Generating and Adapting Game Mechanics

Alexander Zook and Mark O. Riedl
School of Interactive Computing, College of Computing
Georgia Institute of Technology
Atlanta, Georgia, USA
{a.zook, riedl}@gatech.edu

ABSTRACT

Game designs often center on the game mechanics—rules governing the logical evolution of the game. We seek to develop an intelligent system that generates computer games and assists humans in designing games. As first steps towards this goal we present a composable and cross-domain representation for game mechanics that draws from AI planning action representations. We use a constraint solver to generate mechanics subject to design requirements on the form of those mechanics—what they do in the game. A planner takes a set of generated mechanics and tests whether those mechanics meet playability requirements—controlling how mechanics function in a game to affect player behavior. We demonstrate our system by modeling and generating mechanics in a role-playing game, platformer game, and combined role-playing-platformer game.

Categories and Subject Descriptors

Applied Computing [**Computers in other domains**]: Personal computers and PC applications—*Computer games*

Keywords

Procedural content generation, game mechanics, game design

1. INTRODUCTION

Game designers often create a game by iteratively building up a set of game mechanics—rules governing the logical evolution of the game [9, 18]. Some games depend on a small system of tightly tuned mechanics—e.g. platformer movement or simulation game rules—while others employ a large variety of counterbalancing mechanics—e.g. cards in trading card games or creatures in monster-training games. Sicart [21] generalizes these ideas to define mechanics as functions called by game agents. Choosing the right mechanics for a game involves deciding what players should be allowed to do (and when) and working out how the game mechanics should function to achieve these goals for player actions.

In this paper we explore synthesis of game mechanics from low-level game engine primitives related to checking and updating game variables. Automated reasoning on game mechanics requires a low-level and cross-domain mechanic representation. By automatically designing game mechanics, an intelligent system can create games unique to each player, generate novel solutions to design problems humans may have not been able to conceive, create games across a variety of domains, or recombine existing mechanics into new game genres.

Fully autonomous mechanic design requires a solution to the *mechanic generation* problem: *de novo* synthesis of game agent actions given knowledge of a game domain. Supporting iterations on existing designs requires a solution to the *mechanic adaptation* problem: adjusting mechanics to meet new designer goals for given game mechanics and content. We address the mechanic generation and adaptation problems through a formalism to define composable game mechanics and accompanying techniques to generate mechanics subject to playability and design requirements. *Playability requirements* ensure mechanics function properly to allow players to achieve certain goals in given game content (e.g. reaching the end of the level or being able to explore many places). *Design requirements* ensure mechanics have a designer-desired form (e.g. having few exceptions or not overlapping heavily with how another mechanic works). We focus on generating or adapting avatar-centric mechanics—actions taken by agents in the game world—rather than the full set of game rules.

Generated mechanics take the form of planning operators in a representation specialized to game mechanics. Our mechanic design technique is a "generate-and-test" process: (1) generating mechanics that meet design requirements on form and (2) testing mechanics to ensure they meet playability requirements. A constraint solver generates possible mechanics in a given game domain according to hard (required) or soft (optimized) design requirements. An AI planner tests playability by using generated mechanics to prove that designer-specified requirements for good gameplay can be achieved with the mechanics. For example, players must be able to reach the end of a level or win a battle without dying. Unlike AI planners that solve game levels using a fixed set of operators (mechanics), mechanic generation creates the operators. A planning operator representation supports our goals for a composable and domain-independent mechanic representation.

Together, the constraint solver and planner can generate or adapt game mechanics in a (relatively) domain-agnostic fashion while ensuring the mechanics achieve desired play experiences. We demonstrate our mechanic generation system in three game domains: platformer game movement mechanics, role-playing game (RPG) spell systems, and a domain combining the platformer and RPG domains.

2. RELATED WORK

Game generation systems take a human-specified set of possible domain content and synthesize possible game mechanics (and game content). Game generation in arcade games (similar to the games we model) has focused on assigning collision and movement logic to game entities using predefined tables enumerating possible choices. Researchers have used evolutionary search [28], constraint satisfaction [24], and rule-based systems [30] to generate games meeting soft optimization criteria and/or hard constraints. Rather than use top-down enumeration of game mechanics, Cook et al. Mechanic Miner [3] used a bottom-up approach—program reflection—to manipulate game mechanics by changing the values used in program functions. We use a top-down mechanic representation to support *de novo* synthesis of mechanics from game engine primitives subject to several (relatively) domain-independent evaluation criteria.

Complementing automated approaches, Nelson and Mateas argued for recombinable mechanics to support human designers [15]. Researchers have supported human-created mechanic analysis with playability checking, using simulations in Petri net models [4], model-checking and proof in extensions of the event calculus [23,25], and simulations or model-checking in other action languages [16]. Our system draws on related logical models (planning) and can also generate games using the mechanics being modeled.

