Modular Computational Critics for Games

Joseph C. Osborn and **April Grow** and **Michael Mateas**
University of California, Santa Cruz
{jcosborn,agrow,michaelm}@soe.ucsc.edu

Abstract

Formal game modeling tools could support the automated analysis of game rules and rapid automated playtesting, but are not widely used. Furthermore, existing game design support tools are often limited to very specific classes of game, require significant programming expertise to use or customize, or are fully-automatic tools with limited affordances for human designers.

We therefore propose a framework for authoring computational game design critics and a new game definition language (*Gamelan*) grounded in the conventions of board game rules. We show how a set of these critics could have detected specific, attested design problems in the development of Donald X. Vaccarino's influential card game *Dominion*. We also illustrate an extension of this approach to other collectible card games, turn-taking games, and games in general.

Introduction

Game design is a complex domain with few objective quality metrics. This complexity comes both from the emergent behavior of games and the strategies of their players: given a set of rules and a starting state, it can be difficult to predict what *could* happen (in terms of what the rules enable) as well as what players will *cause* to happen. Additional complexity arises in the implementation of a game design: even if the design is flawless, a program which implements it may have bugs, and working backwards from an implementation to a design is not possible in general. Since it is easier to produce an executable program from a model than vice versa, we propose a new modeling language for defining games.

Automated design support for games is an established research tradition, seeking "regression test[s] for design" (Nelson and Mateas 2009). This paper contributes to this body of work by proposing new formalisms for describing games and design issues; we believe these will be more amenable to non-programmer designers than prior approaches.

Game definition languages

For specific games or constrained genres, purpose-built tools can be written by programmers to explore the consequences of specific design decisions. Representative examples include visualizing the mobility of a player character (Bauer and Popović 2012) or synthesizing puzzles (Smith, Butler, and Popović 2013). The design decisions in question are not necessarily ones made by human designers—Browne and Maire's Ludi (2010) defined an extensive set of aesthetic criteria for a class of combinatorial games, successfully employing these judgments in an evolutionary algorithm to generate new games. These approaches do not generalize beyond their target classes.

More abstract tools for game modeling have also been invented to address the problem of design understanding and validation. The Machinations framework (Dormans 2009) models and visualizes feedback loops and resource economies. Machinations is excellent for modeling these dynamic systems, but it has little to say about the interactions between the larger game and the feedback loop. Unfortunately, modeling tools duplicate information about the game design and produce opportunities for desynchronization between the system and the model. Tools which do not produce playable games are prone to this issue; game systems interact in intricate ways, and the borders of their integration are likely to require patches and special-case rules which can only be partially represented in the modeling tool.

Game definition languages attempt to generalize the formal modeling of games by encoding particular design spaces in special-purpose programming languages. From the general game playing literature, GDL-II (Game Definition Language with Incomplete Information) extends its predecessor GDL with support for games of partial information and nondeterministic outcomes (Thielscher 2010). General game playing has not interacted much with the game design support community, but Mahlmann, Togelius, and Yannakakis used GDL as a starting point for a design language for strategy games to support the automated playtesting, appraisal, and evolution of sets of strategy game units (2011).

The prototyping tool BIPED (Smith, Nelson, and Mateas 2009) takes a more direct approach to game modeling, emphasizing search over possible sequences of game actions rather than amenability to AI play. BIPED "game sketches" are defined with an abbreviated event calculus representation (for the logical game engine LUDOCORE) which can be turned into playable prototypes or translated into a formal logic system which acts as a searchable play-trace gen-

Copyright © 2013, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

erator. Unlike Machinations, BIPED and GDL-II can represent complete games rather than just particular subsystems. A BIPED game definition is not just a model of a system in a game, it is (through translation) a game in and of itself.

Unfortunately, the event calculus representations of LUDOCORE and GDL-II bear little resemblance to the ways people naturally describe games. Board game rules are generally explained in a procedural style (do this, then do that), not as successor-state functions (if this happens, this bit of game state changes); furthermore, the event calculus has difficulty with hypothetical situations and with actions that involve complex sequencing of side effects.

