Recombinable Game Mechanics for Automated Design Support[*]

Mark J. Nelson
College of Computing
Georgia Institute of Technology
mnelson@cc.gatech.edu

Michael Mateas
Computer Science Department
University of California, Santa Cruz
michaelm@soe.ucsc.edu

Abstract

Systems that provide automated game-design support—whether fully automated game generators, or tools to assist human designers—must be able to maintain a representation of a game design and add or remove game mechanics to support incrementally modifying the game. The system should then be able to reason about the design to support the designer. For example, it might point out that the set of mechanics makes the game unwinnable; or that there's only one complex possible way to beat the game; or that some room is impossible to get to. In addition, the same representation should be actually playable as a game. Existing game representations encode a fairly narrow range of games, most commonly symmetric board games; these representations are also difficult to extend or revise. We propose an architecture based on the *event calculus*, a logical representation designed for reasoning about time in an elaboration-tolerant way, meaning that designs can be changed by adding or removing sets of axioms rather than modifying brittle hard-coded representations. The resulting game design is a declarative specification in formal logic, so can be critiqued by making queries that are answered through logical inference. Since it specifies the game's simulation rules, it may be executed by logical inference as well; if symbols specifying input and graphical representation are appropriately mapped to input devices and a graphical display, the same declarative representation can be executed as a fully interactive, graphical game. We describe how to organize a library of game-design mechanics using this event-calculus framework, and describe a simple tile-based game, showing how it can be easily modified, critiqued and debugged, and played.

Introduction

Automated and computer-assisted game design are emerging areas of research. An automated game-design system designs new games by maintaining an internal representation of a game, which it reasons about and modifies (Pell 1992; Orwant 2000; Nelson and Mateas 2007; Hom and Marks 2007). A game-design assistant allows a human to drive the game-design process, but automates common or tedious tasks and gives feedback and suggestions on the design in progress (Nelson and Mateas 2008). Both systems require a significant common back end: They need a way of representing game designs declaratively so that the system can reason about aspects of the design in progress; a way of modifying the designs; and a way of producing actual playable games from the declarative representation.

Existing systems restrict themselves to representations designed to capture specific, limited classes of games, such as two-player symmetric board games (Pell 1992; Hom and Marks 2007). To deal flexibly with an open-ended class of games, we propose that there must be a way of defining reusable and recombinable game design mechanics that can be added and removed from a design in progress, resulting in a design that can be both reasoned about and executed as a playable game. For example, we might want to take a two-dimensional, top-down maze game and reconfigure the walls, add enemies that chase the player through the maze, add a time limit, and so on. Moreover, adding new mechanics should, as much as possible, not require complex modification of the existing mechanics, such as re-axiomatizing the entire domain in formal logic to account for the new mechanic. Instead, if we want to take an existing game and add a time limit to it, we should just be able to tack on a timer and a rule saying that the game ends when the timer expires.

Our criteria are therefore: a declarative representation (so we can reason about it) that is elaboration tolerant, meaning that we can, as much as possible, modify it by adding and removing rules rather than editing rules, especially when they shouldn't need to depend on each other (McCarthy 1998). In addition, the representation should be able to deal with time naturally, since the progression of time (whether literal time or turns) is a common feature of games. It turns out that these are the same goals that drove the development of temporal logics for commonsense reasoning, so we adopt one of them for our representation, the *event calculus*. We describe how to formalize recombinable game mechanics in the event calculus so that the resulting games are both executable as fully playable games, and can be critiqued to give feedback during the game-design process. In addition, we give an extended example of useful feedback that we can get automatically from a small game in this representation, which itself may be easily modified to produce markedly different gameplay, which we can again determine automatically.

Automated support for game design

Game design typically follows an iterative process. In commercial game design, iterations frequently alternate between design changes and play-testing. At the prototype stage, play-testing is focused mainly on finding problems with how game mechanics interact. Problems can include outright design bugs, such as the player being able to get themselves into inconsistent or dead-end situations; or more subtle problems such as the game being far too hard for the average player, or completely unbalanced. Some amount of playtesting will likely always be required to gauge subjective responses of the target audience, but we propose that a useful proportion of the process could be automated.

