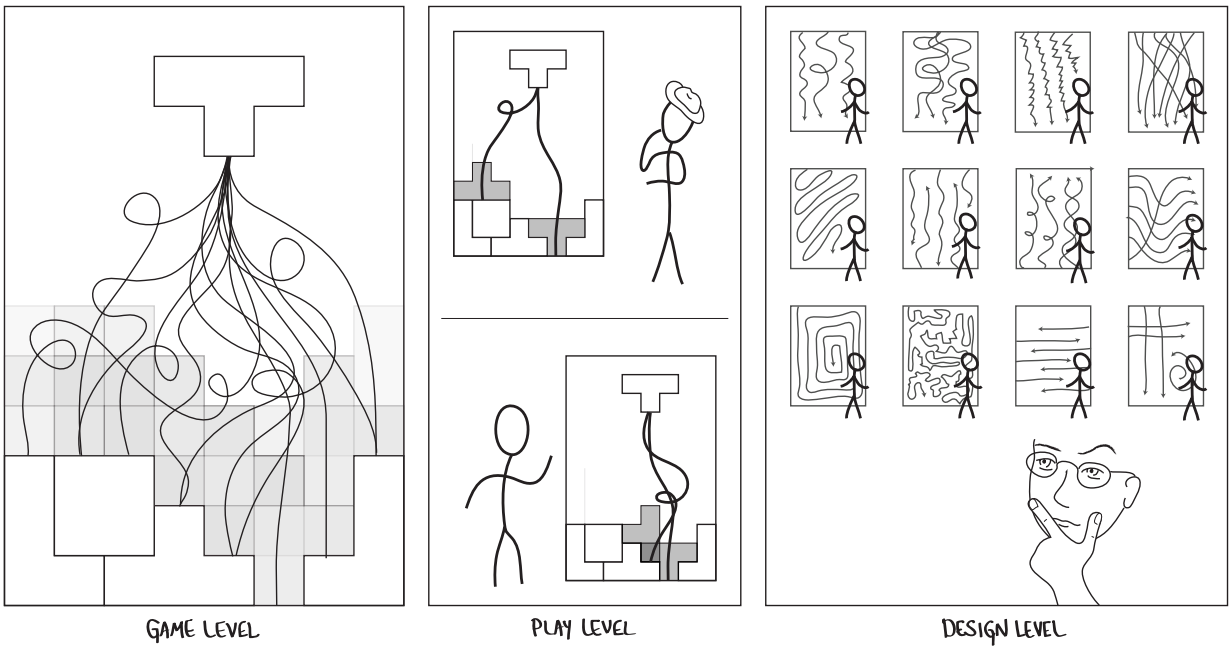

Figure 8: A visual mnemonics for the levels in my game design meta-theory

artifacts and their reactive environment, and guess the designers goals.

4.1.1 A Game Design Meta-Theory

Starting with an agent's knowledge, I propose a layered model which will help me organize all of the relevant knowledge a game designer might posses. The three levels, organized from bottom to top, are: the game level, play level, and design level. Different design actions will reveal knowledge at different levels. Even if a designer is most interested in knowledge at a certain level, they still need to gain knowledge at a lower level to provide sufficient background and experience to formulate interesting knowledge structures at the desired level. Figure 4.1.1 provides some visual mnemonics to help clarify these levels.

At the *game level*, I relate to games worlds as literal physical universes where game rules act as the physical laws for each tiny universe. As such, a game functions as a container for possible histories, which I call *world traces*. A body of game-level knowledge relates the structures used to assemble particular game to the traces it contains. Looking at a possible symbol-level implementation, game-level knowledge may manifest as assertions in symbolic logic about how program elements such as specific lines of code assemble into more abstract game rules and mechanics. Considering other example, records of regularities in traces from the games built with certain rules also encode game-level knowledge. Even just a raw record of traces observed in a particular game counts here. From a knowledge-level perspective, what is more important is what game-level knowledge does not speak about. In particular, game-level knowledge does not help an agent make predictions about which world traces a particular external player might bring out or under what conditions one game should be created by modifying another. Game-level knowledge describes game as structured artifacts as they might be found "in the wild" – independent of player or designer intentions.

Moving up to the *play level*, games cease to work like complete physical worlds and instead describe closed, interactive artifacts which depend on external input to advance in-game histories. Connecting a particular game to a particular player, I identify possible histories of the composite system as *play traces*. An "instance of play" (a game-and-player-pairing) clearly depends on the game and the player but is still

independent of designer intentions. Parallel to game-level knowledge, play-level knowledge relates particular instances of play, as structured artifacts, to traces. Symbol-level representations of play-level knowledge may take the form of models that stand in for the role of particular players, libraries of game-level constructs which elicit certain in-game behaviors on the part of the player, or even just a raw record of play traces extracted from an external, human players for which usable, symbol-level models are unavailable. Play-level knowledge enables a game designer to make predictions about a game as a naturally interactive artifact (and condition these predictions on observable properties of players playing that game), however it does not yet allow generalizations about individual players as agents that exist outside of the games with which they have been seen paired.

Finally, *design-level* knowledge permits predictions about another designer's actions and, importantly, making predictions about the outcomes of design actions taken by an agent itself (via introspection). That is, knowledge at the design level deals with reified constructs from the play level as well as how-to knowledge regarding game design actions. Symbol-level design knowledge representations may include any representations from lower levels as well as additional rules for how lower-level structures should be combined towards particular design goals. Additionally, raw records of design activities and their results is an example of the symbolic precursor to the design-level knowledge of actions. Building the design level atop the play level raises the interesting implication that game designers really design play, not just games. Even the most well designed games can still exhibit "gameplay bugs" when paired with players that do not conform to the models of play the designer had in mind at design-time.

More experience with concrete systems that operate with these distinctions should help me add additional distinctions. With games and play instances being predominantly "software artifacts," a more refined meta-theory might make room for explicit programming, debugging, and other software engineering knowledge. In the current model, there is no conception of players as rational agents or even as recognizable entities across a space of games – perhaps there is room for a player level too.

4.1.2 A Knowledge-Level Account of Game Design

My game design meta-theory only structures an agent's design-relevant knowledge, but we still need *goals* and *actions* to fully account for the practice of game design at the knowledge level. Looking first at actions, I apply the same three-level categorization.

There are three kinds of designer actions and the are partitioned by the kind of knowledge they reveal (as opposed to the kind they consume). At this time, I will only give simple examples, as I have not yet developed a satisfyingly complete set of game design activities. Game-level actions, as one might expect, deal only with the assembly of game artifacts, producing new games from old ones by adding or transforming a known structure, naming compounds of more elementary structural elements, or extracting world traces from games.

Play-level actions enable the induction of play-level knowledge through activities such as pairing existing games and player models to form new instances of play, extracting play traces, and conjecturing (or even proving in some cases) relations from elements of the model of play to observed play traces. Finally, design-level actions might take the form of assembling a plan for game and play level actions which will achieve a designer's goals, or conjecturing design-level shortcuts from the past performance of other activities. It is important to note the distinction between designer activities (encompassing everything a designer does) and design-level activities (those which cannot be described as lower-level activities).

Organizing actions around the kind of knowledge which they reveal instantly suggests that *the purpose of a designer might be to discover new design-level knowledge.* Though we can easily imagine a "designer" who is exposed to new evidence but chooses not to learn from it (leaving this designer to be better described as a "generator"), it is much more interesting to consider a designer who does this learning part as well. If we shift our perspective from an interest in systems that aim to produce games, to those that produce knowledge, we can view the production of games as more of a side-effect of the design process (remnants of experiments in an active-learning process).