Computational critics

Among the systems above that make judgments about the quality of game design artifacts, neither Ludi nor the unit evolution system offers a role for a human designer beyond reweighting predefined metrics. This is fundamentally a communication problem between human designers and automated reasoning systems: these systems require a lot of instruction and then have trouble describing their reasoning. We believe that computational critics (Fischer et al. 1993) offer a simple and powerful solution to this communication breakdown, and are a significant step towards integrating formal modeling with game design practice.

Computational critics have two main attributes: First, critics do not make any changes on their own: they are a support tool for human designers, not independent designing agents. Second, critics can be prioritized and superseded by more relevant critics based on the designer's current perspective.

To emphasize the difference between fully-automated design tools and computational critics, consider a game designer (Alice) working on a Chess variant intended for parents to play with young children. If Alice implemented a rule where the captures are determined by a coin flip, poor luck could lead the stronger player to lose. The "Competitive games reward skill" critic would therefore report a potential issue, but for cross-generational games like Alice's, that critic is superseded by another: if the weaker player is not able to win regularly, Alice must find ways to shift the balance of the game in favor of the child.

Another designer (Bob) working on a Chess variant may find through conventional human playtesting or by visualizing AI behavior that his rules lead to only a small portion of the board being used in most games. If this were undesirable, Bob could invent a new critic: "each match should involve most of the board's spaces." Using this critic could help avoid regressions as the game evolves; Bob could even promote this critic to concern all "territorial games," applying not just to his particular game or to Chess variants, but to any game which features territory control.

Computational critics operationalize domain knowledge in a modular and reusable way which keeps humans in the loop (Fig. 1 summarizes this mode of interaction). As the library of critics grows, more and more concrete knowledge will be added to the design science of games.

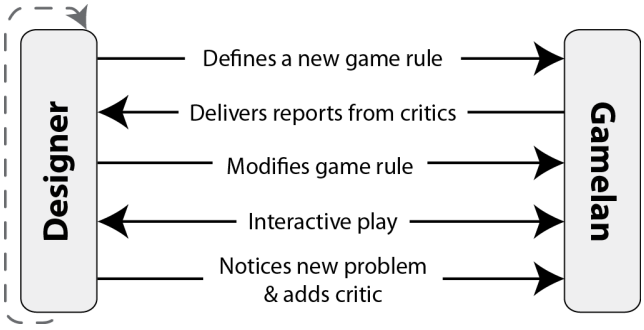

Figure 1: A designer's interactions with Gamelan.

Game modeling in practice

With the exception of Ludi, none of the non-game-specific systems described above has been used to describe complete, publishable games. Games in GDL-II, where they are complex, are generally designed to challenge general game playing programs rather than to afford enjoyable experiences to humans. Here, we propose a game definition language called *Gamelan* which mirrors the definitional style of board and card games while maintaining generality.

The two main advantages of our Gamelan language over previous game definition formalisms are ease of use and composability. Gamelan describes games in terms of procedures, rules, and state. Using board and card games as a foundation for modeling ties into game designers' existing practice of paper prototyping, in which designers build board game versions of a design in advance of a digital implementation. Gamelan is therefore more compatible with human designers' existing mental models than an event calculus representation. Gamelan's procedural semantics also make the composition of side effects easier than in event calculus representations. Our target class of games comprises single- and multi-player nondeterministic games of incomplete information over discrete domains—e.g. board, card, puzzle, and strategy games—and therefore Gamelan is as expressive as GDL-II (Gamelan is also Turing-complete, though there are readily identifiable fragments of lower complexity). Even central concerns like turn structure are defined in pure Gamelan; once defined, these concepts can be easily reused in new games.

Gamelan

In designing our modeling language, we tried to duplicate the methods used by practicing game designers to explain their games to players. Board and card game rule booklets have a fairly uniform structure and vocabulary, and the class of games they describe is quite broad. These specifications are precise and unambiguous enough that strangers can agree on the rules of play, although some prior knowledge of games is assumed—moving pieces on a board, shuffling and dealing from decks of cards, et cetera.