A declarative game representation can tell us whether certain combinations of conditions can come about, and if so, can generate a gameplay trace showing how. Game designers often use playable prototypes to generate these sorts of debug traces in play-testing; having the game in a representation that can be automatically reasoned about allows traces with particular properties to be generated on demand. The simplest feedback is that no trace is possible for some desired condition—say, a game-mechanic change made the game unwinnable, which we can determine without the designer having to play through repeatedly to figure that out. More subjectively, looking at gameplay traces can let us find things that weren't supposed to happen; and looking at traces with particular properties can let us judge subjective properties. For example, if beating a Zelda-like game never requires more than a single weapon to dispatch all enemies, that indicates unbalanced game design that renders the rest of the weapons in the game redundant. Or, if a platformer can only be cleared using a jump exactly at the maximum of how far a player can jump, it might be too hard.

Game-design novices are even more in need of design support. There are tools such as *GameMaker* and *Alice* to ease *implementing* games, but not any to help with designing the game mechanics. An inexperienced designer will typically iterate by trying things out and then playing the result themselves—a tedious process that slows down experimentation, and often results in undiscovered gameplay bugs left in the final result. This process could be sped up by giving designers feedback on common design errors while they're implementing. In addition, novice designers could benefit from a toolkit of common design mechanics to easily experiment with, ideally with suggestions on when to use them.

In both cases, however, feedback from a declarative representation can't fully replace play-testing (even if only by the designer themselves), so it's important that the same representation be playable as well, at least as a prototype—a designer is not likely to want to implement a declarative version of their game as a separate, unplayable model in *addition* to a conventional prototype.

The event calculus

The event calculus is a temporal logic based on *fluents*, which are predicates whose truth values vary over time; and *events*, which happen at particular instants in time and can change the truth values of fluents. When modeling games, fluents represent game state (player position, score, enemy status, etc.), and events represent either events within the game (such as a collision, or an enemy being killed) or player input to the game (such as the player pressing up on the direction pad). Both fluents and events can have parameters, so for example the fluent $At(player, x, y)$ says that the object $player$ is at position (x, y).

Semantics

We use the version of the event calculus described by Mueller (2006), who follows Miller and Shanahan (1999). Four basic predicates specify the relationships between timepoints, events, and fluent truth values: $Happens(e, t)$ says that event e happens at time t; $HoldsAt(f, t)$ says that fluent f is true at time t; $Initiates(e, f, t)$ says that if event e occurs at time t, then fluent f will be true after t; and $Terminates(e, f, t)$ says that if event e occurs at time t, then fluent f will be false after t. These predicates can be combined with standard first-order logic to specify game mechanics. For example, we can say that two objects collide if they're ever at the same point: $\forall t, o_1, o_2, x, y : [HoldsAt(Loc(o_1, x, y), t) \land HoldsAt(Loc(o_2, x, y), t)] \rightarrow Happens(Collide(o_1, o_2), t)$.

Fluents follow the "commonsense law of inertia" by default: their truth values don't change unless changed by an event. This is one source of the event calculus's elaboration tolerance, since inertia keeps us from having to write axioms saying when every bit of state *doesn't* change: If no rule moves a wall, then it doesn't move (if we do want it to move, we just tack on a game mechanic that moves it). Inertia can be turned off by two predicates, $ReleasedAt$ and $Releases$, which take the same arguments as $HoldsAt$ and $Initiates$ respectively.

We turn off inertia in two cases: derived game state and continuous change. Derived state gives shorthand notation for a combination of other game state elements. Since it is defined in terms of other game state, it isn't directly changed by events, but should change whenever the other state changes. For example, we might say that a player can pick up an item if the item is available to be picked up and the player's inventory isn't full. Continuous change *is* primary game state, but changes continuously rather than due to specific events; an example is a missile trajectory. The event calculus provides two additional axioms to make such change easier to specify: $Trajectory(f_1, t_1, f_2, t_2)$ specifies that if fluent f_1 is initiated by an event at time t_1, then fluent f_2 will be true at time $t_1 + t_2$. For example, we could represent the trajectory of a missile traveling rightwards at a constant one unit per second, starting from the time and position at which it becomes active, by: $\forall t_1, t_2, x, y : HoldsAt(At(msl, x, y), t_1) \rightarrow Trajectory(Active(msl), t_1, At(msl, x + t_2, y), t_2)$. The $AntiTrajectory$ predicate is the same, except that the trajectory is initiated when some fluent becomes false.