Within this perspective, I propose that the goal of (interesting) game designers is to produce design knowledge. The same kinds of actions a game "generator" performs can be reorganized in service of knowledge-gain

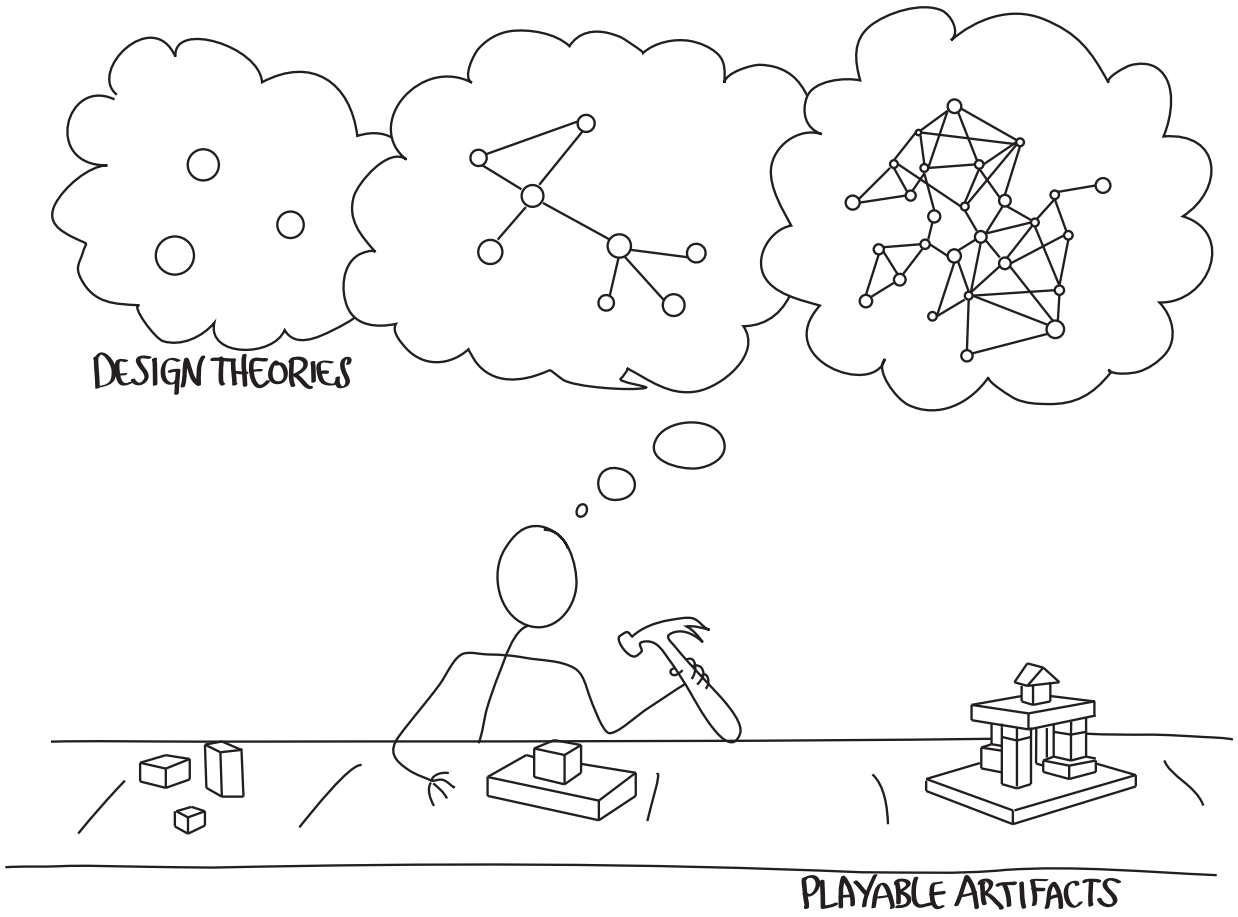

Figure 9: Through experience producing playable artifacts, a game designer may improve his design theories which enables the principled production of more complex playable artifacts, and so on.

goals (which begins to sound a bit like curiosity). This proposal might be offensive to others who have thought about the game design process because, most glaringly, it makes no reference to "fun" in games. It is my hope that fun (and its relation to learning) can be fleshed out as play-level knowledge. If there are certain regularities in how the human players play games that can be related to fun, it is certainly the duty of a effective, intelligent game designer to explore this and produce games (informed by various understandings of fun) as concrete experiments. In terms of design-level knowledge, understanding "good" games is as important as understanding "bad" games because the precursors for new design knowledge can come from "failure" quite easily.

I want to identify the application of the game design meta-theory to the shaping of knowledge, categorization of actions, and ascription of learning design-level knowledge as a top level goal as my initial formulation of a knowledge-level account of game design. Figure 4.1.2 shows how one might visualize the development of incrementally richer design theories and playable artifacts. By organizing game design as a knowledge-seeking practice, it is possible to see it as an instance of a discovery problem. Further, by identifying the important actions in game design as ones that are mediated by machines and the symbolic representations stored therein, game design becomes approachable by automated discovery systems.

4.1.3 Reflexive Creativity

My knowledge-level account of game design hints that the practice is almost purely about knowledge discovery, but it is also aimed to explaining how and why game designers produce games, which are certainly expressive artifacts. The creation of expressive artifacts is almost always the result of a creative process. At this point, I need to introduce my working theory of creativity:

> Creativity is the rational pursuit of curiosity that results in surprise. It can only be directly appraised by the agent defining the sense of curiosity (a thirst for knowledge) and surprise (a break in expectations).

I call this notion reflexive because it applies only when an agent is evaluating its own creativity. More importantly, it suggests that we cannot say something about the creativity of an isolated artifact or of the creative process of another agent without introducing assumption about the knowledge and goals that agent possesses. This differs greatly from other understandings of creativity which allow the direct evaluation of another's creative process or the evaluation of the creativity of a particular artifact, as divorced from the context of its authoring (such as those involving a judgment of quality relative to emotional response [28]).

Recognizing another's creative process in the reflexive regime takes the form of asking whether, if I had the same knowledge and curiosity goals that the other had, I would have chosen the same actions and judged the result as a surprise. However, there are certainly times when I find the other's accidents to seem particularly creative. Initially this would seem to invalidate the reflexive view because I judge the interesting act on the part of the other be out of line with their goals if it was really an accident. Nonetheless, I can explain my interest in terms of the reflexive view by saying that there is another agent that I was really conceptualizing who might have taken that action on purpose as a bold and rational reach toward greater understanding (achieving a part of the curiosity goal, in my eyes).

This is process of ascribing creativity to another is directly patterned after the way one agent makes predictions about another at the knowledge level (and is indeed an instantiation of this process). Neither agent has direct access to the other's knowledge, but they can, by making assumptions about what the other agent knows and using their own symbol-level implementation of the principle of rationality to carry out action selection, determine what the other agent *should* do towards their goals. By matching predictions with observations, one agent can begin to determine what the other agent must have known and expected if a particular action was considered creative, or, the reverse, whether an action was evidence of creativity if the contents of other agent's bodies of knowledge are considered to be sufficiently accurately known. Beyond process, a sense of "creative artifacts" is recovered by asking whether, if I were the agent who produced the artifact, I would consider the artifact as evidence of my creativity. An artifact that serves as evidence that I am rationally pursuing the goal of creativity might look like an experimental setup, where I have constructed some apparatus, given my working knowledge of the environment, in such a way that I expect the environment's interaction with the apparatus to potentially expose new knowledge.

We might judge a painting as creative if we notice a deep resonance between arrangements of strokes on the canvas and our subjective reaction to it. I might read this as the painter performing an experiment in confirming their finely detailed understanding of the interaction of layered perception systems on the part of the audience (where the surprise is that such a delicately crafted artifact indeed performed according to a known model instead of failing for an unimportant reason). We often dismiss uncreative artifacts with claims about their banality, predictability, derivative nature. These claims all work to defeat a story of that artifact as evidence of high-knowledge-gain experiment. Maybe there is something more to a statement like "only an idiot would call that creative" than we might think.

Many other issues in the discussion of creativity such as the role of analogical reasoning, recombination of familiar elements, or communication with peers still live on in the reflexive view. That is to say, reflexive creativity is not a replacement for other understandings of creativity, instead it is primarily a perspective which guides interpretation and prioritizes those explanations that are coherent when an agent applies them to itself.

4.1.4 Expressive Artifact Design

With a rough understanding of reflexive creativity in hand, we can look at attempting to explain a practice of expressive artifact design (such as game design). If creativity is really about satisfying a thirst for knowledge, then designing expressive artifacts must serve the learning process (in addition to any particular expressive goals the designer has). While an agent may have specific expressive goals (whether they be educational, political, economic, or otherwise), I would like to focus, first, on when an agent would like to express its own creativity. That is, when the agent would like to make it easy for others observing the actions it takes (including the construction and sharing of concrete artifacts) to interpret the situation as the agent rationally pursuing its own creativity. When building systems that attempt to prove the machines can be creative, this is a very common expressive goal.

