Automated Playtesting with RECYCLEd CARDSTOCK

Connor Bell, Hendrix College

Mark Goadrich, Hendrix College

Game designers benefit greatly from playtests of their prototypes in development. However, finding experienced, independent playtesters for repeated plays is often difficult. This paper describes RECYCLE*, a card game description language, and* CARDSTOCK*, an implementation of* RECYCLE *which automatically playtests card games with both random and intelligent players. Our system can assist game designers by providing insight into the average player decision branching factor, turn order advantage, game length, and the potential for strategy. We demonstrate its use by playtesting variants of the games Agram, Pairs and War.*

1 Introduction

WHEN game designers seek to create fun and engaging games, they strive to avoid common design pitfalls. First, players can struggle from 'analysis paralysis' in a game with too many viable choices [1, 2]. Alternatively, a game with too few options leaves little room for player agency and control over their fate [3, 4]. A game must find a balance between the opportunity for strategy and the size of the branching factor of options available to the players.

Second, it is easy to inadvertently create a game that is unbalanced, giving certain players an inherent advantage in the game simply based on their turn order. A game should strive to provide a fair experience for all players [5]. Finally, an end condition of a game could be created that is either unachievable or only winnable through extended play. Modern board and card games tend to limit playing time to a reasonable length of less than an hour [1].

However, the rules of a game combined with the choices and motivations of the players can be categorised as a complex adaptive system [6]. This means that emergent properties such as fairness and game length cannot be readily understood from examining the rules in isolation, but are only discoverable in the moment of play.

In pursuit of these virtues of meaningful choice, fairness, and appropriate length, game prototypes typically undergo multiple iterations and modifications while being developed. Depending on feedback from early playtests, a game designer could seek to improve the play experience by changing some elements of the game, for example, by tweaking the point values, altering the mechanics, restructuring the winning conditions, or modifying the components [7]. Each of these changes constitutes a new game variant that must be subsequently playtested. This creates a cycle that can quickly exhaust the time and patience of the available playtesting volunteers.

We introduce RECYCLE, a card game description language, and CARDSTOCK, an implementation of RECYCLE with which designers can automatically playtest card games. After encoding the rules of a game in RECYCLE, designers can perform multiple Monte Carlo simulations of a game to explore the emergent properties and learn the shape of its play space. By separating the game description from the game engine, designers can focus exclusively on modifying the rules and mechanics, and then quickly test for unexpected side-effects using both random and semi-intelligent automated players.

We focus our work on card games because they provide a small yet interesting space of games for automated playtesting tools. Many card games incorporate randomness, via the ability to shuffle and deal from a fixed set of cards without replacement, and hidden information, either through playing cards face down or through player ownership of sets of cards [8].

We first examine related work in the development of card game description languages, followed by a detailed explanation of the elements of RECYCLE and a sample encoded game. We then briefly discuss the CARDSTOCK implementation and the options for automated players. We demonstrate the utility of RECYCLEd CARDSTOCK by playtesting variants of three card games to discover their strengths, weaknesses and potential for strategy. Finally, we discuss RECYCLE's limitations and conclude with thoughts on future work.

2 Related Work

Thielscher [9] states that a general game description language 'includes: knowledge of the players and the initial position; the legal moves, and how they affect the position; and the terminating and winning criteria'. When creating a description language for the elements and mechanisms of card games, one must therefore include the deck, card locations, suit-following rules, and point values of cards.

Font *et al.* [10, 11] described a context-free grammar $G_{cardgame}$ that serves as a card game description language [12]. They demonstrated its use with implementations of Texas Hold'em Poker, and reduced versions of Uno and Blackjack. While much of their work was dedicated to exploring evolutionary mutations of these games, their core language showed the feasibility of such a system, and contained support for card locations, tokens for tracking points, clearly defined player turns, and conditions which must be met to advance game flow.

However, Font *et al.* made a number of simplifying assumptions in $G_{cardgame}$.

