A Requirements Analysis for Videogame Design Support Tools

Mark J. Nelson
School of Interactive Computing
Georgia Institute of Technology
Atlanta, Georgia, USA
mnelson@cc.gatech.edu

Michael Mateas
Expressive Intelligence Studio
University of California, Santa Cruz
Santa Cruz, California, USA
michaelm@cs.ucsc.edu

ABSTRACT

Designing videogames involves weaving together systems of rules, called game mechanics, which support and structure compelling player experiences. Thus a significant portion of game design involves reasoning about the effects of different potential game mechanics on player experience. Unlike some design fields, such as architecture and mechanical design, that have CAD tools to support designers in reasoning about and visualizing designs, game designers have no tools for reasoning about and visualizing systems of game mechanics. In this paper we perform a requirements analysis for design-support tool for game design. We develop a proposal in two phases. First, we review the design-support-system and game-design literatures to arrive at a plausible system that helps designers reason about game mechanics and gameplay. We then refine these requirements in a study of three teams of game designers, investigating their current design problems and gauging interest in our tool proposals and reactions to prototype tools. Our study finds that a game design assistant that is able to formally reason about abstract game mechanics would provide significant leverage to designers during multiple stages of the design process.

Categories and Subject Descriptors

H.5.0. Information interfaces and representation (HCI)
J.6.b. Computer-aided engineering: Computer-aided design

Keywords

authoring tools, videogames, game mechanics

1. INTRODUCTION

Videogame design is a creative design domain in which creativity is fundamentally expressed through engineering interactive rule systems: a game designer combines a set of *game mechanics* such that, when they interact with each other and with the player's actions, they produce the desired gameplay.

Game designers typically prototype these rule systems to understand how they operate. Prototypes range from paper mockups, in which a stripped-down form of the game's rule system is simulated manually, to playable versions implemented on a computer, which can be played by the designer and others to get feedback on gameplay ideas or to discover problems.

Prototypes aim to answer both subjective and objective design questions. The ultimate design questions are mainly subjective: is the game interesting, fun, challenging, balanced, and so on? However, much prototyping gets at these questions indirectly by answering objective questions that help the designer understand how their rule system operates. For example, is there a way to win with a particular combination of items? Can the player ever get to a particular bit of story without having gotten the appropriate set-up? Are there weapons that are redundant because they're never the best choice?

In previous work, we've proposed that the objective kinds of reasoning questions are amenable to being answered by automated methods, and demonstrated a reasoning system based on logical inference in the event calculus, a representation for reasoning about states, events, and change over time [17]. Allowing designers to query a rule system increases what design researchers call the *backtalk* of the design situation [25], allowing the designer to focus on making subjective decisions rather than on working out the implications of those decisions by hand.

That work demonstrates the feasibility of automatically answering at least some objective design questions that we think may be useful in design. However, particularly since there has been little work studying how game designers actually go about their design processes, it remains to be determined what exactly designers would want out of such a system, and how they would use it in their work.

Here we undertake a study with several teams of game designers, in order to investigate what sorts of queries about mechanics they would find useful to be able to get automated answers to during their design processes. This analysis is intended to arrive at a set of functional requirements for the AI reasoning system, which will drive the next iteration of both technical AI work in implementing those features, and user-interface work in providing a way for designers to use them comfortably. Since little work has been done on the broader subjects of game-design ethnography, game-design assistant tools, and so on, we also collect the designers' views on what other game-design tools they might find useful and how they might want to interface with them. Those more exploratory ideas partly take the form of following up on negative results—designers who didn't want the mechanics-reasoning tool we were proposing, but had interest in something different.

The goals of this project have some similarity to work in computer-aided design (CAD)—especially early work before focus shifted to 3D modeling—which also aims to integrate automated reasoning into the design loop, allowing the designer to try

ICFDG 2009, April 26–30, 2009, Orlando, FL, USA.

out design ideas rapidly without having to work out technical details (such as whether an idea has problems with structural integrity). Therefore, we begin by reviewing the literature on CAD and subsequent design-support approaches, which have gone through several decades of requirements analysis, implemented systems, and debate in these other design domains.