Turning to reflexive creativity again, the goal of expressing creativity implies certain practices in the design process. In particular, if an agent would consider itself creative, it is rational to accurately expose its own knowledge, goal, expectations, and criteria for choosing actions (to make it as easy as possible for the audience to agree on the agent's own judgment of creativity). In particular, if the creative agent can explain, before it performs an experiment, what it expects to happen and what it learned after the experiment, it is easy to read the experiment as a move that aims to gain knowledge.

If it is starting to sound like I am proposing that an expressive artifact designer should act the same way a good scientist should act, then I have succeeded. My understanding of expressive artifact design as discovery, while somewhat crude at the moment, is the fundamental motivation for structuring my intelligent game design system as discovery system. While it may be possible to explain expressive artifact design as, say, a product life cycle engineering task, such a pursuit would not further our understanding of creativity. As I operationalize game design within this framework, I hope to expand upon the relation of discovery to creativity and find concrete ways to validate my theory of reflexive creativity.

Admittedly, my conception of expressive artifact design is currently too restrictive in only considering one type of goal. In building systems I will look for ways to mix in expressive goals beyond "show off my creativity" in a principled way. This is a major challenge. Nonetheless, I feel that exploring this by building a discovery system that works in an expressive domain is the best way forward.

4.2 Systems

Adopting the perspective of game design as a discovery practice, I now propose a space of intelligent game designers which starts with the structure of scientific and mathematical discovery systems as a base and builds towards the domain of game design. How game design maps back to scientific or mathematical discovery is something that needs to be developed in practice. The answering of my research questions critically hinges on successful connection of game design to automated discovery in the context of a working system.

4.2.1 Overview

Consider Figure 4.2.1, a depiction of the knowledge-level context of the system I am proposing. The intelligent game designer which my thesis aims to understand is identified with the "core discoverer" element. The core discoverer operates within a larger, real-world context. While I would like the model pictured to be compatible with human game designers, this model is intended, primarily, to describe the system I want to build. Whether the architectural boundaries drawn in my system can always be applied (and thus be candidates for inclusion in the general knowledge-level account of game design described above) remains to be seen.

The first distinct role in the context, the *system author* (myself), is responsible for creating the core discover, and populating its internals. In the analysis of other systems, this role could be filled by other individuals or teams. In the case of an introspective system, I should allow the system to temporarily occupy this role when it is introspecting.

Some of the internals of the system are also made available to set of *designer peers*. Conventionally, these design peers are human game designers who may want to comment on what the core discoverer has produced

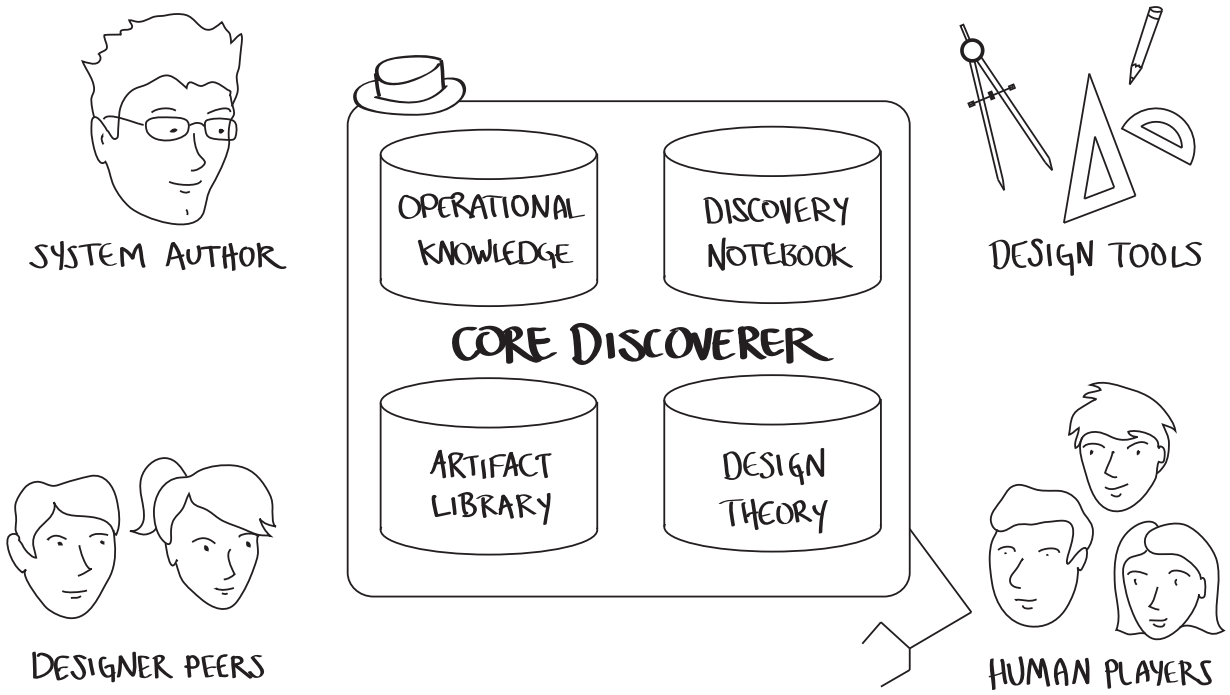

Figure 10: A knowledge-level overview of the intelligent game designer and its immediate environment

(in terms of playable games and design knowledge). However, it is easy to imagine a situation where several automated discoverers act as design peers for each other, sharing artifacts and knowledge. The designer peers are only allowed to inspect, not change the internals of the core discoverer. If a design peer wishes to change the way the discoverer operates, it must temporarily occupy the system author role. That is, one designer peer may pull ideas from another, but they may not push commentary from this role.

Human game designers certainly make use of human players in testing their designs, a key element of established practice. Accordingly, I allow my core discoverer to make queries of *human players*. These queries need not only take the form of requests to play prototypes, they may also solicit sentiments on a player's past play traces or traces resulting from some other means. This allows for non-play interaction between a designer and players.

Finally, the core discoverer interacts with a collection of *fixed design tools* which automate or otherwise aid specific aspects of the design tasks, but do no, in and of themselves, embody an essential piece of an intelligent game designer. File systems, compilers, databases, validity checkers, and classification or clustering tools, while certainly usable in the development of games, have meaningful uses outside of games; so I consider them external tools. I apply the "fixed" label to emphasize that these tools perform a static function and do not improve as the core discoverer learns (though the discoverer may learn to use them more effectively). The "core" label on core discover aims to reinforce the separation of the discoverer from these external tools.

Within the discoverer itself, I identify four bodies of knowledge. This need not clash with the partitioning of knowledge supplied by the meta-theory, as they are not mutually exclusive – I claim any bit of knowledge possessed by a designer can be placed safely in either model. The partitioning here maps more easily to an implementation with computer memories whereas the partitioning in the meta-theory helps distinguish the actions available to an agent.

The discover has (and indeed must have) some *operational knowledge* which embodies its own extended principle of rationality as well as defining the interface of the system to its surroundings. This knowledge can be considered fixed (though its state in an introspective system is unclear). This operational knowledge should encode a functional understanding of game design as a discovery domain (that is, it contains mostly design-level knowledge pertaining to actions).

Clearly, the discoverer has some working *design theory* (game, play and design knowledge that has been discovered from experience). The output for an intelligent game designer should act as a view onto this design theory. Though elements of the design theory may be considered on their own, it will likely be more interesting to understand them in the context of the operational knowledge which created them.

The *artifact library* should be understood to be a catalog of every game or play model that discoverer has ever considered. Artifacts (such as playable games) can be extracted from this library. However, this clearly divorces the artifact from the design theory from which it explains it. The artifact library could be seen as a collection of game and play-level knowledge, but it is separated out here to clarify that a designer can be exposed to playable artifacts that they might not currently possess sufficient design knowledge to construct themselves.

Finally, I will allow the discoverer to possess a body of knowledge which neither tells it how to discover, how to design games, nor describes any particular game or play instance. This final catch-all body of knowledge, which I call the *discovery notebook*, holds knowledge pertaining to partially completed operations and partially formed theories (a primitive kind of design-level knowledge, according to the meta-theory). The notebook will function much like a scientist's notebook.