- All games must use the standard French 52-card deck.
- Each player has only one hand of cards.
- The game table can hold multiple locations for cards, but they all face up, except for the source deck, which faces down.
- It is awkward to move cards from one location to another, relying on intermediary moves to accomplish passing cards to another player.
- Card precedence is fixed at the beginning of the game and cannot be altered based on game state.
- Player turn order is also fixed at the beginning of the game, and while players can pass or drop out, they cannot alter their turn order.
- Each game is a sequence of stages, allowing no way to repeat earlier stages.
- There is no clear separation between player actions and control flow, such that players, and not an external arbiter, determine when a given stage of a game is complete.

These assumptions hamper the broad applicability of $G_{cardgame}$, eliminating many mechanisms and genres from their game space, especially trick-taking games. The following section discusses our grammar RECYCLE and how we overcome the above limitations.

[1]https://www.pagat.com/last/agram.html

3 RECYCLE

RECYCLE is a game description language that allows for an algorithmic representation of common core mechanisms and elements of card games. RECYCLE stands for REcursive CYclic Card game LanguagE, referring to the primary feature of the language: the recursion of game stages containing cycles of player turns. The language resembles the LISP programming language [13], often having a large number of nested instructions that control the flow of the games.

The process of writing game rules in a natural language can be fraught with ambiguities, often necessitating clarifications after publication. Encoding a game in RECYCLE can be useful for illuminating the underlying formal structure of a game design, providing insight into avenues for targeted or large-scale refinement, and resolving potential ambiguities.

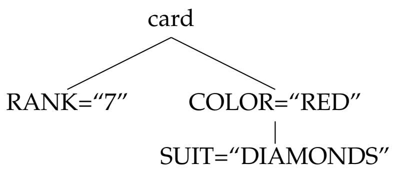
RECYCLE contains several features common to most programming languages, such as Boolean operators (`and`, `or`, `!=`, `==`, etc.), integer manipulation (basic arithmetic), and comments for programmers (using `;;`). Blocks of RECYCLE code indicate data elements, control flow, or conditional execution. RECYCLE also includes reserved words which correspond closely to the semantics of card games.

Other features that are more unique to RECYCLE, such as moving cards, keeping score on a scratch pad, cycling through the players, and assigning value to cards, are explained in the following sections. We use where possible Agram[1] as a running example.

Agram is a simple Nigerian trick-taking card game for 2 to 6 players. Players are dealt six cards from a reduced French deck, and play six tricks. To win a trick, players must follow the suit of the lead player with a higher card; there is no trump suit. The object of the game is to win the last trick.

3.1 Cards

At the heart of every card game, there are cards and players. Cards in RECYCLE are represented as key/value trees, where certain attributes can be represented as children of others. For example, in the standard French deck of 52 cards, the suit is indicative of the colour, such that Hearts are always red, Spades are always black, etc. The card '7 of diamonds' might look like the following tree:



This makes the card more decomposable than just an association of keys and values. The `deck` block creates a set of cards, one for every possible combination of attribute values. A call to `deck` which constructs the standard deck of 52 cards looks like the following:

```
(deck
  (RANK (A, 2, 3, 4, 5, 6, 7,
         8, 9, 10, J, Q, K))
  (COLOR
    (RED (SUIT (HEARTS, DIAMONDS)))
    (BLACK (SUIT (CLUBS, SPADES)))))
```

In this example, RECYCLE creates combinations of all ranks with all suits, and the colours come along with their respective suits. To create multiples of a type of card, an integer parameter is added before the attributes to indicate how many copies are desired.

The only reserved word in the snippet above is `deck`; all of the keys and values in the tree are user defined, requiring no other declaration or context for the terms. In RECYCLE, we use the convention that reserved words are lowercase, and user-defined elements are capitalised.