Mechanic generation and game description languages (GDLs) share a concern for composable mechanic representations. The Stanford Game Description Language [12] models turn-based, competitive games in a declarative language and has extensions for randomization and incomplete information [27]. A variety of research efforts have modeled specific classes of games using similar (context-free or graph) grammar constructs, including arcade video games [19], card games [8], strategy games [13], action-adventure games [6], and puzzle games [11]. Grammars are effective for embedding design knowledge into a generating system, but are not readily combined across genres. We avoid this limitation through a cross-domain mechanic representation that is also amenable to automated generation.

Our model of game mechanic structure draws from work on domain representations used in AI planning. Modern plan representations were developed to scale traditional AI techniques to complex domains by providing additional problem structure knowledge to AI search processes. Planning representations can often be converted (e.g. to SAT problems) to improve the performance of other approaches through additional representational factoring [17]. STRIPS [7] was one of the earliest planning representation languages, modeling actions in terms of logical predicates. Operators are a set of preconditions that must hold before the action can be executed and a set of effects that add or delete predicates from the state of the world. Planning is a process of finding a sequence of operations that transform the world from an initial state into a state in which the goal situation holds. The Planning Domain Description Language (PDDL) [14] is an ongoing project to extend planning representations to address more complex tasks while building a shared language for research competitions. PDDL extended STRIPS-like representations with non-equality constraints, numeric fluents to model continuous domains, operators with duration, and timed initial literals that modify the world state at fixed times regardless of agent actions. By modeling mechanics in a similar manner to planning domains we can leverage existing work on planning technologies to check game playability.

3. MECHANIC DESIGN FORMALIZATION

In this section we define the mechanic generation and adaptation problems, provide a model for cross-domain composable game mechanics, and present methods to automatically generate and test game mechanics for given game content. *Mechanic generation* is the problem of constructing a (set of) game mechanic(s) such that they meet playability requirements to create a desired range of player experiences (allowing and forbidding action sequences) while meeting design requirements on mechanic structure. *Playability requirements* ensure a game is playable to a given goal, potentially subject to limitations on the states entered or actions taken to achieve the goal. *Design requirements* ensure mechanics adhere to designer requirements for how actions work in a game. Both playability and design requirements may be domain-independent or domain-dependent. Design requirements shape mechanics to the form a designer desires while playability requirements ensure those mechanics have desired functions in game content.

Mechanic generation uses a game domain definition to know what may be changed by mechanics. A *game domain* defines the entities that make up a game, their parameters, and how game states change. A game domain consists of a *state model*—specifying domain entities, their parameters, and allowed ranges of values—and a *transition model*—specifying how states change into one another. In our formulation, the transition model is the set of game mechanics. The focus of this paper is *avatar-centric* mechanics—transitions initiated by the player (or other in-game agents) in the process of controlling an avatar. A solution to the mechanic generation problem is a transition model that meets design and playability requirements, given a state model and a set of relevant game instances.[1] A *game instance* is an initial state and a particular setting that corresponds with the initial state; e.g., a level in a platformer or a single battle in a role-playing game. Initial game state is part of a game instance. When a game instance for a state model is combined with a transition model (including avatar-centric mechanics) the result is a playable game experience.

As a running example to illustrate our definitions, consider a simple role-playing game (RPG) battle game domain. RPG battles involve two opposing parties taking turns to attack one another using various spells (mechanics) until one party is slain; the *Dungeons and Dragons* tabletop RPGs are a paradigmatic example. The RPG state model has a player character and an enemy, each with health and mana resources. One game instance has the player starting with 3

[1]By this definition most PCG generates game *instances* given a state model. Some approaches enforce playability requirements given a transition model.

health and 5 mana (a spell-casting resource) while the enemy has 2 health and 2 mana. Many variant instances may be considered simultaneously when mechanics are generated. A playability requirement can ensure that, over all given instances, the player can kill the enemy (reduce enemy health to 0 or less) without being killed. A design requirement can specify that all spells have a cost—e.g. requiring that every spell cost the avatar that uses it some resource (health or mana). Mechanic generation asks: given these requirements what spells should be in the game? Mechanic adaptation is similar to generation but starts from a pre-specified set of mechanics and modifies those mechanics and/or augments those mechanics with new mechanics.

A system that solves the mechanic generation problem requires a state and transition model representation, a process to search for transition models that meet design criteria within the state model, and a process to test that transition models meet playability criteria across a provided set of game instances (potentially all valid game instances in that space). Below we present our state and transition models to generate avatar-centric mechanics as planning operators. Planning operators are a natural representation for game mechanics as operators were designed to represent domains involving sequential choices while readily allowing composition of operator preconditions and effects. Our operator representation can directly reference variables in the game engine, allowing us to operate at a very low level of primitives. We use a constraint solver to search for a transition model (mechanics) that meets design criteria. Constraint solvers are a valuable generic approach to search combinatorial spaces that readily encode hard and soft requirements on solutions [22]. We use a planner to validate transition models against playability criteria. Planners are effective for proving the presence or absence of play traces (sequences of mechanic choices and state updates) within a game domain.