I discuss concrete examples of knowledge populating these four areas later on when I discuss possible symbol-level implementations for them.

4.2.2 Discoverer Architecture

At a high-level, I intend the general architecture of my systems to follow the pattern of production systems (with production rules, a fact-base, and a rule engine responsible for action selection and execution). At this stage, I am not ready to commit to a particular rule engine; however, it is safe to imagine of the system as being organized as a large Prolog program.

To translate the idea of the core discoverer I presented in the overview section into a sketch for a production system, I will look at possible implementation of each of each of the discoverer's bodies of knowledge as production rules and fact-base entries.

The operational knowledge would take the form of a large library of production rules (and some facts for configuration) that focus on the discoverer's discovery-level actions. While most of such productions will work closely with the contents the notebook, these rules will also govern changes to the artifact library, and design theory. By far, working out the discoverer's operational knowledge will be the hardest of the knowledge-engineering tasks required to implement these bodies of knowledge.

The design theory will contribute production rules that describe actions related to artifact design. As the system operates, new design-level production rules will be produced, as directed by discovery-level actions. The fact-base portion of the design theory can be read as body of what-is design knowledge, while the production rules encode how-to design knowledge. A core set of production rules in the design theory will ensure that the representation of the artifact library as a simple set of facts is kept up to date.

The notebook, following the metaphor of the scientist's notebook, would have no production rules of its own. It may be reasonable to factor the notebook as I have presented it into a logbook which records all actions in an append-only fashion and a read-write scratchpad which might be read as blackboard. However, until I come up with similar refinements to the representation of operational knowledge, I would like to leave this bit of the architecture open and flexible. I am not aware of any clear documentation of how human game designers currently use their notebooks (if at all) as a reference point.

As I manually evolve the operational knowledge in the system I intend to produce, I hope to find collections of production rules which have meaningful names and functions and use their presence to conjecture a more refined theory of game design as an automatable discovery task. That is to say, while my proposed systems are best described as production systems now, it would be quite interesting to realize that a particular pipeline or cyclic architecture serves as a better explanation of effective discovery.

At this point, I have introduced several layers of abstraction between the practice of game design in the abstract and a particular implementation of an intelligent game designer as a production system. I think this layering will pay off when it comes time to understand what my software systems are doing in terms of game design when they are performing basic symbol-pushing operations. That is, when some structured term is added to the working memory, I will be able to recognize that this term is, perhaps, a descriptor for a software design pattern in a game's code and that this descriptor (a game-level concept) lives in the system's design theory, and represents a surprising discovery that several very complex games are now easy to describe given the definition of the new construction element.

4.2.3 Operational Knowledge

Focusing in on the operational knowledge in my systems, I should adopt a standard vocabulary for talking about game design as a discovery domain to permit the transfer of existing discovery practices into the game design domain. This vocabulary, due to Shrager and Langley [34], partitions a discovery domain into a set of scientific knowledge structures and scientific processes which create and manipulate these structures in the context of a physical setting. While it is easy enough to say that I will realize these knowledge structures and processes as facts and production rules at the symbol-level, how this will play out critically depends on how I seat game design as a science within their vocabulary.

First, looking at knowledge structures, I need to define several notions. I do not expect to have a richly detailed mapping of these notions to the game design domain until I reach the end of my work; however, I currently have some tentative definitions.

Observations represent recordings of the environment from sensors or measuring instruments. In the context game design, the primary observations are play traces (such as a recordings of game events as triggered by a live, human player playing an interactive game).

Taxonomies are libraries of concepts in a domain along with specialization relations between them. In game design, I can imagine taxonomies of games, play instances, player models, and the various structural components of such artifacts.

Laws are statements that summarize relations among observed variables, objects and events Laws vary from general statements with parameters to fully-ground assertions that apply to only a single situation. In game design these take the form of existence and implication relations between elements in the code-level representation of play instances and the play traces that they contain (relations between syntax and semantics of game-related constructs) in terms of literal game state and game events.

Theories represent hypotheses about structures and processes in the environment. They differ from laws in that they make reference to unobservable elements (which are elements of taxonomies). In game design these may take the form of templated implications of the form "If game-rule-chunk A is present in the game and player-model-chunk B is present in the player and A and B are instantiated in a way to satisfiy a set of conditions C, then named play-structure D should be evident in play traces".

Background knowledge is the set of beliefs that are held fixed during a study. In my systems, such knowledge will be added on an as-needed basis to the body operational knowledge. The source of this knowledge will be experience carrying out game design discovery tasks myself.

Models indicate how a theory or law applies to a physical situation (declaring which values, observable or not, are relevant). By my game design meta-theory, these models will take the form of games and game-and-player pairs.

Explanations are the instantiated links from general theories to the specific laws they predict for a given experiment. Given a conception of theories and laws in game design, explanations in game design will function in the standard way.

Predictions are concrete assertions about observables in the situation at hand (i.e. laws that may or may not hold in the given situation). The form of a prediction is theory independent as it only applies to the set of observables allowed by the current instruments. In game design these will refer to assertions about the presence or absence of game state or game events in isolated situations, not across spaces of games.

Anomalies in a scientific domain are simply examples that fail to have explanations in terms of the known theory. In game design, it makes sense to think of a set of a game or play instances along with a particular world or play trace which it formally contains, but cannot be explained in terms of known predictive elements of the design theory. That is to say, it is not that a game or a trace itself is anomalous, it is their co-occurrence which points out a flaw in the theory.

Each of these scientific knowledge structures of game design will be ground out in concrete data structures, stored in working memories. The exact slots that these data structures posses will depend on what seems important in extensions of the discovery work I have done so far. Having data structures is not enough, of course; I need operations on these structures, and I need them to make sense as scientific processes of game design.

Observation will require my systems to interface with human players. The means of programmatically exposing a game model to human players and collecting trace data from them has already been implemented in the human-testing component of the BIPED system. However, this process does place the rather hefty requirement of possessing a visual representation layer on all games that are to be subject to the process of observing human play. This user-interface layer may be needed to be created with human assistance for many games until the system has built up sufficient knowledge to craft an interface on its own. Currently, my machine play testing tools only address the mechanics of an abstract game world, not its visual representation.

The vocabulary defines several more scientific processes. For now, I envision programmatic implementations of *deductive law formation*, *explanation*, *prediction*, *manipulation*, and *experimental evaluation* to be fairly unsurprising bit of code and strongly dependent on the particulars of the representations I use. For *inductive law formation* and *theory formation*, I plan to leverage recent work in the logic learning community that provides hybrid abductive-inductive logic learning tools which already have proven effective for these tasks in multiple domains [31]. *Experiment design* will take the form of instantiation of general experiment templates out of a fixed ontology (along the lines of the Robot Scientist [15]). Finally, I plan the *taxonomy formation* process to emerge from experience in the recording and indexing tasks I will perform in the manual discovery phase.

In additional to the kind of operational knowledge discussed above (which enables representation with data structures and symbolic manipulation of these structures according to a science-motivated story), the

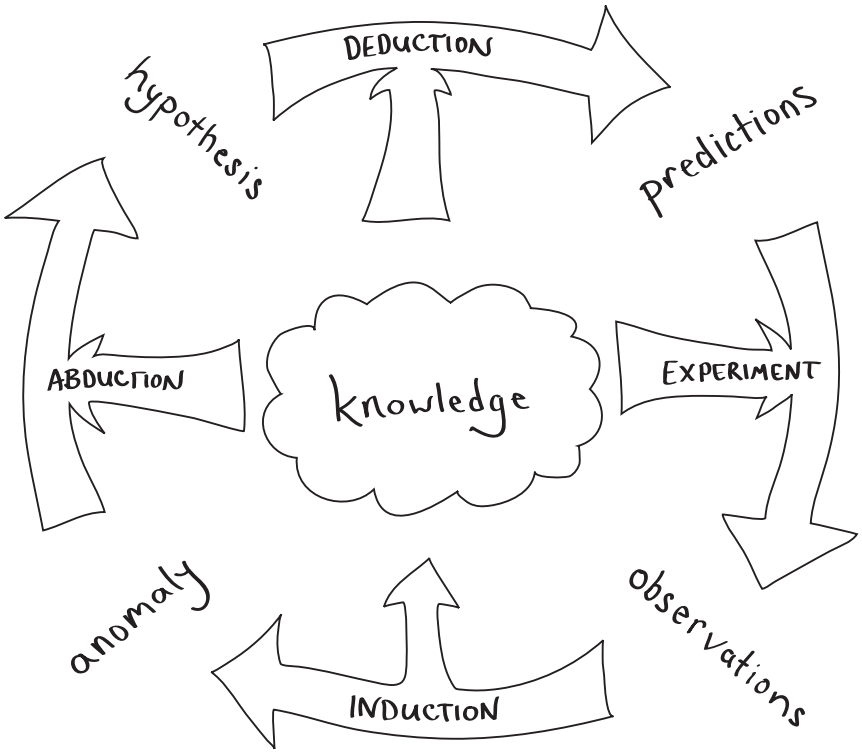

Figure 11: Peirce's cycle of discovery illustrating the iterative development of new knowledge