There is also an existing literature on videogame design. Although there has been little to no game-design ethnography in a descriptive, sociological sense, there are a number of writings on the game-design process from a more how-to, textbook perspective. These summarize conventional wisdom for how the game-design process ought to operate, at least in an idealized sense. We draw from that literature, combined with the design-support literature, to develop plausible ideas for how design-support tools can factor into the videogame design process, considering a broader space of possible tools within which the specific mechanics-reasoning tool we're building is situated.

From that starting point, we conducted a contextual-design study with three teams of independent game designers, to determine if and how the mechanics-reasoning system we propose could be useful for answering design questions that they had about game designs they were working on at the time of the study. This study was aimed in part to validate and revise our ideas, drawn from the design-support and game-design literature, on how a game-mechanics reasoner would fit into their design practices. More directly, it aimed to identify specific types of queries that the designers would find it useful to be able to get automated answers to. Based on this series of interviews, which included designers interacting with focusing prototypes of possible tools, we refined our model of the early-stage game design process and the support a tool could provide in this process.

2. DESIGN-SUPPORT SYSTEMS

Ideas on how to integrate computers with the design process are nearly as old as practical electronic computers. The earliest relevant work, from 1956, conceives of a conversational process between designers and machines, in which the machine carries out tedious calculations involving material properties, while the designer makes high-level design decisions [19,22]. In the terminology of Schön's influential view of design as a reflective conversation with the design situation [25], this is in retrospect a proposal for the machine to participate in the design conversation by increasing the backtalk of a situation, crunching numbers to illuminate current constraints and implications.

2.1 Initial development of CAD

The first serious requirements analysis for a design-support system determined that it should have a graphical input method that would allow designers to make and modify sketches; the system would both display refined sketches back to the designer, and simultaneously convert them into internal representations on which automated numerical analyses (such as stress analysis) could be performed. The result would be a system that should function in two roles: "at some times, it would be the designer's slave, at others it would alert the designer to impossible requirements or constraints being imposed" [20, pp. 95-96]. This dual view of a backend automated reasoning system coupled with a front-end interactive modeling tool has gone through a series of evolutions, with parts variously emphasized or de-emphasized.

Early work on backend reasoning showed that designers were willing to try out more modifications when automated stress analysis was available, and also began exploring giving designers computerized parts catalogs, both so they could simulate the physical properties of a known part, and quickly retrieve parts with specific desired properties [20, pp. 104-105].

Sutherland's Sketchpad system provided a front end, with a light pen and real-time graphics display that were at the time quite novel [27]. He showed some advantages to computer sketching over paper sketching, such as being able to precisely draw diagrams with large numbers of components (especially repetitive ones). He nonetheless concluded, "it is only worthwhile to make drawings on the computer if you get something more out of the drawing than just a drawing", so rather than positioning Sketchpad primarily as a computer drawing tool, he positioned it as an "man-machine graphical communication system", with sketching the input method by which a designer communicated design information to the backend reasoning systems. To that end, it supported semantic annotations about the meanings of lines in the sketch and their relationships to each other, allowing, for example, force-distribution analysis on a sketch of a truss bridge, or simulation of sketches of electronic circuits [27, pp. 137-138].

In contrast, later systems did see interactive modeling and production of design diagrams as a major use of CAD [28, ch. 6], and an influential line of work developed a set of graphically editable three-dimensional surface primitives that could be combined to produce arbitrary shapes [3].

2.2 Supporting domain knowledge

A second wave of systems, coinciding with a shift from the engineering to the design community, took design-support systems in several different directions, mostly focusing on the importance of knowledge in specific design domains.

In many domains, vocabularies and representations have evolved to encode useful ways of thinking about problems. Using a generic set of geometric surfaces as the representation for all design problems was criticized for losing that domain-specific knowledge, at worst encouraging a design style that leads to visually impressive but poor designs, akin to using a lot of fonts and visual effects in desktop-publishing software [13]. Even when it didn't have outright negative effects, the focus on visual modeling led to criticisms that CAD was failing to fulfill its original vision as a design assistant, and instead serving a narrower role as computerized draughtsman [14]. An early attempt to improve that situation built a domain-specific tool for roof design, using a traditional architects' vocabulary of ridges, verges, valleys, eaves, hips, and so on—representations that bring relevant design questions to the fore, such as the relationship between structural support and space enclosure, and interior and exterior surfaces [21].