In addition to the commonsense law of inertia, the event calculus aims at elaboration tolerance by using circumscription, a form of non-monotonic reasoning, to minimize the extension of the $Initiates$, $Terminates$, and $Releases$ predicates. Informally, this means that events only initiate,

terminate, or release fluents when we either explicitly say they do, or it can be inferred that they do from something else that we say. While the commonsense law of inertia allows us to assume that fluents don't change unless some event changes them, circumscription also allows us to assume that an event doesn't change any fluents unless we say that it does. This keeps us from having to write down, for example, statements specifying that the player jumping doesn't change the score.

Using the representation

Given an event-calculus formalization of a game, we'd like to do two main things: play it, and ask questions about it.

Playing a game is done by temporal projection—simulation forward in time via logical deduction. A full game will have a set of game mechanics that specify the initial game state and the rules of state evolution. With recombinable mechanics, this translates into a set of event-calculus statements saying what $HoldsAt$ time $t = 0$ (the initial configuration of the game world), along with a set of statements saying when events happen and what state they change (the game mechanics). To these axioms, we add statements saying what user input events happened at time $t = 0$ (if any). We then use circumscription on the $Happens$ predicate, meaning that we'll assume events only happen if we explicitly said they did (*e.g.*, for user input) or if a game mechanic makes them happen (*e.g.*, for collisions). From this combination of statements, we can deduce what the game state is at time $t = 1$. We repeat the process to continue to additional timesteps. The game state at any given time is simply the truth value of fluents. Since watching a readout of fluent values is only a "playable" game in a low-level debugging sense, we graphically display some fluents' values to produce the on-screen representation (with the rest remaining as internal state). For example, if $At(sprite, x, y)$ is true at the current timestep, we draw $sprite$ at position x, y on the screen.

Asking questions about a game is done by planning—finding sequences of events that would make a particular condition true, via logical abduction (Shanahan 1989). For example, we can ask if the player can ever get to a particular position on screen by asking for sequences of events that would make $At(player, 2, 2)$ true. If we didn't want the player to be able to get there at all, the sequence gives us a debug trace explaining how it could happen. Alternately, we might have thought it was possible to get there, but didn't foresee some of the possible ways of doing so; the traces let us figure out why. Based on looking at some of them, we could further refine the planning, and ask for only traces in which the player gets to position $(2, 2)$ without first having killed any enemies.

We use Mueller's Discrete Event Calculus Reasoner to perform both types of reasoning; it handles a version of the event calculus with discrete time steps, and compiles to a boolean satisfiability (SAT) problem (Mueller 2004).

Defining and organizing the mechanics

To specify and reuse game-design mechanics, we need a way of factoring the game-design process into various compo-nents, which helps organize the mechanics and relate them to other aspects of the design.

Factoring game design

In previous work (Nelson and Mateas 2008), we proposed factoring the game-design process into four areas. A game consists of *abstract mechanics*, which specify state and state evolution; *concrete representation*, the audiovisual realization of game state; *thematic content*, the real-world references a game makes; and *control mappings*, the ways the player interacts with the game. For example, an abstract mechanic like "being chased" might be represented concretely by running around a maze to evade an enemy. The maze might make thematic references to a medieval dungeon, and control might be via a direction pad. Changing any of these elements would result in a different version of the game.