We closely read the rule texts of twenty board and card games which were known to the authors and whose rules were readily available online. All but one began with a catalogue of the game's pieces and some basic definitions in

a declarative style (including the game's termination condition); we call these "rules." This was generally followed by an overview of the game's turn structure phrased as a sequence of steps ("procedures") to be taken by specific players. Each step is then explained in a roughly depth-first traversal; this makes up the main body of the rule text. Special cases and exceptions to previous definitions are given as needed, and sometimes a larger set of special cases is collected following the main body. The exception to this textual format, *Stratego*, was a numbered list of mixed rules and procedures. Each procedural rule either extended the immediately preceding item in the list or began with a "when..." clause denoting its position in the game flow.

Semantics

A Gamelan game is therefore described as a set of rules (in a relational language) and a set of procedures. Procedures may await specific user inputs, make random choices, modify the game's state, and execute other procedures (possibly concurrently). Rules are side-effect-free relations and logical functions (Lifschitz 2012) defined over the game's current state. Logical functions differ from relations in that a function has exactly one value for a given set of inputs, whereas relations have no value; they can only succeed or fail. The term `logical` indicates that these functions (like logic programming relations) can be run with any combination of bound and unbound arguments (or return value), and in modes where not all input arguments are bound the query may succeed multiple times with different satisfying inputs.

The game's previous states are not forgotten, and rules can also be written which make queries in the context of previous game states. Rules may also refer to hypothetical situations arising from executing a particular procedure, with those effects being unwound after the query is made. This is a substantial improvement in convenience over GDL-II or LUDOCORE which require bookkeeping to track prior activity and often duplicate state transitions as speculative rules in order to make hypothetical queries. Hypothetical evaluation is useful in games like Chess, where a move is invalid if it would put the moving player's king in check.

Every rule and procedure has a name, and each name can be defined in multiple places. Definitions are ordered with respect to a particular name, and higher-priority definitions may take precedence over, nullify, or delegate their work to lower-priority ones. By default, earlier definitions supersede later ones. These semantics are derived from Nute's defeasible logic (2001).

In this paper, we consider a primitive version of Gamelan called *Core Gamelan*, which only includes the features described above along with some control flow statements for procedures and some aggregation operators for rules. Core Gamelan is currently implemented in XSB Prolog, with the limitations that the file-ordering precedence mechanism is not supported (all priorities must be defined explicitly) and that there is no parser for the concrete syntax.

Dominion

Dominion is a so-called *deck-building* game in which players collect cards during play that provide resources and actions, and each player's deck of cards is distinct. It is a turn-taking, adversarial, nondeterministic multiplayer game of incomplete information. On each turn, a player has the option to play exactly one of the cards in their hand, then use any Treasure cards remaining in hand to purchase a single additional card (from the Supply) whose cost is less than the sum of the value of these Treasure cards. At the end of each turn, the player's hand is discarded and a new hand of five cards is drawn from that player's deck. Each card may also provide its own rules and procedures on top of the base game: for example, a card might give its user additional purchasing power on top of the player's Treasure cards; or let the player play extra cards before proceeding to the card-buying phase; or force each other player to draw a Curse card. The game ends when three of the card piles in the Supply are empty. The player with the most Victory Points (determined by the possession of Victory cards and reduced by that player's number of Curses) wins.

The representational and computational complexity of *Dominion* would be a challenge for any modeling tool. Core Gamelan can express the game in 675 lines of code (with 385 for core rules and 290 for the game's special cards), given about 220 lines of code for background knowledge like decks and turn-taking. One C++ implementation of the same rules required more than 3000 lines of code, with over 700 lines for the special cards alone (Fisher 2012).

Syntax

In this section, some rules and procedures from *Dominion* will be used to illustrate Core Gamelan's syntax. *Dominion* is a turn-taking game, with control cycling between the game's players. Each turn procedure comprises three phases: action, purchasing, and cleanup. In Gamelan, the former is a piece of background knowledge (a predefined procedure) about turn-taking games, and the latter is Dominion-specific.