The development of knowledge-based AI systems in the 1980s provided an opportunity to bring automated reasoning to these kinds of symbolic representations (rather than numerical simulations like stress analysis). For example, if a building were designed using terminology from municipal codes (windows, floors, hallways, etc.), and the municipal codes themselves were encoded in a design-support system, the system could determine which parts of the fire code applied, and whether a design met them [9].

Domain-oriented design environments (DODEs) combine and extend several of these approaches [5]. They start with building blocks meaningful in a particular domain (e.g. sinks, counters, ovens, and windows for kitchen design) and allow designers to compose them into higher-level representations. They extend the

idea of domain-specific knowledge to include not only factual knowledge (such as structural soundness or building codes), but also *design* knowledge, such as best practices and common solutions. This knowledge can be employed to do things like critique a proposed design (the sink isn't in front of a window), or to provide design suggestions and the reasons for them (the sink should be placed near the range, due to common workflow), shifting the computer's role in the design conversation from providing backtalk to actively participating on the design side as well [7].

Knowledge-based systems run into the problem that few domains are well-defined and static enough to effectively capture domain knowledge in a tool that can be built and deployed to users, leading to the necessity of *open systems* that can be evolved and extended [11]. Applying that principle to DODEs, they've been extended to support designers evolving (and sharing amongst each other) their representations and design knowledge [6], which has developed into a concept of *metadesign* systems that support the designer not only in a specific design domain, but in the process of specifying and evolving the design domains themselves [10].

A different line of work at around the same time argued that CAD had fundamentally erred in being based around graphically interacting with a drawing—that of the two things that can be found in any design office, namely conversation and drawings, the conversation was where the design took place, with the drawings being secondary, and mainly representing the end result of design [15]. In particular, this work argued that, early in the design process, there is rarely a single design in progress for which there could be a drawing, but instead many, often disconnected, bits and pieces of design goals, tentative conclusions, design decisions, and ideas being pursued in parallel. Although admitting that drawing does play a role in this process, this work instead built a prototype system that converses textually with the designer, learning about his or her design goals, bringing them up later as reminders, making suggestions, critiquing ideas, answering questions, and so on.

2.3 Creativity support

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

Schön proposes four main uses of a design system: enhance the seeing-drawing-seeing loop, allow construction and exploration of microworlds, help manage a repertoire of prototypes and apply them in specific design situations, and allow the designer to discover and reflect on their design knowledge [26].

Lawson and Loke propose five roles for a system in the design conversation: learner, informer, critic, collaborator, and initiator [15]. As a learner, the system makes note of the design goals and preferences of the designer, facts about the current design situation, proposed ideas or design decisions, and justifications for decisions or preferences. As an informer, it answers questions based on what it knows so far. As a critic, it checks the validity of comments the designer makes, and warns if there are problems with proposed design decisions. As a collaborator, it tries to elaborate on the designer's comments or proposals. As an initiator, it jumpstarts dead-ends by starting new lines of discussion or suggesting new perspectives on a problem.

Giaccardi and Fischer [10] propose that a creativity-support system needs to help designers cope with ill-defined problems by integrating problem *framing* with problem *solving*; to support reflective conversation with the design situation; and to support sharing of knowledge among people with different perspectives and backgrounds. To get at this, they propose (among other things) that design-support systems need to have embedded critics; need to support reuse and sharing of design representations and solution; and need to support collaboration among designers.

3. VIDEOGAME DESIGN SUPPORT

Given the lessons of several decades of design-support systems, what should go into a videogame design-support system? There is little research studying the work processes of videogame designers that can be used to inform design-support systems. However, quite a bit has been written on game design from a more idealized perspective—designers' views of how games are, or at least ought to be, designed—that can be used as a starting point.