We extend this view by noting that concrete representation may, in addition to directly representing abstract game state (such as a meter representing an abstract time limit), also contain concrete mechanics. For example, a two-dimensional Zelda-style game assumes a number of concrete mechanics, such as moving around, not being able to move through certain types of obstacles, collision detection, and so on. These concrete mechanics can map to elements of the abstract mechanics; for example, colliding with a coin in the concrete world abstractly results in picking it up and increasing the player's money total. Abstract state can also influence the concrete mechanics; for example, gaining an ability to walk through walls would modify the concrete collision-detection mechanics.

A game formalized in the event calculus therefore consists of abstract mechanics, concrete mechanics, connections between the two sets of mechanics, mappings from controls to either or both sets of mechanics, and references to thematic elements from both sets of mechanics. Connections and mappings are themselves a type of game mechanic, the simplest of which is a causal connection: the player dies (abstract event) when she collides with an enemy (concrete event), or moves up (concrete state change) when pressing up on the direction pad (control event). Thematic mappings are not represented in event calculus, but instead involve common-sense reasoning about real-world objects and references (Nelson and Mateas 2007; 2008).

Vocabulary and mechanics

To specify mechanics, we first define sets of *vocabulary*, which contain fluents representing bits of state, events that relate to that state, and event-calculus axioms defining the semantics and relationships of the fluents and events. Individual mechanics can then be specified by reference to one or more sets of vocabulary.

Figure 1 shows several sample sets of vocabulary. The first represents a simple inventory system. It defines a new object sort (equivalent to a type in programming languages), *item*, which is a subsort of *object*, the sort we use as a generic top level for all abstract objects. It also defines a fluent, *InInventory*, that specifies whether an item is in the player's inventory at a particular time, and three events that involve the

```
(defvocab 'inventory
  "A simple inventory system."
  :sorts '((item object))
  :fluents '((InInventory item))
  :events '((Gain item)
            (Lose item)
            (UseOn item object))
  :axioms
    '((Not (HoldsAt (InInventory item) 0))
      (Initiates (Gain item) (InInventory item) time)
      (Terminates (Lose item) (InInventory item) time)
      (Implies
          (Happens (UseOn item object) time)
          (HoldsAt (InInventory item) time))))

(defvocab 'tile-world
  "A simple 2d tile system."
  :sorts '(sprite)
  :fluents '((At sprite integer integer)
             (HasSprite object sprite))
  :events '((Collide sprite sprite))
  :axioms
    '((Implies
          (And (!= sprite1 sprite2)
               (HoldsAt (At sprite1 x y) time)
               (HoldsAt (At sprite2 x y) time))
          (Happens (Collide sprite1 sprite2) time))))

(defvocab 'controller
  "A generic controller; four dirs, two buttons."
  :events '(Dpad-up Dpad-down Dpad-left Dpad-right
            Button-a Button-b))
```

Figure 1: Several vocabulary definitions.

```
(defmechanic 'single-use-item
  "Using ?item makes it disappear from inventory."
  :vocab 'inventory
  :template-vars '((item ?item))
  :axioms
    '((Implies (Happens (UseOn ?item object) time)
               (Happens (Lose ?item) time))))

(defmechanic 'move-up
  "Event ?event moves sprite ?sprite up a tile."
  :vocab 'tile-world
  :template-vars '((event ?event)
                   (sprite ?sprite))
  :axioms
    '((Implies
          (And (HoldsAt (At ?sprite x y) time)
               (Happens ?event time))
          (And (Initiates (At ?sprite x (- y 1)) time)
               (Terminates (At ?sprite x y) time)))))
```

Figure 2: Game mechanics using the vocabulary in Figure 1.

stances of the mechanic will substitute a specific item for that variable. The second mechanic, using the *tile-world* vocabulary, specifies that some event causes some sprite to move up a tile; both the event and sprite can be filled in to make different uses of this mechanic. This particular mechanic provides an obvious way to connect concrete mechanics with user input: If we specify that the *?event* is *Dpad-up*, and the *?sprite* is the player's sprite (not shown in this set of vocabulary), then we've added a control mechanic. It can be reused in completely different ways as well, for example to make a wall move up a tile when the player presses a switch.