3.2 Card Locations

Cards are often given meaning in a game based on their organisation into physical locations. Locations for cards are associated with either the game or with a player. They can be invisible or visible locations, denoted as `iloc` and `vloc` in RECYCLE. When the players in the system request the game state, all cards in locations which are `vloc`s, and those `iloc`s associated with that player, will be completely known. All cards in other `iloc`s, belonging to either the game or other players, will be hidden and therefore unknown.

Cards can be instantiated in a location using the `deck` command as follows:

```
(create deck (game iloc STOCK)
       (deck ...))
```

Cards can be retrieved from a card location using either an integer index, or the keywords `top` and `bottom`. These cards can then be moved between locations, using the `move` command:

```
(move (top (game vloc DISCARD))
      (top (game iloc STOCK)))
```

This command moves the top card from `DISCARD` to become the top card in `STOCK`, both table locations as indicated by the keyword `game`. To move multiple cards at once, the two locations can be followed by an integer *n*, or the keyword `all` to continue moving cards until the first location is empty.

The command `size` takes a location and returns an integer with the value of the number of cards in that location. For example to find the size of the `DISCARD` location, one would say:

```
(size (game vloc DISCARD))
```

3.2.1 Memory vs Physical Locations

The card locations and actions discussed above conform to the idea that a physical card can only be in one physical location at any time. A card can also exist in any number of virtual memory locations, or `mem`s, at a time, however, these 'locations' only signify game state, not the physical presence of the cards. For example, in many trick-taking games, the `mem` location `LEAD` is used to remember the card which led the trick. Once the first player chooses a card to play, the card is 'remembered' as the `LEAD` card.

```
(remember
  (top ((current player) vloc TRICK))
  (top (game mem LEAD))))
```

All memory locations are visible to all players, but players should only base their decisions on the physical locations of the cards.

3.2.2 'Where' Clauses

Depending on the game state during execution of a RECYCLE program, a card location may contain an assortment of cards. Filtering the cards at a location is a necessary feature for enforcing the rules of many card games. Again, in many trick-taking games, it is important to ensure that the subsequent players follow the lead player's chosen suit if possible. To this end, consider the following code:

```
(!=
  (size
    ((current player) iloc HAND
      where (all
        (== (cardatt SUIT each)
          (cardatt SUIT
            (top (game mem LEAD))))
          ))) 0))
```

This checks if the current player's hand contains at least one card for which the suit of the card matches the suit of the top card of the memory location `LEAD`.