Nearly all treatments of games agree that designing games begins with the design of game mechanics, the systems of rules that evolve the game state over time and in response to interaction [1,8,24]. One guide for how to prototype games suggests stripping away all the "window dressing" of a game to focus on a simple model of just the mechanics, even acting them out on paper or with physical models, "to allow you to wrap your brain around the game mechanics and see how they function" [8, ch. 7].

The primary design issue with game mechanics is figuring out how they interact to produce the gameplay: the challenges, rewards, and decisions encountered by a player. Games are built by adding, removing, or changing mechanics, but design goals are usually formulated in terms of gameplay rather than mechanics themselves. An influential view describes this as a mechanics/dynamics/aesthetics hierarchy: *mechanics* define the rules of the game, which interact with each other and the player to produce the *dynamics* of gameplay, which in turn interact with the game's art, cultural context, and the player's preferences to produce the *aesthetics* of the game [12]. A good starting point for a design-support system might therefore be helping the designer to negotiate the relationship between mechanics and dynamics.[1]

Dynamics are largely defined by the interaction of mechanics with each other to produce *constituative mechanics*, the set of rules that are logically implied by the game mechanics [24, ch. 12]. Although constituative mechanics are not literally game mechanics in the sense of being explicitly represented in the game (those are the *operational mechanics*), they nonetheless describe how the game's rule system operates. For example, if due to the constraints imposed by various mechanics there are no ways to win a game without acquiring a particular item, then a constituative mechanic of the game is that the player needs to acquire that item in order to win, even though it isn't explicitly a win condition.

Negotiating the relationship between operational mechanics and constituative mechanics is particularly well suited to automated reasoning, since constituative mechanics are quite literally the sets of rules that can be formally derived from the operational mechanics. Designers get at this relationship by building simple prototypes, with everything except the bare rule system stripped out (the visual representations employed by such prototypes are typically abstract geometric figures, such as circles and triangles), in order to figure out how the rule system operates. Allowing auto-

[1] We have separately done some preliminary work to assist with the aesthetic design problem in the context of *skinning* games, i.e. mapping graphical elements onto a game's elements [16].

mated derivation of the logical implications of a set of game mechanics can speed up this process considerably. By making implications of a set of game mechanics immediately available, the assistant increases the "backtalk" of the design situation and allows for quicker design iterations.

In terms of Lawson and Loke's roles, a design-reasoning system primarily plays the informer role, answering questions from the designer about the implications of a set of mechanics. We answer these queries using a formalization of game mechanics in the event calculus, a symbolic logic that represents state and change of state over time [17]. In this representation, various types of reasoning and queries can be done both forwards and backwards in time. A game can be played or simulated forwards in time. If we want to see if a particular state is reachable (say, player wins without ever getting the key), we can reason backwards in time to find a sequence of events that would result in that outcome. One obvious use is to find sequences that the designer thought *shouldn't* be possible, i.e. we thought our operational mechanics produced a constituative mechanic that they actually didn't (or vice versa), which is usually discovered by looking through debug logs collected in playtesting. Additionally, we can query for states or sequences of events meeting some criterion, such as all ways of beating a game in less than 10 seconds, or all enemies that could possibly be the first enemy the player encounters. The system can also play a critic role (again using Lawson and Loke's terminology), maintaining a set of such "should be possible" and "shouldn't be possible" invariants and checking them as the design is modified—something akin to software-engineering regression tests, but for game designs rather than their implementations. There are a number of possibilities for other such design queries; the purpose of our study in the rest of this paper is to understand which types of design queries and prototype reasoning are useful to designers engaged in a real game-design process.

A difference from many CAD systems is that this proposed mechanics-reasoning starting point has no real interactive modeling component. This is mainly because games have no canonical visual representation. While a 3d model of a building is a natural visual representation for a physical building, what is the natural visual representation for a game, which is fundamentally a procedural system (a process)? There do exist tools, such as *GameMaker* and *Alice*, that support novice designers in visually designing restrictive classes of games. Such tools, however, are aimed at easing game *implementation* rather than design, e.g. by making it easy to put objects on a 2d screen and have them move around, but not for helping designers think about the design space of possible mechanics that result in the movement of the objects on screen. Further, since such tools provide implementation support for restricted subclasses of games, they are not generally used by professional designers.