The first listing defines the `game` procedure, which has no extra parameters (the keyword `proc` indicates that a procedure is being defined). In turn-taking games, `game` cycles through each active player with the `for each player Player` looping construct (any term beginning with a capital letter or underscore is a variable; an underscore by itself is an anonymous "don't-care" variable). `game` then calls the `turn` procedure with its sole parameter set to `Player`. If the rule `game_over` is satisfied after that player's turn, the `game` procedure immediately succeeds (`pass`); otherwise, play proceeds with the next player in the game-defined turn order. Once all players have had a turn, the `game` procedure repeats itself from the top. `game` itself is called by the pre-defined `root` procedure after calling `setup_game(PlayerCount).turn(Pl)` is even simpler than `game`; it could be read as "A player's turn comprises an action phase, a buy phase, and a cleanup phase."

```
proc game
  for each player Player
    run turn(Player)
    if game_over()
      pass
  repeat
```

```
proc turn(Player)
  run act(Player)
  run buy(Player)
  run cleanup(Player)
```

The `game_over` rule is a logical relation. That means it may succeed or fail, but does not evaluate to anything. Rules are called simply by naming them along with their parameters in parentheses; if the rule has no parameters, the parentheses are not optional.

The first query in this listing passes the atom `province` as an argument to the logical function `card_supply(Type) = HowMany`, and succeeds if the result is `0`. Atoms like `province` begin with a lowercase letter (or are enclosed in single quotes) and have no parameters. An atom is a value which is distinct from all other atoms; in Gamelan encodings, atoms tend to name things like areas of a board, enumerated types, distinguished values, et cetera. Logical functions like `card_supply(Type) = HowMany`, unlike relations, have a value: in this case, the number of cards of the given type in the Supply. This listing also shows logical disjunction (`;`) and the `count` aggregator, which counts the number of satisfying bindings to `Type` such that `card_supply(Type) = 0` (the curly braces and vertical bar amount to a set-building notation).

```
rule game_over()
    card_supply(province) = 0
  ;count{Type | card_supply(Type) = 0} ≥ 3
```

Periods can be used to put multiple top-level definitions on a single line, and commas permit multiple queries or statements on a single line. Queries can also be made on the right hand side of logical functions.

```
rule card_type(estate). rule card_subtype(estate, victory).
rule cost(estate) = 2. rule vps(estate) = 1.
rule victory_points(Player) =
  sum{victory_points(CardType) |
        owner(Card) = Player, card_type(Card, CardType) }
```

Interactivity is the defining feature of games, and Core Gamelan has easy-to-use primitives for requesting user input. A `select` statement takes a query defining which players may make the choice in question, followed by a series of options. Each option is either a single value or a set of valid choices along with a series of steps to perform when that choice is made. In this case, an action phase consists of either skipping the phase or selecting an action card in the player's hand and playing it. The atom `base` followed by the symbol `::` explicitly gives a precedence level to this definition of `act(Player)`.

```
base :: proc act(Player)
  select Player
    {card_in_hand(Player, Card) |
     card_type(Card, Type), card_subtype(Type, action) }
      run play_card(Player, Card)
    true
      skip
```

Next, the `base` definition of `act(Player)` is made inferior to the `bonus` level (which we are about to define) with the `>:` operator. This statement implements the game mechanic where players may take extra actions on their turn if they have played cards which grant bonus actions. The number of extra actions is tracked in the `bonus_actions(Player)` logical function, which is only backed by `state` and modified by the `set` statement. The `after` keyword causes this sequence of steps to occur anytime the named procedure would otherwise finish. Note that `repeat` causes the whole `act` procedure to repeat, not just this block.

```
bonus >: base
bonus :: after act(Player)
  if bonus_actions(Player) = BonusActions, BonusActions > 0
    set bonus_actions(Player) = BonusActions − 1
    repeat
```

Finally, we will show the definition of a complex card, Thief. It begins with some logical functions establishing Thief as a card type, setting its cost, and describing it as both an Action card and an Attack card (Action cards are implicitly Kingdom cards). Thief's `play_card` procedure is simple: it only executes the thief's attack procedure.

```
card_type(thief). cost(thief) = 4.
card_subtype(thief, action). card_subtype(thief, attack).
proc play_card(Pl, thief)
  run attack_all(Pl, thief)
```