3.1 State Model

Our representation for game domains uses a subset of the ideas used in PDDL with extensions specific to games. Currently we focus on turn-based domains with deterministic actions to simplify our early exploratory work. The state model defines a domain of game entities in terms of their allowed states as these are core to modeling avatar-centric mechanics (state transitions).

The state model is a set of terms defining entities, parameters, and allowed parameter value ranges for entity parameters in the game world (*AbsRange*) or mechanic changes to those values (*RelRange*). Terms have the following forms:

$$Entity(e) \qquad Parameter(p) \qquad Has(e,p)$$
$$AbsRange(p,e,r) \qquad RelRange(p,e,r)$$

where e is a symbol representing an entity, p is a parameter of an entity, and r is a range of values, which may be discrete or continuous. We currently approximate continuous ranges with integer-valued ranges for simplicity. A game engine can use this formalism by exposing an API with 'get' and 'set' methods for game engine variables corresponding to these logical terms. Referring back to our example RPG spell system we can define the player with the predicates in Table 3.1 (*RelRange* relates to the transition model described in section 3.2).

Game instances give concrete settings for state model entities and parameters. We use fluents to represent these values in our planning model, allowing states to change according to the transition model. In our RPG example, we can set player health to initially be 3 using $Initial(health(player), 3)$ where *Initial* sets entity parameter values that hold at the beginning of the game.

Table 1: Partial RPG domain definition.

Entity(player)	
Parameter(health)	*Parameter(mana)*
Has(player, health)	*Has(player, mana)*
AbsRange(player, health, [0,3])	*AbsRange(player, mana, [1,5])*

3.2 Mechanic Model

The transition model is a set of mechanics that allows forward simulation (making results playable as simple text-based games) and playability checks as planning. Consider modeling an RPG spell that causes damage over a period of time. Such a spell needs to specify several things: conditions on when the spell may apply (e.g. not affecting dead characters), how much damage is done, and at what time(s) the damage is done. To address examples like this we have drawn from PDDL's action schemas to define an avatar-centric mechanic as a tuple: $\langle i, P, E\rangle$ where i is a unique identifier for a mechanic, P is a set of the preconditions needed for mechanics to occur, and E is a set of effects of performing the mechanic.

Our preconditions and effects extend traditional PDDL action schemas with time-indexing and coordinate frames of reference. Time-indexing allows preconditions to reference state at times other than the present and allows effects to reference states other than the next game state. Games often incorporate delayed effects or checks on historical state, motivating our time-indexing extension. Coordinate frames distinguish between traditional world-state terms and "perceived" avatar-relative versions of world terms. Absolute frames of reference model requirements on the state of the world. Relative frames of reference capture the intuitive notion that many avatar-centric game mechanics have preconditions and effects relative to an avatar, rather than absolute world state (e.g. adjacency as relative position).

Our planning model implements semantics for a subset of PDDL with extensions appropriate to our definition. *AbsRange* is used to specify valid absolute frame of reference values while *RelRange* is used for relative frames of reference. Preconditions test game state; we allow tests for equality, inequality, and lesser-than and greater-than relations. All preconditions and effects are tuples of the form $\langle frame, time, condition\rangle$; where $frame$ indicates a coordinate frame of reference, *time* specifies a time-index, and *condition* specifies a game state value to check for (or update). In our formalism, a condition takes the form $F(parameter(entity), value)$ where F is a logical function that either tests two values and returns a boolean value (for preconditions) or updates an entity parameter value (for effects). Testing for the avatar currently being alive would be $\langle absolute, 0, GreaterThan(health(player), 0)\rangle$.

Effects update game state. For absolute frames of reference updates set state to a particular value (constrained within *AbsRange*); for relative frames of reference updates change state values by a given amount (constrained within *RelRange*). A spell that checks for the enemy being alive and reduces enemy health by 1 on the two next turns is:

$\langle damageOverTime,$
$\{\langle\ absolute, 0, GreaterThan(health(enemy), 0)\rangle\},$
$\{\langle relative, 1, Update(health(enemy), -1)\rangle,$
$\langle relative, 2, Update(health(enemy), -1)\rangle\}\rangle$

Mechanic *recombination* occurs when one mechanic references another mechanic having occurred. Fighting game or rhythm game combo systems exemplify avatar-centric recombination. Mechanic recombination naturally encodes event-relevant mechanics, rather than being limited to mechanics that reference state. For mechanic recombination we allow preconditions and effects to reference the *event* of a mechanic occurring with $Performed(i)$. Semantically, a mechanic as a precondition requires that mechanic to have (or not have) occurred at a time index. For example, a double-jump may require a player to have jumped at the previous time-step:

$$\langle absolute, -1, Equal(performed(player), jump)\rangle$$

When $Performed(i)$ appears as an effect the preconditions and effects of that mechanic are applied. The mechanic using $Performed(i)$ as an effect indicates the time to apply the performed mechanic. Note that frames of reference are not relevant for mechanic indexes (these are provided by the indexed mechanics themselves) and are ignored.