Professional designers do use graphical tools in a few limited contexts, such as level design, but levels are not really the core of a game design, and in fact the reliance on tools that make it easier to do things like add more levels to a game rather than improve its gameplay has been criticized [4, pp. 120-124]. If a visual tool promoted focus on visual design, as critics argued it has in some cases with CAD, this could also exacerbate a similar problem in game design, where some games overly rely on cosmetics to the detriment of good gameplay [4, pp. 107-115]. There may be ways to develop a visual representation of games that is more useful for design (and we collect some preliminary ideas in our study), but we start with requirements analysis for the backend reasoning as likely to be the larger short-term gain.

We do start with a set of simple primitives that can represent any game mechanic—state and state evolution rules, such as events causing a change in state, a combination of state causing events, events causing other events, and so on. This might seem similar to the way in which graphical modeling in CAD was criticized for using a generic set of surface primitives to represent all designs. A main difference here is that, whereas representing roofs as geometric surfaces did not follow traditional design representations, representing game mechanics as state and state evolution rules is precisely how current game prototyping is done, usually by directly writing C++ code that stores state in variables and calls functions to update their values.

Game designers have also periodically called for a design vocabulary to allow them to discuss higher-level design concepts [2]. If a well-developed vocabulary of that sort existed, along with associated rules of thumb for when to use various mechanics, a library of such concepts and design guidelines could be used to build a domain-oriented design environment, analogous to the one Fischer et al built for kitchen design [5]. The lack of a well-developed design science for videogames, however, means that we currently lack an agreed upon design vocabulary and guidelines to encode in such a design environment (though a restricted subset could perhaps be captured to produce a tool targeted at novices). The fact that game designers recognize a need for the development and exchange of a design vocabulary in the first place aligns better with Giaccardi and Fischer's suggestion that tools ought to support problem framing as well as problem solving [10], suggesting that one useful tool would be one that let designers build up and share libraries of game-design vocabularies and pieces. One way of integrating this with the mechanics-reasoning tool would be to define the higher-level vocabulary in terms of the low-level language of state and state-evolution rules.

4. INTERVIEW METHODOLOGY

To validate the concept of a game-design assistant that helps designers reason about the interaction of game mechanics, and to collect a set of requirements for the kinds of reasoning it should be able to perform, we conducted a study with three small teams of independent game designers, each of whom was in the midst of a design project. We followed a contextual design methodology [29], investigating to what extent a game-design assistant would be useful for the design questions they were facing at the time, by proposing and testing out design-assistant prototypes on the problems they were actually working on. Since game designers don't necessarily have a good model for what an assistant might actually be able to do for them, this required an iterative process of interviews with focusing prototypes, where feedback from one interview feeding the next focusing prototype.

We started by interviewing each team about their project, current design questions, and existing prototyping techniques. This wasn't intended as a full ethnographic interview, but rather as a lightweight process mapping that allows us to understand enough of their design practice to converse with them intelligently in the rest of the interviews, make sensible proposals, and refer to agreed upon elements of their design process [18].

From there, we proposed some scenarios where we thought a system that reasons about mechanics could provide answers to relevant design questions. These proposals varied from floating an

idea to see if it sounded interesting to the designer, to paper mockups of a hypothetical interface, to prototypes of a backend reasoning system that could provide answers in specific situations.

It's worth emphasizing that these prototypes were intended primarily to collect requirements for the automated-reasoning system, not at this stage for the interface. This AI-centered contextual-design approach differs from a more common methodology in HCI of doing interface-centered contextual design: prototyping non-functional interfaces in order to understand how a user would interact with a system, and, from that understanding, identifying required functionality. That approach, however, tends to work well only if the functional hooks are not overly complex. During the study, the researcher needs to tell the potential user what the system would have done if it were functional; and, after the study, the identified functionality needs to be implemented to the specifications. With complex AI systems, it is difficult to accurately tell the user what a system would have done if it had existed, and to gauge their response to this non-existent functionality; it is also fairly easy to identify wishlist features that turn out to be impossible to implement as envisioned. Therefore, we followed a functionality-first style of prototyping, identifying *what* it is designers would like such a system to be able to do. We served as the interface, translating designers' queries into the system, and translating the answers back for them; this allows us to get at what queries, if any, are useful, before we move on (in future research) to considering how to make it easy for designers to specify and interpret the queries. This approach does still contrast from a purely AI-driven development style, by applying HCI-derived methodology to the design of the AI system's features. We did also collect, in a more exploratory fashion, ideas for interfaces and graphical modeling, largely gleaned from paper mockups, sometimes proposed by us, and sometimes drawn by designers who had an idea of what they'd like to see in an interface.