In *Dominion*, attacks are untargeted and apply to every other player; the core rules manage this and call `attack_player` once for each victim. To attack, the Thief first draws two cards from the victim's deck and puts them in a designated area. Next, the attacking player selects a single Treasure card (if any) out of the two set-aside cards and sends it to the common Trash pile. The remaining card (or both cards, if neither was a Treasure) is sent to the owning player's discard pile.

```
proc attack_player(Pl, thief, Pl2)
  run draw_cards(Pl2, 2, set_aside),
  TreasureCards = {C | position(C) = set_aside
    card_type(C,CT), card_subtype(CT,treasure) }
  select Pl
    {C | member(C, TreasureCards)}
      run(trash_card(C))
    empty(TreasureCards)
      skip
  for each {C | position(C) = set_aside}
    run push_card(C, player_discard(Pl2))
```

The Thief's effects do not end there, however. After each enemy's cards have been examined and possibly trashed, the attacking player may select some subset (using a range notation with `select`) of the cards trashed by this particular attack (in other words, those trashed since the attack began). `since[ProcA][ProcB]` is a query which examines the game's history; it succeeds if at any time since the last time a procedure matching `ProcA` was invoked, a procedure matching `ProcB` has also been invoked.

```
after attack_all(Pl, thief)
  select Pl 0..all
    {C | since[attack(Pl,thief)][trash_card(C)]}
      run push_card(C, player_discard(Pl))
```

Other so-called time travel queries include `as_of[Proc]{Query}` (as of the last time a procedure matching `Proc` was called, was `Query` true?) and `next[Proc]{Query}` (when a procedure matching `Proc` will next be called, will `Query` be true?). It goes without saying that `next` will always fail if the current time point (from the perspective of the query) is not in the past. Time-travel queries can be combined in various ways to achieve complex and powerful effects.

Hidden information and random choice, while not shown in the examples above, are implemented in Gamelan. Visibility uses a designated `visible(Rule,Player)` relation, and visibility is enforced by the player agents. For nondeterministic outcomes, the `select` statement may take the atom `random` as its player filter.

Modular computational critics

Critics are defined in a superset of Gamelan with access to a library of static and dynamic analysis tools. Static analysis considers Gamelan games as data structures and examines them directly. A simple symbolic execution mechanism is currently used to statically approximate a game's behavior. For dynamic analysis, we provide three player models: one which acts randomly, one which attempts to win using ensemble-determinized Monte Carlo tree search (MCTS), and a third which prolongs games for as long as possible.

Our use of MCTS for general game-playing AI is based on promising results in the related game *Magic: The Gathering* (Cowling, Ward, and Powley 2012). MCTS is not, however, a privileged method in Gamelan; any technique which generates play traces would suffice for dynamic analysis. These play trace generators are not even required to share the same host language or OS process as Gamelan; the system could call out to purpose-built AIs, general game-playing programs, or recorded games from human play.

A critic's definition includes a query which determines whether the critique applies along with a term describing the conclusion (e.g. `game_too_short` or `procedure_unused(attack)`). When Gamelan evaluates a critic, it also keeps a record of the analysis so that its reasoning can be inspected later. Examples of game design critics might include:

- No rule or procedure should go unused in a game.
- Turn-taking games should give every player equal turns.
- Repeating similar actions should be a losing strategy.
- The relative ranking of the players should shift frequently over the course of the game.

Since these critics are composable—and integrated into the same formalism designers build games from—we can give designers valuable feedback on their games with a minimum of annotation overhead.

Critiquing *Dominion*

Another reason we selected *Dominion* is that its designer has published extensive design histories of particular rules and cards that appear in the game (Vaccarino 2011); all of the quotations in this section are due to this record. It is easy to criticize game analysis tools by arguing that they operate on toy games or identify problems which would be obvious to human designers; thanks to this design history, we have specific cases of flaws reaching human playtesters (or even publication) that we believe Gamelan could have detected at design time. Even if the critics are *Dominion*-specific and could not have been devised in advance, once defined they could prevent future design bugs from creeping into the game as the rules are changed, new cards are added, or derivative games are designed.

Every game evolves and changes when it comes into contact with human players. *Dominion* is not unique for having had design issues, but its documentation affords us an opportunity to observe acknowledged design bugs in the wild.

Unplayed cards

A common concern in collectible card games is that no card should be useless: every card in the game should be there for a reason. Sometimes, a card can go unused because one card is always preferred to it in the same situations. This phenomenon is called *shadowing* in CCG parlance.