As in PDDL, we assume inertial state and circumscription: any entity parameter not affected by a mechanic continues to hold its previous value. $Performed(i)$, however, is treated as an *event* and not subject to inertial state.

4. MECHANIC GENERATION

Mechanic generation creates a set of mechanics within a game domain subject to playability and design requirements. We use a constraint solver to search for a set of mechanics constrained to meet the given design requirements. Design requirements help avoid low-quality mechanic solutions. Hard design requirements (as used by Smith and Mateas [22]) enforce conditions on the form of mechanics or relations among a set of mechanics—e.g. not allowing a mechanic to have both equality and non-equality preconditions for the same game state or requiring no two mechanics to have identical preconditions and effects. Soft design requirements (as reviewed in [29]) give optimization criteria for what makes (sets of) mechanics better or worse—e.g. aiming to minimize the number of preconditions and effects used by a mechanic in favor of simplicity. Playability is evaluated using a planner (described in the next section) to prove a player can meet playability requirements on given test game instances. We used Answer Set Programming (ASP) [1]—a form of declarative programming—to implement the constraint solver.

Mechanic generation creates mechanics by choosing preconditions and effects for each mechanic while ensuring the mechanics conform to design requirements. Some design requirements apply across types of games (not requiring a state hold and not hold at the same time) while others are more domain-specific (spells should be "balanced" in terms of resource costs to execute vs effects on avatars). Given a set of operators, a planner proves whether a plan exists for given game content subject to playability requirements. The process of generating mechanics using a constraint solver and testing those mechanics with a planner repeats until all hard requirements are met and all soft requirements are optimized. While this is an expensive process we have started with small game domains to explore the relevant research problems. Note that many game mechanics and game systems consist of relatively small abstract systems (e.g. RPG spell systems, platformer mechanics, etc.).

4.1 Playability Checking

We implemented a simple planner that proves that playability requirements can be met in game instances with a given set of mechanics. The planner uses playability requirements as goal situations to prove whether a plan exists that can meet playability requirements. For convenience we used ASP as our implementation language for the planner.

Playability requirements come in three forms: (1) goals, (2) maintenance goals, and (3) engine constraints. Goals give a game agent target situations to seek; the planner must prove the presence of a plan that meets the goal. Maintenance goals give situations that begin true and must hold throughout the plan (e.g. being alive); the planner must prove a plan achieving the goals always upholds maintenance goals. Maintenance goals are useful for specifying failure criteria in a game as the negation of a failure state must always hold. Engine constraints enforce semantics mapping to non-avatar rules in a game engine (e.g. preventing two entities from occupying the same space); the planner must follow these constraints when making plans.

In our RPG battle example, the player goal is to kill all enemies while maintaining the state of being alive (not being killed) and we have an engine constraint that the player cannot drop below 0 mana. By building our model off planning domain representations we gain a simple, factored logical model of the game world that affords game mechanic combination and synthesis while also yielding playable games.

4.2 Mechanic Adaptation

Instead of generating mechanics from scratch, *mechanic adaptation* starts with a set of mechanics and produces a minimally changed set of mechanics. Mechanic adaptation uses mechanic generation for iterative design. In iterative design a set of mechanics are tested and adjusted to meet new insights about the game—adaptation requirements. Mechanic adaptation is given the same inputs as mechanic generation along with an initial set of mechanics and new adaptation requirements. *Adaptation requirements* specify additional playability or design requirements for mechanic generation. New playability requirements may indicate additional goal states for the player to pursue or identify unwanted states. New design requirements may control the amount of change to make to a set of mechanics. The definition of 'minimal change' varies by game domain and must be specified to adapt mechanics.

Mechanic adaptation takes the same input game state and transition models as mechanic generation augmented with a pre-existing set of game mechanics. Adaptation adds or removes preconditions and effects from existing mechanics and may also generate new mechanics. Changes to mechanics must meet designer-specified criteria for minimality while adhering to all adaptation requirements. We adapt mechanics by having the constraint solver perform the standard generation process but seeded with the additional mechanics. The previous set of design requirements are given along with new adaptation requirements and a definition of minimality (e.g. minimizing the total number of changes made). Mechanic adaptation performs the same loop of generating and testing possible mechanics as in mechanic generation.

5. EXAMPLES

Our game domain formalism supports a variety of avatar-centric mechanic systems. In this section we illustrate how to represent a simple role-playing game (RPG), a simple platformer, and a game that merges these two systems. RPGs require a balanced and diverse set of character spells. Platformers are games where a character navigates physical obstacles in a virtual space, exemplified by the *Super Mario Bros.* games. Platformers require a finely tuned and widely reused small set of spatial navigation mechanics. We generate spells in the RPG and movement mechanics in the platformer. By concatenating these two domains we illustrate how our model affords cross-domain mechanic generation.