A particularly strong participatory aspect of this design process was necessary for a number of reasons. There is little existing work on game design, so beginning with a purely observational study to understand how designers work, and using that as the basis for designing a system, would be unlikely to inform the design of a game-design assistant in the short to medium term. In addition, game design is, as with many creative design practices, quite idiosyncratic, with design styles often strongly influenced by a designer's personal design practices. As a result, a significant degree of deference is necessary to designers' control of their own artistic practices, and therefore we need specific opinions and reactions about how our proposed tool might fit into those practices. On a more practical level, independent game designers, the most likely early adopters of such a tool due to the focus on mechanics innovation in independent game design, have nearly complete control over the tools they use, so perceived usefulness is at the very least necessary for such a tool to be actually useful.

5. CASE STUDIES

We studied three teams of independent game designers, one consisting of a single individual, and two of two-person teams. Due to the sensitivity of publishing design information for games that have not yet been released, we partially anonymize two of the three case studies, by substituting similar examples from existing commercial games when specific references to game-design features are necessary, and discussing other design issues in general terms. The first case below, however, is discussed without anonymization by agreement with the designers, Chronic Logic.

5.1 Case study 1: NARPG

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

In this case study, the designer was in the later stages of design; the core mechanics (fundamental rule systems) had already been established when we began interviewing. The design effort was therefore focused on level design. During level design, given fixed core mechanics, the designer creates objects and spatial layouts that appropriately balance challenge and reward. Design questions that arose during level design for NARPG include: Are some items unnecessary, in that a player can effectively ignore them and still win? Are some items too powerful? Given a specific level (spatial layout, enemy encounters and objects), how well do various player strategies (as suggested by the designer) fare? When we began working with the designers, they were answering these sorts of questions via cycles of modifying and playing a prototype version of the game, which was more or less a working version of the game with placeholder art and interfaces.

We implemented a formalized version of their game mechanics in our reasoner, and worked with them to answer a series of design questions. One large category of design questions they had was what gameplay would be like for different types of players; for example, how would the player fare who always picks up the strongest armor and weapons they can find, uses health potions, and does nothing else? While forward simulations using the standard prototyping process could begin to provide insight on this question, the formal representation of the mechanics allowed us to generate simulated play traces with specific characteristics. Forward simulation within a traditional procedural prototype could require potentially millions of runs until one with the desired properties is generated. In addition, the designers were particularly interested in "backwards" reasoning from outcomes to mechanics changes, e.g. what the smallest or largest value for a particular quantity (health, sword strength, etc.) should be to still achieve a desired outcome. In general they had no shortage of design questions they felt comfortable posing within or formalized game mechanics framework, and found the mechanics-reasoning approach fairly easy to work into their design process. In fact, they wished we were further along than the requirements analysis phase as they really wanted a finished prototype with implemented front end that they could use within their design process.

An interesting query type that the designers brought up, and that we do not currently implement in our prototypes, involves finding player models that can achieve a particular outcome. This type of reasoning would be able to generate different hypothetical play styles given particular outcomes. In the context of our event-calculus back end, answering such queries would involve logical induction, which is an avenue we shall certainly investigate.