system will possess basic rules telling it how to string several scientific processes together to carry out complete cycles of discovery. Pierce's cycle of discovery (summarized in Figure 4.2.3) has been used to motivate closed-loop models of scientific discovery in automated discovery systems [30]. In game design, some elements of the diagram should be renamed and redefined, but, I cannot say which at this time. Fortunately, the bulk of the three inferences processes are now well supported computationally, and my initial work with formally representing games seems to be a promising model for the the experimentation process.

Envisioning a single cycle of discovery within this vocabulary, I can imagine a game designer starting with, say, an anomaly. Perhaps the designer has observed an new player making some unexpected (yet still valid) moves in a game with which the designer was already familiar. This occurrence is not explainable in terms of the designer's current play-level knowledge, so he must apply some inference to repair his theory to account for the anomaly (we may call it abduction, but induction and deduction are likely to be involved as well). This repair includes making a hypothesis about how this new player operates. The repaired theory may account for the one anomaly, but the designer should validate this new conception with more evidence. Applying inference to the current design theory and known games, the designer can make a prediction about how this new player should play a slight modification of the original game. As an experiment, the designer asks the player to play through the new scenario, which may result in another anomaly.

It is unfortunate that the various vocabularies I have referenced so far have not been consistent with one another nor complete enough to directly dictate a clear course of action for an automated game designer. By focusing on game design and not trying to account for all areas of scientific activity, I hope to be able to produce effective operational knowledge for my system fairly easily as a result of experience.

4.2.4 Artifact Library

The artifact library, storing games, player models, and game-and-player pairings, should be a store for key meta-data elements. These will include references as to which design moves were used to construct a particular artifact from another (or whether they were entered by the system author) and the auxiliary decisions involved in pairing two artifacts to make a coherent composite. Any traces recorded in the artifact library should reference, of course, the models from which they were derived, any constraints that were applied when looking for that trace, and whether the trace was derived from human players or through computation.

If there is a particular hand-crafted game or play model that I would like the system to analyze and use as a basis for future exploration, it is a simple matter of inserting this artifact into the artifact library and making a minor modification to the notebook to queue this artifact for study.

4.2.5 Design Theory

The design theory is the system's answer to the question: What are games, players, and play and how do I make them? Answering this question clearly requires what-is knowledge and how-to (or when-to) knowledge.

Focusing on what-is knowledge, the main issue is naming and detailing game and player structural elements. That is, pieces of games and player which can be reused in various contexts while retaining certain certain theoretical properties. Fundamentally, knowing a useful set of structural abstractions lets the system explain how to build particular games or players in a smaller number of steps than building them at the level of primitives (state, events, and elementary predicate clauses). As such, these should have representations which consist of chunks of state, events and other predicates, which are parametrized in some way, and come with conditions under which they are meaningful to instantiate.

Consider some design theory element examples in Figure 4.2.5. The "per-entity bit" pattern discussed in my prior work is another example of such an abstraction. Each element describes a coordinated set of assertions that implement some mechanic or other structural element of a game. It is possible to read representations as recognition rules ("if all of the conditions are satisfied by a game, then named mechanic is instantiated in that game") or as construction rules ("to instantiate the named mechanic, add structures to satisfy the conditions").

```
setting_construct(dungeon_map(RoomPred,DoorPred), [
    {RoomPred is predicate defining rooms},
    {DoorPred is predicate defining doors}]).

mechanic(movement_between_rooms(R,D), [
    {has rooms R and doors D},
    game_state(at(agent/1,R)),
    game_event(moves_to(agent/1,R)),
    {corresponding initiates/terminates clauses (trigger logic)}]).

setting_construct(pc_avatar(PcPred),[
    {PcPred is predicate with unit recall defining the name of the
        symbol used for the player's avatar}]).

player_construct(pc_avatar(PcPred),[
    {new predicate for deciding if a symbol represents the
        player's avatar in terms of PcPred}
    {some predicate production rule maps
        agent_at(Agent,Location) to avatar_agent_at(Location)
        and the like}]).

helper_logic(short_path_selector({...}), [
    {predicate template that identifies simple, short paths on room/door networks}]).

play_construct(location_memory({...}), [
    player_state(visited({location predicate})),
    player_event(visits({location predicate})),
    {cooresponding trigger logic}]).
```

Figure 12: Example game and player structure elements of a design theory

To support the scientific explanation process, the discoverer's design theory needs to contain more than just structural definitions. The other part of the theory is the relations between these structural elements and the traces that artifacts created with them contain. These are implications or other statements for state and events in traces like the following: "if the victory event happened, then the boss-kill event must have happened (indirect dependence)", "monster-teleports never happens (non-existence)", or "if ever in this room and not holding this item, the hero always leaves the room on the next step (forced move)". These examples are actually laws (which may even be provable from a game's representation), so there must be additional relations which link named structural elements to such laws such as "if the game contains the 'boss-kill victory' mechanic, then the law 'if victory event happened, the boss-kill event must have happened' must hold".

Because there may more than one way to construct a given game in terms of known structural elements, there may be more than one explanation for particular regularities in a game's trace. The statement of this multiplicity for a particular design theory might serve as a law in an introspective discoverer.

Beyond what-is knowledge, the system will possess how-to knowledge that describes when those structural elements can be used to make a new game or player from an old one whenever direct preconditions are met. That is to say, even though structural elements come with preconditions for their safe application, there are additional considerations which dictate when adding an element makes sense from a design perspective (possibly hinging on recently added entries in the notebook).

How-to knowledge, then, takes the form of when-to-always and when-to-never assertions. By default, it is safe to make a game with a room-movement mechanic from any game which already has predicates that fit the constraints for rooms and doors. However, a useful bit of how-to advice might tell the system that the presence of another room-movement mechanic in the game should make this design move impossible. On the other hand, if a game has a PC-avatar construct, how-to knowledge might suggest that the system always add the PC-avatar construct to the player model as well. In this way, how-to knowledge constrains the space of artifacts that the system will ever produce to those that are more interesting.

Note that knowledge that reads like "to produce traces with interesting property X, try adding structural element Y" is not captured by this particular notion of how-to knowledge. If the system knows that "property X is present when games include element Y" as a matter of what-is knowledge, then the construction rule is already implied. That is, all what-is knowledge entails a kind of default how-to knowledge which does not require extended design experiences to learn. On its own, the what-is knowledge is a complete, predictive theory of play; with how-to knowledge, it becomes simply a more effective theory in terms of reducing the number of possibly uninteresting explanations the theory may provide.

As a concrete example of how adding how-to knowledge makes a theory more effective without changing its predictions, consider the case of when the system has only the barest of what-is knowledge: it knows how to assemble games from only isolated predicate clauses, effectively programming at the assembly-language level of my game representation. When presented with a very complex game, the system can only explain the game's construction by asserting that several tedious, unrelated, construction moves were made. When asserting a bit of how-to knowledge that says never to apply more than, say, three low-level construction moves in a row, the system now looses the ability to explain the game's construction and the corresponding ability to justify its trace prediction in terms of such a poor explanation. Adding the how-to knowledge made the overall design knowledge more realistic by making it more conservative, however it did not modify predictions at all (it only blocked them being made). When pressed, we can imagine the wiser system claiming "the game functions as expected, but nobody would ever do that way in practice."

For now, I expect how-to knowledge to be only added by-hand to help guide the system. However, I can imagine systems which learn these rules on its own, possibly even by such simple means as speed-up learning.