Some cases of shadowing can be detected via static analysis. In any game with custom cards, if one card type has greater benefits than another, it must have a greater cost or else the better card type *shadows* the worse one. In the example below, we assume that a designer has annotated certain game happenings with a `value` of a particular category and numeric level (for a particular player; the same move may be a benefit to one player and a cost to another). For now, we assume that values of different types are incomparable.

This gives designers a way to integrate some domain knowledge (things that are expected to be good or bad) without requiring too many annotations. Symbolic execution can determine which procedures might be triggered or which values might be changed when a specific top-level procedure is executed. This only gives an estimate of the card's value, but if enough real play traces were obtained through metrics or dynamic analysis, the behavior of those traces could be used instead of the static approximation.

value(Player, cost, proc(play_card(Player, C))) = −cost(C)
value(Player, actions, proc(get_actions(Player,N))) = N
value(Player, buys, proc(get_buys(Player,N))) = N
value(Player, money, proc(get_money(Player,N))) = N
value(Player, vp, proc(gain_card(Player,curse))) = −1
value(Player, vp, proc(gain_card(Player,Card))) = vps(Card)
  card_subtype(Card, Victory)
value(Player, deck, proc(gain_card(Player,C))) = 1
  C ≠curse

While this does place a burden on game designers, that burden is lesser for designers working in established genres than it is for designers working in brand new ones, and lesser still for designers who are only adding content to a game.

In the `shadows` critic, the function `values` infers the set of all values for the given procedure with respect to a particular player. Then, an `all` aggregator ensures that for all members of `C1Values` (each of which is a compound term `value(Type, Value)`), either there is no value of

that type in `C2Values` or the corresponding value is lower than the one in `C1Values`.

```
critic(shadows(C1, C2))
  card_type(C1, action), card_type(C2, action), C1≠C2
  player(Player)
  C1Values = values(Player, proc(play_card(Player, C1))),
  C2Values = values(Player, proc(play_card(Player, C2))),
  all{member(value(V1Type, V1Value), C1Values) |
    (not member(value(V1Type, _), C2Values)
    ;member(value(V1Type, V2Value), C2Values)
     V2Value <= V1Value
    )
  }
```

It is easy for a human to tell that one card shadows another, but human analysis does not scale to hundreds of cards. Although detecting an individual occurrence of card shadowing is a small victory, these detection events can also be meaningful in the aggregate. If a designer notices that one card is shadowed by many others, he might improve or reduce the cost of the shadowed card; if one card shadows many others, the shadowing card should be worse or the shadowed cards better. Tracking which cards shadow and which cards are shadowed is also helpful aid during long-running design repairs which could leave the game in a partially-broken state for some time.

As another justification for this simple shadowing critic, consider a designer who sees shadowing issues pop up repeatedly over the course of a game's design. Generally, the cards being shadowed are simple—a single benefit or a special ability that originally was unique to that card—but reducing their cost is not justifiable in the larger game. Through the difficulty of pricing variations on earlier cards, Vaccarino developed a particular design aesthetic: "If your basic version of a concept includes a bonus, you can vary the bonus and keep the cost the same. Only when you do the bonus-less version are you stuck with increasing the cost."

This is a complaint against "vanilla" cards which have only one bonus of any type or whose bonuses are only "basic" (for a game's definition of "basic" bonuses). Defining such a critic (e.g. calling out cards with only one or two types of `value`) could help future designers make new cards for *Dominion*. This critic might even generalize to all games that feature asymmetric choices and cost-benefit analysis.

Game speed

Game speed was another significant concern in the design history of *Dominion*. Many cards in the game involve actions, but not all actions require an equal number of player choices. The domain of each choice is also an important factor ("There was an attack that could steal any type of card... it slowed the game down way too much."), and this can be approximated statically.

The symbolic execution system described earlier could give estimates on the number of choices and their domain; in general, this problem is undecidable, but it could give a fast first approximation while waiting for a representative sample of play traces. This assumes annotations for `decision_duration`, `desired_game_time`, and `desired_turn_count`. The `too_slow` critic employs `duration`, a special predicate available to critics; here, it guesses the duration of a call to `turn`, given that a particular card was played sometime during that turn.

```
critic(too_slow(Card))
  card_type(Card, action)
  Time = duration[turn(Pl)]{
    since[turn(Pl)][play_card(Pl, Card)]
  }
  DesiredTurnTime = desired_game_time() / desired_turn_count()
  not between(Time * 0.8, DesiredTurnTime, Time * 1.2)
```

If a designer sees this critic come up frequently, he can find the common thread and write a more game-specific rule which is less likely to produce false positives. In *Dominion*: "Spy is slow to resolve... ideally Spy-type attacks don't have +1 action, or don't involve a decision, or both."