An example game

To illustrate prototyping a game with this representation, we'll specify a simple game, describe how it can be modified in a number of ways, and explain the types of automated feedback we can get to either discover some common design flaws or verify that they aren't present. For brevity, we'll discuss each area of vocabulary, game mechanic, and connection, but without showing all the code.

Abstract mechanics

Start with a simple set of abstract mechanics for a short snippet of gameplay. The player has to get a key to open a locked door, and then once she passes that door, can pass through a second (unlocked) door to exit the area (and end this mini-game). We use the inventory system from Figure 1, and add to that a *door* vocabulary, which specifies doors that can be open, closed, locked, or unlocked; the events *Open-attempt*, *Open*, and *Close*; and the obvious relationships between the three events and the door state. Finally, add a vocabulary for talking about the state of the game, in this simple version just containing *PlayerWins* and *PlayerLoses* events.

Using this vocabulary, we specify that there are initially two doors, both closed, with one locked and one unlocked;

inventory, *Gain*, *Lose*, and *UseOn*. The definition is completed by four axioms that say, respectively: that the inventory is initially empty; that the *Gain* event puts an item in the inventory; that the *Lose* event removes it from the inventory; and that an item can only be used if it's in the inventory.

The second vocabulary set in Figure 1 defines a simple 2d tile system. There are *sprites*, and at any given time they may be *At* some position, and it may be true that some object *HasSprite* that sprite; and they may *Collide* with each other. The one axiom specifies that if two different sprites are ever *At* the same position, then they *Collide*. Finally, the third vocabulary set specifies a controller with a four-directional D-pad and two buttons; the only things we need in this vocabulary are the six events corresponding to those six possible input signals. The *tile-world* vocabulary defines a concrete representation, so we graphically display its game state (the values of its fluents): at each time t, if $HoldsAt(At(sprite, x, y), t)$, we display a sprite at tile (x, y). The *controller* vocabulary defines input,

Figure 2 shows two mechanics we can define using this vocabulary. The first, using the *inventory* vocabulary, makes an item single-use: If we use an item, we lose it and no longer have it in the inventory. As shown in this example, mechanics can be templated. The axioms that implement this mechanic refer to a template variable *?item*; specific in-

and a key, not in the player's possession. We add several abstract mechanics for how these objects relate. If the player uses the key on a locked door, it becomes unlocked (a more complicated game might also check if the key matched the door it was being used on). If the player passes through the door that was initially unlocked, the game ends.

Concrete mechanics

To flesh this game out, we need some concrete mechanics, and connections between them and the abstract mechanics. We'll use a slightly more complicated version of the tile-world vocabulary from Figure 1, adding a width and height of the world, and a notion of impassable sprites (things like walls that other sprites can't move onto). For convenience, we define a derived fluent saying whether a particular tile is *Passable*, which is true unless the tile is either off the edge of the world or occupied by an impassable sprite.

Start with six sprites: two walls, two doors, a key, and the player. We use $HasSprite$ to connect the two door sprites to the abstract doors they represent, and similarly with the key and the player (the walls only function in the concrete mechanics). We say that the wall sprites are impassable. We add four mechanics giving the obvious effect to the player pressing the four directions on the D-pad: they move in the requested direction if the tile in that direction is passable. In addition, we specify that if the player collides with a sprite representing an inventory item (the key, in this example), it gets added to their inventory. Furthermore, any item that's added to a player's inventory has its sprite disappear from the concrete world. If the player collides with a sprite representing a door, they first attempt to use a key on it if they have a key; and then they attempt to open it. If the door is still closed, they get bounced back to the tile they were previously on. Then specify an initial configuration where the two walls and the locked door block off a corner of the world in which the unlocked door (which exits from the game) is located; the player and the key are on the other side.