4.2.6 Discovery Notebook

It is easy to see the notebook as the discoverer's internal, mental state. Given that the system will be a general production system, the notebook can be used as a place for the discoverer to record its current design and discovery goals and progress through task decompositions. In this way, depending on how I populate

the operational knowledge, the system can function, at times, more like a planner or more like a reactive agent, without having to commit to a particular architecture up front.

I would like the system to make a detailed log of every action it takes in the notebook, firstly, as a way for me to understand what it is actually doing, and, secondly, to enable production rules to inspect the past performance of the system and conjecture design knowledge, literally, on the basis of past experiences.

4.2.7 Design Moves (Actions)

Having described the state of the discoverer, I should lay out the actions available to the discoverer in the exploration process. Generally, these actions fall into two categories: design moves and discovery moves.

Design moves apply to processes that manipulate game and play-level artifacts. As such, obvious design moves include constructing a pairing of game and player, constructing a new game or player from an old one and a structural unit, getting a machine-generated trace for a model, or getting a human-generated trace for a game.

Constructing a new pairing of game and player involves more than just picking candidate artifacts. Player models, I imagine, are glued to games by attaching player model constructs to game model constructs. If a player model has a mechanic that plans a simple path on a map for an agent they think they control, this player model needs to be instantiated with the correct object bound to the player model's avatar slot. Beyond this, which predicates the player model looks at when forming paths (the raw room-placement predicates or the room-placement-modulo-open-and-closed-doors predicates) matter as well. Operational knowledge will guide this pairing process. From an experimental perspective, it will be interesting to learn if there are simple heuristics for this process or if it is up to building-in strict preconditions on player-model structural elements to make the design process work.

Constructing a new game or player from an old one and a structural unit requires selecting a known game or player, and finding a compatible structural unit which is not forbidden by the how-to rules, adding the construct, storing and indexing the new artifact, and then performing any required steps required by additional how-to knowledge. It is possible for the system to focus on only the mechanics of a game by only assembling the predicates used by my logical game engine. However, given examples, the system may also be able to propose structures for the predicates used to build a game's interface board-game-like interface. Unfortunately, this interface logic can only be understood in the context of human play testing (as machine play testing focuses only on the mechanics).

Getting a machine-generated trace for a model, following the practice established in the BIPED project, involves applying model-finding tools game's representation, using the logical game engine as a background library. Depending on the context, the system may add any number of constraints on the traces to be generated. Forced-move, non-existence, and indirect-dependence laws are easy to formally verify in this way (expecting no models to be found that illustrate the negation of the statement of the law). However, human players make a much more interesting source of evidence from which to conjecture laws. Getting human-generated traces will work exactly as it does in BIPED currently.

4.2.8 Discovery Moves (Actions)

While design moves primarily create new content for the artifact library by applying the current design theory as a fixed program, discovery moves directly modify the design theory. These moves will consist of proposing new what-is structures, new how-to rules, performing taxonomy maintenance and generating notebook manipulation activities.

What-is structures extend the system's vocabulary by coming up with new names for old things. To generate new structural elements, the system may (until I think of a better method) generate several valid but possibly uninteresting candidate structures. It should filter these candidates by trying to find instances of them in artifacts from the library. Finally, given sufficient evidence for the existence of that construct "in the wild" the system should propose it as a new element in the design theory. This process of identifying and naming new abstractions follows the long tradition of empirical induction in past discovery systems. Primitive structures (such as raw game state and events) allow the slow and unintelligent explanation of

initial games, from which, more useful structures will be found to form better explanations. As for relations from structures to regularities in traces, I expect off-the-shelf inductive logic programming to be sufficient for conjecturing data-supported hypotheses.

Recall that the how-to rules summarize the system's design experience. If I decide that the system should manage its own how-to rules, I imagine that the process would take the form of another empirical induction task, this time looking abstracting rules from recorded logs of design actions.

Managing the taxonomy is something that I have not thought through in detail yet. In general, the goal of taxonomy maintenance is to keep an index up to date so that useful concepts are easy to recall. I cannot say more until a good structure for the taxonomies (which are distinct from the artifact libraries which they aim to organize) emerges from my discovery practice.

I describe the final category of discovery moves as notebook manipulation tasks. The high-level of process of experiment design will likely be realized as an inspection of recently added design theory elements, some recall from the artifact library, followed by the writing of a rough plan and predictions into the notebook. As the system progresses through its tasks, it will eventually compare the result of experiments to their documented expectations, and write additional notes to the notebook, triggering further actions. While it may be possible to read most of the notebook manipulation tasks as the execution of a reactive planner, I hope to find a more nuanced, game-design-discovery interpretation.

4.2.9 Fixed Design Tools

Outside of the core discoverer, there is still more that I propose to do. Specifically, preparing effective tools for the discoverer to use is nearly as important as implementing the discoverer itself. The goal for the fixed design tools is enable a kind of mechanized game design.

Consider the metaphor of a carpenter producing wooden chairs. A table saw, lathe and wooden jigs make a carpenter more effective. They make it easier to produce the kind of artifacts the carpenter intends to produce at the cost of removing the expressive power of (and the need for expertise with) a whittling knife. Certainly, a reasonable "robot carpenter" would make use of such "power tools". Incidentally, the robot in the Robot Scientist project uses the power tools of biology (including several other robots as tools) to effectively perform tasks such as specimen preparation, handling, incubation, and observation.

What are the power tools of game design? There is no canonical list, however I have identified some that may be useful. Some exist as complete, freely-downloadable software tools, and others I will have to invent. Fortunately, figuring out what these tools are and how they fit into the game design process is also being tackled in Mark Nelson's parallel research [25].

The first tool that I intend my systems to use is a simple game-design validator. The BIPED system includes a very simple design validation component which can be seen as a prototype of this tool. Specifically, this validator should focus on syntactic properties of a design. Simple checks would include verification of the absence of trivial contradictions in a game's rules and the presence of definitions (such as critical predicates) that are required by other tools. Beyond this, elementary game-ness tests could be included into a design validator by adopting a detailed definition of games such as Juul's[12]. Juul's definition entails several required properties of games, some of which seem amenable to syntactic checking such as basic interpretations of "quantifiable outcome", "valorization of outcomes", and "negotiable consequences". Other aspects of Juul's definition which require semantic, even subjective, evaluation of a game design should be left to the core discoverer to keep the design validator simple, and reusable in a variety of contexts.

Next, a set of trace-finding tools should automate the extraction of world and play traces from designed artifacts. The BIPED system includes functionality for finding traces given certain constraints via a propositionalizing, satisfiability-based approach[13]. As needed (if critical performance or expressivity bottlenecks are discovered), I will explore alternate approaches which may include integration of recent satisfiability-modulo-theories work, first-order methods, or randomized forward-simulation (which is incompatible with certain classes of trace constraints). BIPED's approach to eliciting play traces from human players appears to be satisfactory for the time being.

[12] `http://www.jesperjuul.net/text/gameplayerworld/`
[13] `http://www.tcs.hut.fi/Software/smodels/`

When certain elements of a trace are unexpected, it is not immediately clear which aspect of the design of an artifact to blame. As such, a tool, which I am tentatively calling a "logical debugger" is needed to assign this blame. Such a tool, given an artifact and a trace which is not compatible with it, should be able to come up with a sets of rules that would need to be added or removed to allow the trace. While there is always a trivial solution ("remove all of the rules and add new ones that require only the given trace to be reproduced, exactly, step-by-step") I expect the debugging practices I identified in my earlier experiments (annotating rules with abducible flags that indicate various rules should be ignored) to be useful in producing much more interesting solutions. Additionally, the SPOCK system [3] serves as a principled foundation for this process; however, as distributed, it is not compatible with the particular tools I have been using so far.

While I intend my logic debugger to be primarily a game design diagnosis tool, similar approaches can (and probably should) be applied to the theory formation and revision processes at the design level. Specifically, inductive logic programming and abductive logic programming [31] tools provide off-the-shelf components for manipulating the representations of design theories I am intending to use. These theory-manipulation tools are better seen as discovery power tools than game design power tools, though they will require some additional work to be bound to my representation schemes. In addition to logic learning tools, other freely-available statistical-relational learning tools such as Alchemy[14] may be useful to automate the extraction of general rules from statistical regularities. While such activities are normally considered first-class discovery activities, in my system I relate to such revision and conjecture activities as invocations of external, domain-independent discovery tools – the knowledge that guides the discoverer to use such tools will be, by design, specific to the game design domain.