5.1 Role-Playing Game

RPG combat mechanics can be specified in terms of a set of entity attributes and resources (here health and mana for the player and a set of enemies). Our earlier RPG spell example defines this basic domain. We have playability requirements for: a player goal situation of having all enemies dead, a player maintenance goal of not being dead; and an engine constraint preventing negative mana. Together, these playability requirements encode the basic notion of an RPG battle as killing an opponent without being killed while having bounded resources. We have two domain-independent design requirements: a hard requirement to prevent mechanics from having preconditions that force a predicate to equal more than one value and a soft requirement to minimize the number of preconditions and effects of mechanics to produce the simplest set of mechanics. Many domains have a notion of actions having costs; we used a third, domain-specific version of costs by requiring all actions incur a mana or health cost.

Our system was able to generate a variety of RPG spells using the game domain, a game instance with two enemies, and the playability and design requirements above. Playtraces are plans: a series of player actions (spells used) to damage each of the enemies while costing the player health or mana. One example spell was given above, others typically perform simple effects such as inflicting damage at a single time point or affecting multiple targets. Our system generated the following mechanic to damage all enemies:

$\langle damageAll,\ \{\},$
$\quad \{\langle relative, 1, Update(health(enemy1), -1)\rangle,$
$\quad\ \langle relative, 1, Update(health(enemy2), -1)\rangle,$
$\quad\ \langle relative, 1, Update(mana(player), -2)\rangle\}\rangle$

where there are no preconditions and the effects damage both enemies while costing the player mana. Note that we have given human-readable names to the mechanics; internally i (the name) is an integer. Also note that our examples were chosen to illustrate the most semantically sensible mechanics generated; by definition all mechanics achieve playability and design requirements.

5.2 Platformer

Two-dimensional platformers can be described in terms of a set of entities (here the player, blocks, and enemies) each assigned spatial coordinates corresponding to two spatial dimensions (Table 2). The initial state of the player for our example (see Figure 1) is $Initial(xPos(player), 1)$, $Initial(yPos(player), 2)$.

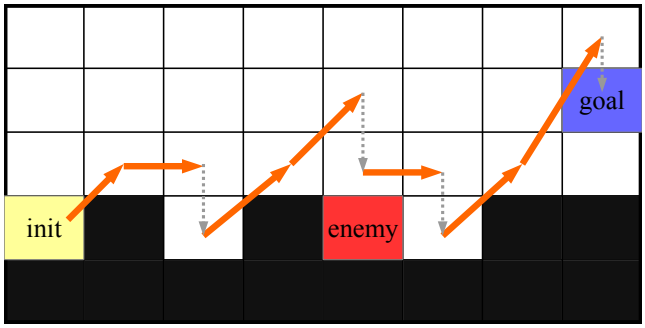

Figure 1: Platformer level showing a playtrace using a generated mechanic set. Arrows indicate generated mechanics, dotted arrows indicate gravity.

Table 2: Partial platformer domain

Entity(player)	
Parameter(xPos)	*Parameter(yPos)*
Has(player, xPos)	*Has(player, yPos)*
AbsRange(player, xPos, [1,8])	*AbsRange(player, yPos, [1,6])*

The platformer has playability requirements for: a player goal situation of reaching the end, a player maintenance goal of not overlapping with an enemy; and an engine constraint preventing the overlap of any entity and a block. Another engine constraint enforces gravity by requiring all entities to move down one unit each turn if that space is not occupied by a block. We reused two design requirements from the RPG example: preventing exclusive pre-conditions and minimizing the number of mechanic preconditions and effects. A third soft requirement optimizes for as few mechanics as possible (to create a "tighter" game system) and a fourth soft requirement minimizes the number of different entities referenced by mechanics (favoring motion of a single avatar).

Figure 1 illustrates a simple platformer level and shows one trace found by the planner that moves the player avatar to the goal position. The planner generated mechanics for moving forward, jumping, and double-jumping (indicated by arrows). Dotted arrows indicate the effects of gravity. The example shows a variety of movement mechanics we have generated, including two forms of jumping:

$\langle jump,$
$\quad \{\langle relative, 1, Equal(yPos(e), yPos(block) + 1)\rangle,$
$\quad\ \langle relative, 1, Equal(xPos(e), xPos(block))\rangle\},$
$\quad \{\langle relative, 1, Update(xPos(e), 1)\rangle,$
$\quad\ \langle relative, 1, Update(yPos(e), 1)\rangle\}\rangle$
$\langle double\ Jump,$
$\quad \{\langle relative, 1, Equal(yPos(e), yPos(block) + 1)\rangle,$
$\quad\ \langle relative, 1, Equal(xPos(e), xPos(block))\rangle,$
$\quad\ \langle absolute, -1, Equal(Performed(i), jump)\rangle\},$
$\quad \{\langle relative, 1, Update(xPos(e), 1)\rangle,$
$\quad\ \langle relative, 1, Update(yPos(e), 2)\rangle\}\rangle$

$jump$ tests for the presence of a block to jump off of and, if so, moves the avatar diagonally up. $doubleJump$ does the same check while also requiring a jump to have occurred immediately before; the jump effect is slightly larger.