Critiquing and modifying

Since the game is represented declaratively, we can query it to exhaustively explore implications of the design. For example, can the player win? Planning tells us yes: we can get a sequence of events leading up to a win. Looking only at input events, the shortest possible sequence is a sequence of direction-pad presses that leads to the key, then the locked door through the wall, then the unlocked exit door. We could look at series of abstract events too, to see that the item-gaining and door-unlocking we expect occurs; or at the series of concrete events, to see the collisions. We can directly ask whether the abstract game's sequence of events is as we intended, too: If we ask if the game can end without the player having the key, the answer is no. If we make minor changes to the game, they have the expected effects: removing the key from the world, or sticking a wall in front of the door both make winning the game impossible. Of course some of these answers could be gotten by hand-coding a lower-level test, such as analyzing reachability of tiles from other tiles to catch level-design problems. The queries here, though, can deal with arbitrary changes in mechanics—if

we remove the concrete mechanic that makes colliding with a key add it to the inventory, we find that winning is impossible despite everything being reachable, since there's now no way for the player to pick up the key (a problem with the mechanics rather than the level design).

Consider another minor change: swap out our controller vocabulary for a similar vocabulary that uses an 8-direction D-pad instead of the 4-direction one. Now add four more mechanics with the obvious effect, letting the player move diagonally with the diagonal button presses. Now if we ask our series of questions again, we find that the game is still winnable, but also find that it can be won without the player ever getting the key—not what we intended! The event-calculus planner gives us a sequence of events for winning without getting a key that demonstrates the problem: the player can jump through walls diagonally, since crossing walls that only touch at corners doesn't require ever passing through an impassable tile. Thus a bug in the concrete mechanics led to a bug in our abstract model of intended gameplay. The problem here is obvious, but similar problems arise even in shipped games, where the unexpected interaction of low-level mechanics results in the player being able to get somewhere they shouldn't be or at a time they shouldn't be there. Most of these bugs are caught in playtesting, by looking through debug logs when the play-tester manages to break the game somehow. Using the planner here to generate a plan for exiting the level without getting the key is in effect giving us a debug log for that particular problem on demand. Again, we found the problem only by querying the abstract representation: If we want to check that some bit of gameplay can never be seen out of order, we simply ask that directly, and get a debug log with a counterexample regardless of whether the problem was in the abstract mechanics, the concrete mechanics, or in something else.

A number of other changes can be made fairly easily to this game. In particular, adding new mechanics that don't change existing mechanics doesn't require modifying any of our existing game; we just tack on the new ones. For example, we could add a time limit, which makes the player lose if they don't exit within 30 timesteps. We could add an enemy that chases the player around, and makes the player lose if they collide. This allows for quick experimentation with adding and removing mechanics.

Related work

Game designers and scholars have proposed abstracting and cataloguing common game-design patterns in order to provide a set of reusable design elements and a language for discussing them (Kreimeier 2002; Zagal et al. 2005). These efforts have similarities to ours, but their game-design elements and patterns are specified at a fairly high level and described in natural language, with many of the details left out. They provide (as intended) elements that a human game designer could combine and reuse, but they cannot literally be combined and reused automatically, nor can we automatically determine anything about what happens when different patterns are combined.

The main previous work in formally specifying game mechanics also uses formal logic, but standard first-order logic, and in the context of two-player board games (Pell 1992; Love et al. 2006). These representations are more cumbersome and difficult to modify for our purposes. Since they have no built-in notion of events or time, everything must be specified in terms of state-transition rules, which are less natural for many types of games; in addition, the lack of an inertial property for states requires explicit frame axioms to be written out specifying what things don't change as time passes, making it difficult to modify a game simply by adding new axioms, since these frame axioms must be modified at the same time. These sorts of concerns led to the development of the event calculus in the first place, so we adopt its approach to addressing them.

Alternatively, we could have used a procedural representation for game mechanics as executable code, which could be suitably recombined given good modular design and appropriate interfaces, as some of our previous work has done (Nelson and Mateas 2007). The main disadvantage to this approach is that for any automated support to be built on top of such a class library, it would need declarative information attached to the mechanics anyway. Our approach defines the mechanics declaratively, making all the information about the mechanic available to be reasoned about. Future work might, however, explore including some code-generation to generate more efficient procedural code from the declarative representations, especially in cases where the goal is to produce end-user-playable games rather than prototypes.