Finally, any other tools produced from the game design workbench project [25] are candidates for consideration as fixed design tools in the context of my intelligent game designer. Several candidates are: a richer validator, a design query suggester and query answerer, and game rule visualization tools.

4.3 Experimental Validation

The system experiments I propose are intended to validate the answers to my research questions.

4.3.1 How Does an Intelligent Game Designer Function?

First, to verify that the "games" my system produces are valid, I propose a small, player evaluation. This evaluation need only be carried out by a few experienced video game players. My goal is for the games my system produces to be recognizable as video games. Concretely, I will evaluate rule sets produced by the system with user interface logic created by hand as well as rule sets with interface logic automatically produced by the system. If it should turn out that games with manually-created interfaces are seen as games and those with automatic interfaces are not, I will try to determine the role of the manually created interfaces in producing this reaction. Using a random, primitive rule set generated by running the system with only minimal design knowledge I should be able to tell if *any rule* system visualized by hand appears to be a game or only those resulting from a detailed, learned design theory.

It should also be interesting to evaluate the games produced by the system as creative artifacts, to ask whether they are novel and/or valuable. However, a negative response to this question (by these same player-evaluators) is acceptable as the system's creative efforts are primarily directed at knowledge production.

Looking at the "intelligent" nature of the intelligent game designer, I should focus on the knowledge produced by the system – the result of its learning. As the output of a discovery system, there are traditionally two modes of evaluation that map back to the original varying motivations of discovery: to explain historical discoveries and to produce new knowledge.

To validate the knowledge produced on the basis of historical discoveries, I will need to have, in hand, some historical discoveries for reference. The Game Innovation Database[15]. has a collection of such historical discoveries, however they are not described in a language compatible with the knowledge my system aims to discover. As I would like these discoveries to be made with the same tools my system is using, I propose an

[14] `http://alchemy.cs.washington.edu/`
[15] `http://www.gameinnovation.org/`

expert evaluation whereby I ask game design experts to familiarize themselves with tools used by the system and carry out some discovery on their own. If the system is able to reproduce discoveries made by the expert designers, this is a positive validation of my system as an intelligent game designer in a history-oriented sense.

If expert designers are not able to produce any discoveries using the tools, then there are serious problems with my theories and this is a negative validation overall. However, initial experience with BIPED and subsequent manual discovery suggests that the formal representations I intend on using are quite ripe with patterns to be discovered, named, and rediscovered by my system (recall the "per-entity bit" pattern from my prior work) – and this is not even considering knowledge which predicts trace patterns from the construction path of games, which I have not yet attempted in my manual discovery.

Concretely, I am looking to validate three types of knowledge structures (two are what-is and one is how-to): definitions of structural design patterns in games, players and play; relations between these structures and the contents of game and play traces; and the heuristic knowledge of when to always (or never) apply one design move after another.

To validate the knowledge produced on the basis of being original discoveries, an obvious plan is to find patterns that my system discovers that expert designers from the previous evaluation did not. To get a more detailed evaluation of the knowledge my system discovers, these same expert designers-evaluators will be asked to comment on the various design theories. The comments should appraise the patterns as both as creative artifacts (looking for novelty and value, or justification and documented surprise in the reflexive case) and as literal design patterns (answering the question "Would you apply this pattern in your own designs?").

One final aspect of testing the "intelligent" nature of the intelligent game designer is to ensure that it is not "cheating" in obvious ways. In evaluation, the various *reasonable* ways in which the system might be considered to be cheating should be cataloged. If the system is cheating in some way required of it by the theory, then I have run into a problem, otherwise this may just be a fluke of the implementation of my particular system. If it is revealed that the representation language used to encode design theories is particularly good at representing interesting design discoveries (to the point of suggesting that any search process could have discovered these interesting theories, whether it was "intelligent" or not), this is still a win for my project as it would show that the theory of game design embodied by the use of those representation was a good one.

To evaluate the sense of "game design" used in my system, I propose two experiments. In the first, I should validate that a human is capable of using the same tools to produce interesting games. That is, it should be possible for expert game designers to create examples of games they like using the design tools. In the second, I should ensure the system is capable of producing qualitatively similar games. This may be as simple as ensuring the system can make meaningful predictions about the expert-produced games.

Various pieces of the answer to "how does an intelligent game designer function?" take the form of theories – the knowledge-level account of game design in particular. For now, the primary way I can imagine testing the theories is to validate the systems produced according to them.

During development, there is one particular question I should keep in mind with respect to the aims of building an intelligent game design: How will I know my system is discovering "play" and not "abstract state-progression systems" or other nonsense?

My answer to this is that I am designing my system to deal with the live interaction of human players as a first-class interest of the system. Though I can imagine building a system that did little more than collect massive piles of observations from human players, such a system would spend most of its time waiting. Making a move similar to the one made by the authors of NEvAr [13], I want to build a detailed model of play from sparse observations of humans that allows the same multiplying effect. This should allow the system to perform many more game design experiments than it would otherwise. *If the system's theory can successfully predict and explain a human player's interaction with a game in terms of clear, structured assertions, and the players feel like they are playing real games, then the system really is discovering play.*

4.3.2 What Does Such a System Imply for the Relationship Between Discovery and Expressive Domains?

My primary plan for answering my secondary research question is by generalization from experiments with my system. As I come up with explanations for, say, how game design is like a science (or the other way around), I should test these experiments by toggling the relevant parts of my system's architecture. In this way I should be able to find out which architectural elements are really responsible for the desirable behavior.

At the time of writing this proposal, all of the experiments that might validate my working understanding of game design as a science all start with building a system according to that theory. In my time line below I budget time for inventing the experiments that will let me test my theoretical generalizations.

During development, there is another question to keep in mind with respect to the aims of my second goal: How do I know my games count as expressive artifacts – are my generalizations to expressive domains sound?

For now, my answer is that my theory should have a clear place to put in the designer's goals and describe how these goals interact with the models of audience interaction in terms of play. Currently, my theories do not support this, so it is something I need to address going forward.

4.4 Time Line

4.4.1 Year One

My first year of research will focus on stretching game design into a science-like practice. I expect automated discovery to only emerge at the very end.

- Summer 2009: "play with more manual discovery"
  - Flesh-out first system architecture (identifying and describing all compontents in details)
  - Perform manual discovery with the raw tools and representations I have currently (identifying some game-level knowledge to be re-represented in with better tools, obtaining a reference for how to accomplish formal game-design discovery tasks, observing what conventions for the notebook emerge, sketching operational knowledge, and proposing any new tools that are found to be lacking)
- Fall 2009: "figure out those scientific knowledge structures"
  - Write up game modeling language with Mark Nelson
  - Assert knowledge structures (deciding on representation for concepts such as rule set sub-structures and player model elements, deciding on representation for conjectures such as implications for trace properties in terms of sub-structures, deciding on taxonomy structures, and specializing the tools in-hand to work with these representations
  - Perform more manual discover using these representations and record activities
- Winter 2010: "figure out those scientific processes"
  - Write up representations and a position paper on discovery using them
  - Assert processes (including fixed heuristics for when to modify the various slots in the various representations and triggers for when to make certain kinds of conjectures ala BACON)
  - Perform more manual discovery using these processes as the elementary moves
- Spring 2010: "integrate that system"
  - Write up preliminary view of game design as a scientific domains (with knowledge structures and processes in-hand)

  - Assert (initial) complete operational knowledge for the designer
  - Assert clear design moves (which have applicability rules and a means for predicting outcomes)
  - Perform system integration (it is working if basic theory goes in and a refined theory comes out and a human was asked to play a few games in the mean time)
  - Design/Arrange human-related evaluation (with an existence proof that it is possible to build an intelligent game designer, requires players to play some games, designers to encode some knowledge, comment on knowledge, and produce some games)

4.4.2 Year Two

My second year will focus on architectural experimentation, evolving the system, and generalizing to my desired theoretical contributions (and writing the dissertation, of course).