In initial platformer mechanic generation runs $jump$ and

doubleJump lacked preconditions as this minimized the complexity of mechanics. To address this problem we generalized the notion of costs from the RPG domain to a cost–benefit system based on ideas discussed by Schreiber [20]. Specifically, we associated each effect with benefit equal to the update effect absolute magnitude. Preconditions constraint mechanics and each have a cost of 1. A hard design requirement enforces 'balanced' mechanics by requiring the net costs and benefits of a mechanic to be equal. Adding cost–benefit accounting led to the mechanics we report here.

Two more unique mechanics appeared when using our system on a slightly simplified version of the above domain. The simplification removed blocks at even height with the player to create a plain. Our system generated a 'lift' mechanic to move the enemy and a 'ride' mechanic in two different solutions. $lift$ raises the enemy behind the player and was used to allow the player to move the enemy behind them while advancing to the goal:

$\langle lift,$
$\{\langle relative, 1, Equal(yPos(e), yPos(enemy))\rangle,$
$\langle relative, 1, Equal(xPos(e), xPos(enemy) - 1)\rangle,$
$\langle relative, 1, Equal(yPos(e), yPos(block) + 1)\rangle,$
$\langle relative, 1, Equal(xPos(e), xPos(block))\rangle\},$
$\{\langle relative, 1, Update(xPos(enemy), -1)\rangle,$
$\langle relative, 1, Update(yPos(enemy), 2)\rangle,$
$\langle relative, 1, Update(xPos(e), 1)\rangle\}\rangle$

ride was a mechanic used to slide the player and enemy forward both by one unit and was used to have the player jump atop an enemy and 'ride' the enemy to the goal (shortening the jump distance needed):

$\langle ride,$
$\{\langle relative, 1, Equal(yPos(e), yPos(enemy) + 1)\rangle,$
$\langle relative, 1, Equal(xPos(e), xPos(enemy))\rangle\},$
$\{\langle relative, 1, Update(xPos(e), 1)\rangle,$
$\langle relative, 1, Update(xPos(enemy), 1)\rangle\}\rangle$

We also test mechanic adaptation in our platformer domain to adapt mechanics generated without gravity to work in the same domain with gravity. First we generated a set of movement mechanics in our platformer domain, resulting in three mechanics: a long horizontal jump (*longJump*: 2 forward, 1 up), a short vertical jump (*highJump*: 1 forward, 2 up), and a dash forward (2 forward). Adding gravity requires the agent to increase the amount of vertical movement when gravity is added. We included gravity as an engine constraint and adapted the above mechanic set by reusing the same platformer domain and requirements. The resulting modifications made two changes: (1) the dash added vertical movement to now move 2 forward and 1 up and (2) the long jump added an initial lift phase moving 2 up, but at a time one step earlier than the rest of the mechanic. These results illustrate the flexibility to reuse our generation system for adaptation when baseline design considerations change. Adaptation only required seeding the generation with output from a previous generation step and specifying how many mechanics to use after adaptation (in this case preventing new mechanics from being added).

5.3 Combined Game

To demonstrate the modularity of our representation we concatenated the previous two domains to create a "platformer-RPG" game. All game state definitions are unchanged: combining RPG resources and platformer location only makes entity state more complex. We retain the previous playability requirements from both domains with conjunctive (all criteria must be met) goals, maintenance goals, and engine requirements. With these simple changes we can generate mechanics appropriate to the domain such as attacking at a distance with a spell:

$\langle magicMissile,$
$\{\langle relative, 0, Equal(xPos(enemy), 2)\rangle,$
$\langle relative, 0, Equal(yPos(enemy), 0)\rangle\},$
$\{\langle relative, 0, Update(health(enemy), -1)\rangle\}\rangle$

where the preconditions check for an enemy two spaces in front of the player and the effect reduces enemy health.

6. RICHER AI DESIGN

To further develop the design tasks encountered in the previous example domains we have extended our system to generate mechanics for multilevel progressions, multiagent competition, and mapping controls to mechanics. These additions illustrate how our representation can model some more complex design tasks that directly relate to mechanics.

6.1 Multilevel Progression

Platformers (and most game genres) often gradually introduce new mechanics to players over a sequence of levels. Generalizing mechanic generation to include requirements on which mechanics are used along a progression requires two additions: planning across multiple levels and providing requirements on mechanic use. To implement multilevel progression we augmented the initial state and playability requirement definitions with a level index of the form $Initial(level, paramter(entity), value)$. Playability checks must ensure the given mechanic set can yield valid playtraces for all levels provided, treating each as a separate planning problem with the same set of mechanics.

The constraint solver can enforce types of progression across multiple levels. For example, we have required the number of mechanics used in each level to increase over a level progression. We have also required that the specific mechanics used in each level reappear in all subsequent levels. This has resulted in the sequential introduction of the $jump$ and *doubleJump* mechanics above. In general the generated mechanic sequences often involve enveloping mechanics where a weaker and stronger (larger effect) version of the same mechanic are used. These progression requirements encode a notion of training players by needing to master additional skills (c.f. Butler et al. [2]; Dormans [5]). We have also used our more atomic representation to require the progressive introduction of preconditions or effects (as in the *doubleJump* introduction of an event precondition). These requirements allow more nuanced ideas of progression than previously done by using elements of the mechanics being introduced to the player.