5.2 Case study 2: A real-time strategy game

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

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

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

Since the goal of our requirements analysis at this stage was to collect requirements for the reasoning backend, we mainly used prototypes of a visual design language as focusing prototypes to elicit ideas on what kinds of reasoning questions he might want to ask once he'd modeled a design in. However, to a great extent, he was most interested in a visual design language itself as a way of storyboarding games. Similarly, what he found most helpful about the abstract model of economies as interconnected sources and sinks was simply that it was a useful representation for thinking about the problem, regardless of whether any automated reasoning was provided. Likewise, he was interested in our proposal for an abstract rock-paper-scissors model of unit combat because it provided him with mental tools for thinking about the unit combat design problem. Thus, interestingly, this designer was very interested in our formal representations, but primarily not for the backend reasoning they support, but for the front-end representational ability they provide, that helps in thinking about the design. He perceived the backend reasoning as being useful for tuning a design later in the design process.

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

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

5.3 Case study 3: An evolution-based game

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

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

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

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

A stumbling block preventing our prototype tools from being more useful to this team at their early stage of the design process was the fact that we served as the tools' interface, since a user-friendly interface hadn't been built yet. With the brainstorm-heavy design process, they would have preferred to have a tool they could take home and interact with on their own to really get an idea of how it could help them explore a design space, or what else they might want it to do. The opportunistic aspect of this early stage of design necessitates a tighter interaction loop than is possible with us as intermediaries, though the designers were enthusiastic about the potential for the reasoning system to quickly answer questions that came up during brainstorming.

6. CONCLUSIONS

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

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

In addition, that designer, perhaps not coincidentally, had a much more interface-in view of what our tool should do. He was most interested in the possibility of a storyboarding tool for game designers to be able to use to quickly sketch and visualize designs, with a visual design language and sets of built-in vocabulary for common design domains. This leads to an interesting proposal for what the game-design equivalent of a CAD tool's 3d modeling is: not sketching of the game's *appearance* in a superficial sense, but sketching of the game's interconnected processes and elements. Methodologically, this designer also disagreed with our backend-first approach: He had some interest in automated reasoning, but mostly thought of it as a possible next step to consider after we first built a visual modeling tool, which might present opportunities to hang automated reasoning off of some of its widgets.

For backend reasoning, we found it useful to frame most suggested queries in terms of simulation. Our initial attempts to follow the logical-reasoning literature's conceptual splits into simulation, planning, abduction, and so on, mostly led to confusion about what the tool could do. For example, planning, i.e. finding a sequence of player actions that would cause a particular outcome, is conceptually a form of directed simulation: find a simulation run that has a particular outcome, and then see what happened in it. These sorts of metaphors led to a good deal of interest in how the system could do reasoning more complicated than standard simulation, such as "backwards" reasoning to find what the largest or smallest value of some parameter would have to be to result in a particular outcome. An interesting technique that developed in several of the prototypes was using different questions to isolate different causes of outcomes. For example, sticking with a fixed player model and asking if a particular situation is possible gets at to the role world conditions play in possible outcomes; fixing world conditions and asking if a player can achieve an outcome gets at whether a particular kind of gameplay (for example, a player exploiting a design flaw) could cause an outcome.

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

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

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

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

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

Finally, the direction proposed by domain-oriented design environments and especially metadesign [10] mapped well to some of the design issues we encountered. Game-design vocabulary is a mixture of existing terms inherited from previous games (e.g. RTS-design vocabulary) and novel ideas. Thus designers may want the ability to design higher-level abstractions than the state and state-evolution rules at which our tool (and actual game implementations) currently works, and to import existing representa-

tions where they exist. For example, our second case study would have found it useful to have his thinking prompted by a toolbox of off-the-shelf RTS design vocabulary.

7. FUTURE WORK

The immediate future work is to build more fully functional prototypes implementing the requirements collected here. The backend work involves integrating probabilistic reasoning and logical induction into the framework we currently have, plus work on computational tractability to allow the system to be used frequently and on large problems. There are, equally importantly, many interface questions. How should a designer represent design goals, partial designs, and so on? How should they query the system? Should the system have built-in common design elements; and how should designers specify free-form mechanics? What's the balance between a visual design language and programming to add arbitrarily complex new mechanics to the system?

A first step on the interface could be fixing a design and backend, and building a user-friendly interface solely for the query facility, to allow designers in the brainstorming phase to interact with the system in a tighter loop. Further on, designers will need ways of inputting, modifying, and building representation language for their designs, which involves potential work on anything from domain-specific languages for specifying mechanics to visual design languages for interactive design sketching.