Conclusions and future work

We propose that automated game-design support tools (and fully automated game-design systems) will need a representation that allows them to add and remove mechanics from games, reason about the resulting game, and produce a playable version of the game being represented. We presented an architecture using the event calculus to represent games, with a way of organizing a game's abstract mechanics, concrete mechanics, graphical display, and user input within that representation. We demonstrated a number of incremental modifications and critiques of a game design that can be done with such a representation.

Future work can take a number of directions. Most of the potential applications would benefit from a large library of game-design mechanics and common design-criticism queries, which would also serve to stress-test this approach's flexibility and scalability. Other future work diverges based on the target application.

To support expert game designers in rapid prototyping, designers will need to be able to specify their own mechanics in a reasonable fashion. While the event calculus is a more natural representation for this purpose than other formal logics, it is not designed specifically for specifying game designs, and so a number of aspects could be made simpler, with the appropriate event-calculus axioms generated from a higher-level representation in most common cases.

To support novice game designers, the requirements are somewhat different. We expect that they would mainly use mechanics from a built-in library of existing mechanics, so the work will be in making it easy for them to select mechanics and play and critique the resulting design. Much of this may take the form of heuristics for when to use particular mechanics, or which fixes to suggest for common game-design problems. A fully automated game-design system will need quite similar heuristics, but will require work on control flow instead of on the user interface.

On the logic side, compiling event calculus to a SAT problem will not scale to many types of games. Since a boolean variable is needed for each possible set of parameters to a fluent, specifying the position of a sprite on a 640x480 grid would take over 300,000 variables. Doing inference in first-order logic avoids this problem, and tools for it also exist (Shanahan 2000; Mueller and Sutcliffe 2005).

References

Hom, V., and Marks, J. 2007. Automatic design of balanced board games. In *Proc. 3rd Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*, 25–30.

Kreimeier, B. 2002. The case for game design patterns. *Gamasutra*. http://www.gamasutra.com/features/20020313/kreimeier_01.htm.

Love, N.; Hinrichs, T.; Haley, D.; Schkufza, E.; and Genesereth, M. 2006. General game playing: Game description language specification. Technical Report LG-2006-01, Stanford Logic Group.

McCarthy, J. 1998. Elaboration tolerance. In *Proc. 4th Symposium on Logical Formalizations of Commonsense Reasoning*.

Miller, R., and Shanahan, M. 1999. The event calculus in classical logic — alternative axiomatizations. *Electronic Transactions on Artificial Intelligence* 3(A):77–105.

Mueller, E. T., and Sutcliffe, G. 2005. Discrete event calculus deduction using first-order automated theorem proving. In *Proc. 5th Intl. Workshop on the Implementation of Logics*, 43–56.

Mueller, E. T. 2004. Event calculus reasoning through satisfiability. *Journal of Logic and Computation* 14(5):703–730.

Mueller, E. T. 2006. *Commonsense Reasoning*. Morgan Kaufmann.

Nelson, M. J., and Mateas, M. 2007. Towards automated game design. In *AI*IA 2007: Artificial Intelligence and Human-Oriented Computing*. Springer. 626–637. Lecture Notes in Computer Science 4733.

Nelson, M. J., and Mateas, M. 2008. An interactive game-design assistant. In *Proc. 2008 Intl. Conference on Intelligent User Interfaces (IUI)*, 90–98.

Orwant, J. 2000. EGGG: Automated programming for game generation. *IBM Systems Journal* 39(3–4):782–794.

Pell, B. 1992. Metagame in symmetric, chess-like games. In *Heuristic Programming in Artificial Intelligence 3: The Third Computer Olympiad*. Ellis Horwood.

Shanahan, M. 1989. Prediction is deduction but explanation is abduction. In *Proc. 11th Intl. Joint Conference on Artificial Intelligence (IJCAI)*, 1055–1060.

Shanahan, M. 2000. An abductive event calculus planner. *The Journal of Logic Programming* 44:207–239.

Zagal, J. P.; Mateas, M.; Fernández-Vara, C.; Hochhalter, B.; and Lichti, N. 2005. Towards an ontological language for game analysis. In *Proc. 2005 Digital Games Research Association Conference (DiGRA)*.