6.2 Multiagent Actions

RPG battles typically involve competing agents. To incorporate multiagent modeling we have augmented our planner to track actions and perceived state relative to each agent. We now indicate agent-specific goals and maintenance goals (engine constraints are currently treated as universal).

Playability checks optimize toward all agent (potentially competing) goals. To ensure plans are possible we typically require the player be able to achieve her goal situation before any opposition, but that both goal situations can be achieved within a prescribed number of plan steps. Alternatively, we have also provided goals to adversaries that are intended to improve player experience without directly negating the player's maintenance goals (e.g. trying to minimize player health, rather than kill the player). Adding multiagent modeling is computationally costly but allows broader modeling of competition (or collaboration) interactions.

6.3 Controls

Platformers depend heavily on the game controls. Our modular representation can readily map a given set of input buttons to generated mechanics. We define the input commands, add these controls as additional preconditions for mechanics, and require there is always a single unambiguous mechanic for an input. Hard design requirements state that all mechanics have at least one input and no two mechanics with the same preconditions use the same set of inputs. Soft design requirements encode simplicity by minimizing the number of inputs used in total and number of inputs used per mechanic. Additional soft design requirements encode a simple notion of 'intuitive' mappings by maximizing the use of the same buttons for mechanics with overlapping effects on the same entity-parameter-value settings.

Our control mapping scheme was able to generate (relatively) semantically sensible platformer controls. We provided our control mapping system with $jump$, $doubleJump$, $lift$, and $ride$ as above and a set of 6 input buttons for a 4-directional pad with two action buttons (A and B). Control assignments used either 3 or 4 input buttons, trading off minimizing the total number of buttons used against minimizing the number of buttons used per mechanic (Table 3). Different assignments used different specific buttons for the same results. Note that no two buttons had identical preconditions, meaning a (non-optimal) assignment could have used a single button for all actions.

While we have not yet addressed more complex control mapping problems than these (e.g. mapping to real-world physics or durative button presses [26]) the ability to add these capabilities readily to our system shows the promise of our formalism. Future research on controls can work to capture more sophisticated control models and develop techniques to better match controls to the semantics of the game system created by the mechanics. Directly encoding controls for mechanics also opens the possibility to model agents that must use a control set (with the associated design complications), rather than directly triggering mechanics. Capturing the influence of controls on agent actions can then address the influence of control sets on player interaction, broadening the scope of PCG for games.

Table 3: Control assignment examples

	3 button	4 button
jump	↑	↑
doubleJump	A + ↑	→
ride	A	A
lift	B	B

7. FUTURE WORK

We plan to develop a variety of further extensions to our system: non-avatar mechanics, cost–benefit balancing, probabilistic mechanics, and turn structures and higher-level game systems. While we started with avatar-centric mechanics, the multiagent extensions we developed can model mechanics that do not use concrete in-game entities. For example, a 'gravity' agent can act to move all in-game agents down every turn.

Cost and benefit balancing opens many questions for further research. How can the amount of constraint imposed by a precondition be encoded? How can a system automatically detect when an effect is harmful or beneficial? How should cost and benefit magnitudes be scaled? These capabilities may depend on additional design requirement knowledge as input or additional inference about the affects mechanics have on playtrace solutions (e.g. encoding costly effects as those that move the game further from the goal state).

Probabilistic action effects are common in many games and designers often reason on expected or "average-case" scenarios [20]. Planning researchers have developed models for probabilistic outcomes that we believe can be applied to automatically synthesize and reason on playtraces in a probabilistic action semantics [10]. Adding probabilistic mechanics can open additional kinds of soft playability requirements around expected player experiences and may be valuable for expressing cost–benefit balancing schemes.

Many game domains have more complex temporal structure: e.g. continuous time, discrete turns with action costs or other complex turn structure. An extension to our representation could allow explicit reasoning on how time indexes are tracked and modified. In AI planning successor-state axioms are used to reason on how states update and how many actions are used. Exposing the game turn structure as successor state axioms used by the planner can enable this level of control for automated generation.

A core challenge in developing our system into an AI assistant is recognizing which mechanics are valuable to designers. While our system can generate mechanics according to designer constraints the system requires a designer to encode the notions of 'interesting' or 'non-trivial' into design requirements. An important open problem is developing metrics that allow the system to automatically recognize the mechanics that would be most useful to designers without as much explicit guidance. Learning from designers can help provide such metrics, either through a corpus of human examples or through explicit feedback about generated mechanics. Algorithms that appropriately generalize from human examples may enable the system to introduce new primitives into the game engine. Many primitives humans add to games are based on complexifying core gameplay: e.g. adding a new resource in a city-building game to balance economic costs and benefits.