- Summer 2010: "plot a course"
  - Write up initial results from integrated system (and its architecture)
  - Perform final literature review (reviewing the state of mechanized design and theory formation and the state of artifact generation in expressive domains)
  - Formulate initial implications between discovery and expressive domains (ideally employing affordances and semiotics)
  - Produce a detailed thesis outline with seconds conditioned on experimental results
- Fall 2010: "perform those experiments"
  - Design architecture-based experiments
  - Carry out experiments
- Winter 2011: "synchronize results"
  - Reconcile experimental results with proposed implications (given that I now have which architectural features were critical and how, will not worry about repairing all implications unless it is easy, just record the evidence for and against my guess)
- Spring 2011: "finish writing and defend"

5 Closing

I have just proposed a thesis in the field of artificial intelligence, specifically one that ties together expressive intelligence, computational creativity, and automated discovery. Beyond these areas I expect there to be some interest in game design (both in terms of the knowledge contained in the output design theories as well as the the story of design as a discovery process). In more general artificial intelligence I imagine the architecture of my "design agent" and integration of active learning to have broader interest. Finally, I expect the various tools and processes developed in assembling the system will be of great interest to game developers in the form of stand-alone parts (particular machine play testing, automatic analysis, a vocabulary of actionable design issues, and various representations as a sketch for new kinds of game engines optimized for exploration instead of raw performance).

As I reach the end, I want to note a view of my system that I had left unacknowledged through the forgoing discussion. Consider an alternate picture of my intelligent game designer, not as embodiment of a general theory of game design, but as an interactive art project, an executable caricature of game design as a science and as a unit of digital media to be analyzed from a multitude of perspectives.

References

[1] Staffan Bjork and Jussi Holopainen. *Patterns in Game Design (Game Development Series)*. Charles River Media, Inc., Rockland, MA, USA, 2004.

[2] Margaret Boden. Computer models of creativity. In *Handbook of Creativity*, chapter 18, pages 351–372. Cambridge University Press, 1998.

[3] M. BRAIN, M. GEBSER, J. PHRER, T. SCHAUB, H. TOMPITS, and WOLTRAN. "that is illogical captain!"–the debugging support tool spock for answer-set programs: System description. In *Workshop on Software Engineering for Answer Set Programming*, pages 71–85, 2007.

[4] Bruce Buchanan. Creativity at the metalevel: Aaai-2000 presidential address. *AI Magazine*, 22(3), 2001.

[5] Simon Colton. Automated puzzle generation. In *In Proceedings of the AISB'02 Symposium on AI and Creativity in the Arts and Science*, 2002.

[6] Simon Colton. *Automated Theory Formation in Pure Mathematics*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.

[7] David Cope. *Computer Models of Musical Creativity*. The MIT Press, 2005.

[8] Susan Epstein. Learning and discovery: One system's search for mathematical knowledge. *Computational Intelligence*, 4(1):42–53, 1988.

[9] DH Feldman, M Csikszentmihalyi, and H Gardner. *Changing the world: A framework for the study of creativity*. Praeger Publishers, 1994.

[10] Tracy Fullerton. *Game Design Workshop: A playcentric Approach to creating innovative games*. Morgan Kaufmann, 2008.

[11] Martin Gunther, Stephan Schiffel, and Michael Thielscher. Factoring general games. In *In Proceedings of GIGA'09 – The IJCAI Workship on General Game Playing*, 2009.

[12] Ken Haase. Discovery systems: From am to cyrano, 1987.

[13] Penousal Machado Instituto. Model proposal for a constructed artist, 1997.

[14] Peter D. Karp. Hypothesis formation as design. In Jeff Shrager and Pat Langley, editors, *Computational Models of Scientific Discovery and Theory Formation*, chapter 10. Morgan Kaufmann, San Mateo, California, 1990.

[15] R. D. King, K. E. Whelan, F. M. Jones, P. G. Reiser, andS. H. Muggleton C. H. Bryant, D. B. Kell, and S. G. Oliver. Functional genomic hypothesis generation and experimentation by a robot scientist. *Nature*, 427(6971):247–252, 2004.

[16] Raph Koster. *Theory of Fun for Game Design*. Paraglyph, November 2004.

[17] Deepak Kulkarni and Herbert A Simon. Experimentation in machine discovery. In Jeff Shrager and Pat Langley, editors, *Computational Models of Scientific Discovery and Theory Formation*, chapter 9. Morgan Kaufmann, San Mateo, California, 1990.

[18] P. Langley, G l Bradshaw, and H A Simon. Rediscovering chemistry with the bacon system. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, pages 307–329. Springer, Berlin, Heidelberg, 1984.

[19] C. E. Larson. An updated survey of research in automated mathematical conjecture-making.

[20] Douglas Lenat. Automated theory formation in mathematics. In *Fifth International Joint Conference on Artificial Intelligence*, pages 833–841. Morgan Kaufmann, 1977.

[21] Douglas Lenat. Eurisko: A program that learns new heuristics and domain concepts. *Artificial Intelligence*, 21(1-2):61–98, March 1983.

[22] Erik T. Mueller. *Commonsense Reasoning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[23] Stephen Muggleton, Wolfson Building, and Parks Road. Inverse entailment and progol, 1995.

[24] Mark J. Nelson and Michael Mateas. Towards automated game design. In *AI*IA '07: Proceedings of the 10th Congress of the Italian Association for Artificial Intelligence on AI*IA 2007*, pages 626–637, Berlin, Heidelberg, 2007. Springer-Verlag.

[25] Mark J. Nelson and Michael Mateas. A requirements analysis for videogame design support tools. In *4th International Conference on the Foundations of Digital Games*, 2009.

[26] A. Newell. The knowledge level. *Readings from the AI magazine*, pages 357–377, 1988.

[27] J. Orwant. Eggg: automated programming for game generation. *IBM Syst. J.*, 39(3-4):782–794, 2000.

[28] Alison Pease, Daniel Winterstein, and Simon Colton. Evaluating machine creativity. In *In Workshop on Creative Systems, 4th International Conference on Case Based Reasoning*, pages 129–137, 2001.

[29] Zachary Pousman, Mario Romero, Adam Smith, and Michael Mateas. Living with tableau machine: a longitudinal investigation of a curious domestic intelligence. In *UbiComp*, pages 370–379, 2008.

[30] Oliver Ray. Automated abduction in scientific discovery. *Studies on Computational Intelligence*, 64:103–116, 2007.

[31] Oliver Ray, Krysia Broda, and Alessandra Russo. A hybrid abductive inductive proof procedure. *Logic Journal of the IGPL*, 12(5):371–397, 2004.

[32] Katie Salen and Eric Zimmerman. *Rules of Play : Game Design Fundamentals*. The MIT Press, 2003.

[33] Rob Saunders and John S. Gero. The digital clockwork muse: A computational model of aesthetic evolution. In *The AISB'01 Symposium on AI and Creativity in Arts and Science, SSAISB*, pages 12–21, 2001.

[34] Jeff Shrager and Pat Langley. Computational approaches to scientific discovery. In Jeff Shrager and Pat Langley, editors, *Computational Models of Scientific Discovery and Theory Formation*, chapter 1. Morgan Kaufmann, San Mateo, California, 1990.

[35] Adam Smith, Mario Romero, Zachary Pousman, and Michael Mateas. Tableau machine: A creative alien presence. In *AAAI Spring Symposium on Creative Intelligent Systems*, 2008.

[36] Adam M. Smith, Mark J. Nelson, and Michael Mateas. Computational support for play testing game sketches. In *5th Artificial Intelligence and Interactive Digital Entertainment Conference*, 2009.

[37] Gillian Smith, Mike Treanor, Jim Whitehead, and Michael Mateas. Rhythm-based level generation for 2d platformers. In *Proceedings of the 2009 Int'l Conference on the Foundations of Digital Games*, 2009.

[38] Julian Togelius and Jrgen Schmidhuber. An experiment in automatic game design. In *2008 IEEE Symposium on Computational Intel ligence in Games*, 2008.

[39] Scott Turner. *The Creative Process: A Computer Model of Storytelling and Creativity*. Lawrence Erlbaum, 1994.

[40] Jose Zagal, Michael Mateas, Clara Fernndez-Vara, Brian Hochhalter, and Nolan Lichti. Towards an ontological language for game analysis. In *Digital Interactive Games Research Association Conference*, 2005.

[41] J. M Zytkow. Creating a discoverer: an autonomous knowledge seeking agent. In *Foundations of Science*, pages 253–283. Springer, 1995.