Apart from the specific mechanics-reasoning tool we're building, there are many avenues for future work in building other design-assistant tools. A tool along the lines suggested by our second case study, providing a sort of storyboarding-for-game-mechanics environment, would likely be useful to a number of designers. Tools focused on assisting novice game designers also have a large potential audience, and likely have different requirements.

8. ACKNOWLEDGMENTS

Many thanks to the game designers who generously gave their time to meet with us, including Josiah Pisciotta from Chronic Logic and Chaim Gingold. Thanks also to Intel for funding.

9. REFERENCES

[1] Adams, E. and Rollings, A. 2007. *Fundamentals of Game Design*. Prentice Hall.

[2] Church, D. 1999. Formal abstract design tools. *Game Developer* (August 1999).

[3] Coons, S.A. 1967. Surfaces for computer-aided design of space forms. Technical Report TR-41, Massachusetts Institute of Technology.

[4] Crawford, C. 2003. *Chris Crawford on Game Design.* New Riders.

[5] Fischer, G. 1994. Domain-oriented design environments. *Automated Software Engineering* 1(2)*:* 177-203.

[6] Fischer, G. 1998. Seeding, evolutionary growth and reseeding: Constructing, capturing, and evolving knowledge in domain-oriented design environments. *Automated Software Engineering* 5(4): 447-464.

[7] Fischer, G., McCall, R., and Morch, A. 1989. Design environments for constructive and argumentative design. *Proc. Human Factors in Computing Systems (CHI)*, 269-275.

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

[9] Gero, J.S. 1986. An overview of knowledge engineering and its relevance to CAAD. *Proc. CAAD Futures 1985*, 107-119.

[10] Giaccardi, E. and Fischer, G. 2008. Creativity and evolution: A metadesign perspective. *Digital Creativity* 19(1): 19-32.

[11] Hewitt, C. 1985. The challenge of open systems. *Byte* 10(4): 223-242.

[12] Hunicke, R., LeBlanc, M., and Zubek, R. 2004. MDA: A formal approach to game design and game research. *Working Notes of the Challenges in Game AI Workshop at AAAI 2004.*

[13] Lawson, B.R. 2002. CAD and creativity: Does the computer really help? *Leonardo* 35(3): 327-331.

[14] Lawson, B.R. 2005. Oracles, draughtsmen, and agents: The nature of knowledge and creativity in design and the role of IT. *Automation in Construction* 14(3): 383-391.

[15] Lawson, B.R. and Loke, S.M. 1997. Computers, words and pictures. *Design Studies* 18(2): 171-183.

[16] Nelson, M.J. and Mateas, M. 2008. An interactive game-design assistant. *Proc. Intelligent User Interfaces (IUI)*, 90-98.

[17] Nelson, M.J. and Mateas, M. 2008. Recombinable game mechanics for automated design support. *Proc. Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 84-89.

[18] Palmiter, S., Lynch, G., Lewis, S., and Stempski, M. 1994. Breaking away from the conventional usability lab. *Behaviour & Information Technology* 13(1-2): 128-131.

[19] Price, G.R. 1956. How to speed up invention. *Fortune magazine* (November 1956), 150-228.

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

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

[22] Ross, D.T. 1956. Gestalt programming: A new concept in automatic programming. *Proc. Western Joint Computer Conf.*, 5-10. Summarized with commentary in [23].

[23] Ross, D.T. 1986. A personal view of the personal work station: Some firsts in the fifties. *Proc. History of Personal Workstations*, 19-48.

[24] Salen, K. and Zimmerman, E. 2004. *Rules of Play*. MIT Press.

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

[26] Schön, D.A. 1992. Designing as reflective conversation with the materials of a design situation. *Research in Engineering Design* 3: 131-147.

[27] Sutherland, I.E. 1963. *Sketchpad: A man-machine graphical communication system*. PhD thesis, Massachusetts Institute of Technology.

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

[29] Wixon, D., Holtzblatt, K, and Knox, S. 1990. Contextual design: An emergent view of system design. *Proc. Human Factors in Computing Systems (CHI)*, 329-336.