8. CONCLUSIONS

In this paper we formalized the mechanic design problem, presented a cross-domain representation for avatar-centric mechanics, and illustrated how to generate and combine mechanics using a constraint solver and planner. Our cross-domain representation can model a subset of many common game mechanics and integrate reasoning on mechanics into

game level progressions and controls. Our system generates playable and sensible games from appropriate requirements.

We believe generating a game from the mechanics up allows new types of AI game designers that can reason explicitly about how elements of a design integrate to achieve design goals. By using a domain–agnostic representation our system can readily work in a variety of game domains, focusing on the higher–level problems of designing mechanics rather than genre–specific concerns. Autonomous mechanic generation (given designer initial inputs) holds promise for creating AI designers that generate games starting from mechanics. Mixed-initiative mechanic adaptation can lead to alternative approaches to iterative mechanic design where an intelligent system takes more of the burden of identifying mechanic changes (or new mechanics) to improve a design. In the future, when coupled with the ability to learn design guidelines, we see automated game design as a powerful tool for people without the means to build computer games themselves; an AI designer can create games from scratch from high-level descriptions or work directly with the human to iteratively construct the game.

9. REFERENCES

[1] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[2] E. Butler, A. M. Smith, Y.-E. Liu, and Z. Popović. A mixed-initiative tool for designing level progressions in games. In *ACM Symposium on User Interface Software and Technology*, 2013.

[3] M. Cook, S. Colton, A. Raad, and J. Gow. Mechanic Miner: Reflection-driven game mechanic discovery and level design. In *EvoGAMES*, 2013.

[4] J. Dormans. Machinations: Elemental feedback structures for game design. In *GAMEON-NA*, 2009.

[5] J. Dormans. Adventures in level design: Generating missions and spaces for action adventure games. In *1st Workshop on Procedural Content Generation in Games*, 2010.

[6] J. Dormans. Generating emergent physics for action-adventure games. In *3rd Workshop on Procedural Content Generation in Games*, 2012.

[7] R. E. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1972.

[8] J. M. Font, T. Mahlmann, D. Manrique, and J. Togelius. A card game description language. In *Applications of Evolutionary Computation*, pages 254–263. Springer, 2013.

[9] T. Fullerton, C. Swain, and S. Hoffman. *Game Design Workshop: A Playcentric Approach to Creating Innovative Games*. Morgan Kaufmann, 2008.

[10] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory & Practice*. Elsevier, 2004.

[11] S. Lavelle. PuzzleScript. website, 2013.

[12] N. Love, T. Hinrichs, D. Haley, E. Schkufza, and M. Genesereth. General game playing: Game description language specification, 2008.

[13] T. Mahlmann, J. Togelius, and G. N. Yannakakis. Towards procedural strategy game generation: Evolving complementary unit types. In *Applications of Evolutionary Computation*, pages 93–102. Springer, 2011.

[14] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL - the planning domain definition language. Technical report, Yale Center for Computational Vision and Control, 1998.

[15] M. J. Nelson and M. Mateas. Recombinable game mechanics for automated design support. In *4th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2008.

[16] J. Osborn, A. Grow, and M. Mateas. Modular computational critics for games. In *9th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2013.

[17] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3rd edition, 2009.

[18] K. Salen and E. Zimmerman. *Rules of Play: Game Design Fundamentals*. MIT Press, Cambridge Mass., 2003.

[19] T. Schaul. A video game description language for model-based or interactive learning. In *IEEE Conference on Computational Intelligence in Games*, 2013.

[20] I. Schreiber. Game balance concepts. `http://gamebalanceconcepts.wordpress.com/`.

[21] M. Sicart. Defining game mechanics. *Game Studies*, 8(2), 2008.

[22] A. Smith and M. Mateas. Answer set programming for procedural content generation: A design space approach. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):187–200, 2011.

[23] A. M. Smith, E. Butler, and Z. Popović. Quantifying over play: Constraining undesirable solutions in puzzle design. In *8th International Conference on the Foundations of Digital Games*, 2013.

[24] A. M. Smith and M. Mateas. Variations Forever: Flexibly generating rulesets from a sculptable design space of mini-games. In *IEEE Conference on Computational Intelligence and Games*, 2010.

[25] A. M. Smith, M. J. Nelson, and M. Mateas. LUDOCORE: A logical game engine for modeling videogames. In *IEEE Conference on Computational Intelligence and Games*, 2010.

[26] S. Swink. *Game Feel: A Game Designer's Guide to Virtual Sensation*. Morgan Kaufmann, 2009.

[27] M. Thielscher. A general game description language for incomplete information games. In *AAAI*, volume 10, pages 994–999, 2010.

[28] J. Togelius and J. Schmidhuber. An experiment in automatic game design. In *IEEE Symposium on Computational Intelligence and Games*, 2008.

[29] J. Togelius, G. Yannakakis, K. Stanley, and C. Browne. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):172–186, 2011.

[30] M. Treanor, B. Schweizer, I. Bogost, and M. Mateas. The micro-rhetorics of Game-O-Matic. In *7th International Conference on the Foundations of Digital Games*, 2012.