Conclusion

In this paper, we introduced our game definition language Gamelan along with sophisticated tools for analyzing games. These analyses use the framework of computational critics to provide useful context: a perspective, and a justification in terms of game elements and events. We have also shown how these critics could have detected design issues in the development of the card game *Dominion*. Combining an expressive game definition language grounded in game design idiom with modular computational critics is a major step forward for automated game design support.

Future Work

Our immediate goal is to improve the ease of use of Gamelan with a parser, modules, metaprogramming, and a graphical user interface. Following this, we plan to package Gamelan and its support tools for public release. This will require expanding the library of game mechanic modules and adding more built-in critics. As a source of inspiration, we would also like to see which of Ludi's aesthetic judgments could generalize beyond that tool's constrained class.

Gamelan's performance for large games must also be improved in order to validate our dynamic analyses on games of that scale; this may involve replacing our current player models with general-game-playing programs. We also hope to improve our static analysis, potentially by incorporating existing symbolic execution systems. For games as large as *Dominion*, it is likely that a combination of static and dynamic analysis will be required, with each feeding information to the other; the complexity of such games is too large for real-time feedback using only one technique or the other.

Acknowledgements

Special thanks to Donald X. Vaccarino for designing *Dominion* and for freely sharing the game's design history. Thanks also to Adam Smith and Peter and John Mawhorter for feedback on Gamelan's semantics.

References

Bauer, A., and Popović, Z. 2012. RRT-Based Game Level Analysis, Visualization, and Visual Refinement.

Browne, C., and Maire, F. 2010. Evolutionary game design. *IEEE Transactions on Computational Intelligence and AI in Games* 2(1):1–16.

Cowling, P. I.; Ward, C. D.; and Powley, E. J. 2012. Ensemble Determinization in Monte Carlo Tree Search for the Imperfect Information Card Game Magic: The Gathering. *IEEE Transactions on Computational Intelligence and AI in Games* 4(4):241–257.

Dormans, J. 2009. Machinations: Elemental feedback structures for game design. In *Proceedings of the GAMEON-NA Conference*.

Fischer, G.; Nakakoji, K.; Ostwald, J.; Stahl, G.; and Sumner, T. 1993. Embedding computer-based critics in the contexts of design. In *Proceedings of the INTERACT'93 and CHI'93 Conference on Human Factors in Computing Systems*, 157–164. ACM.

Fisher, M. 2012. Provincial: An AI for Dominion.

Lifschitz, V. 2012. Logic programs with intensional functions. In *Proceedings of the Thirteenth International Conference on Principles of Knowledge Representation and Reasoning*.

Mahlmann, T.; Togelius, J.; and Yannakakis, G. 2011. Towards procedural strategy game generation: Evolving complementary unit types. *Applications of Evolutionary Computation* 93–102.

Nelson, M., and Mateas, M. 2009. A requirements analysis for videogame design support tools. *Proceedings of the International Conference on Foundations of Digital Games* 137–144.

Nute, D. 2001. Defeasible logic. *Web Knowledge Management and Decision Support* 151–169.

Smith, A. M.; Butler, E.; and Popović, Z. 2013. Quantifying over Play: Constraining Undesirable Solutions in Puzzle Design. In *Proceedings of the International Conference on the Foundations of Digital Games*. Center for Game Science, Dept. of Computer Science & Engineering, University of Washington.

Smith, A.; Nelson, M.; and Mateas, M. 2009. Computational support for play testing game sketches. *5th Artificial Intelligence and Interactive Digital Entertainment Conference* 1–6.

Thielscher, M. 2010. A general game description language for incomplete information games. In *Proceedings of AAAI*, 994–999.

Vaccarino, D. X. 2011. The Bible of Donald X.