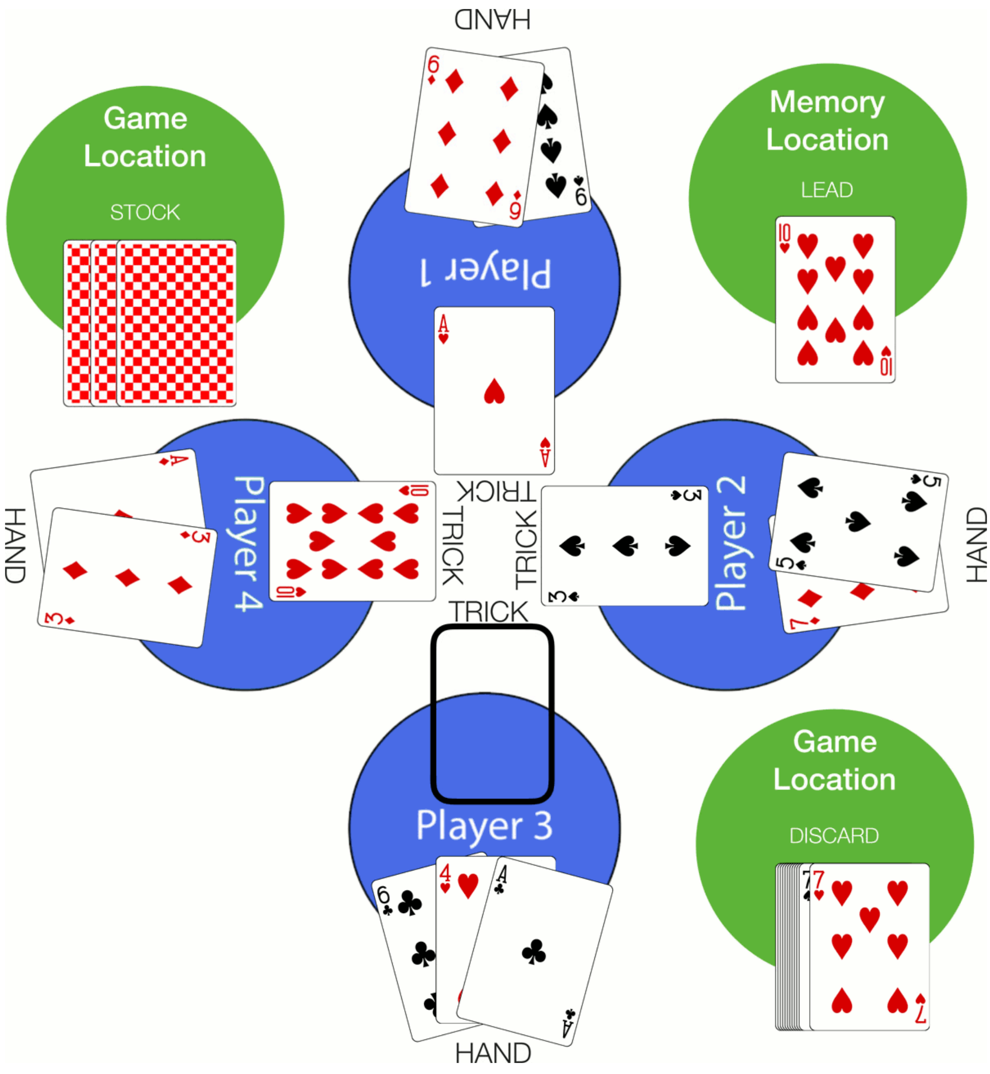


Figure 1. A sample game state of Agram.

3.3 Card Ownership

Another relationship implemented in RECYCLE is card ownership. Given a card, the function `owner` will return a reference to the player who owns the card, allowing indirect access to integer storage and other card locations for that player.

3.4 Integer Storage

Typically some type of storage is necessary to keep track of the underlying game information over time. This is often represented by tokens or chips in the game, or recordings on a scratch pad. RECYCLE offers integer storage collections for players, teams and game, as follows:

```
((current player) sto TRICKSWON)
```

The functions `set`, `inc`, and `dec` respectively modify the value of an integer storage by setting to, incrementing by, or decrementing by a given value. The given value can be a literal integer, e.g. `5`, the result of reading the value of another storage, or the result of evaluating an expression.

Figure 1 depicts the internal state of a game of Agram where the fourth trick is in progress. Players have a hand and a trick card location. The game remembers the value of the lead card. Each of these elements can be captured by the above blocks in RECYCLE.

3.5 Control Blocks

Three operations are used to control game flow: `choice` to present game action options to the player; `comp` to perform scripted computer moves; and `stage` to nest and aggregate other `choice`, `comp`, and `stage` blocks.

3.5.1 Choice

Choice blocks present the current player with a collection of actions. The keyword `any` can be used in place of a specific index when accessing a card from a location, allowing the representation of rules such as: *play any card from your hand to the table which has the suit Hearts*. The engine will enumerate all cards in the hand location and present all possible actions to the player, asking them to choose one:

```
(move (any ((current player) iloc HAND
  where (all (== (cardatt SUIT each)
                  HEARTS)))))
```

Actions often have preconditions, so to account for these, choice blocks have the following structure:

```
(choice
  ((condition)
    ...actions...
  )
  ((condition)
    ...actions...
  )
)
```

This allows for intermixing of choices of which the player gets to pick one. If there are no conditions for an action, such that it should always be available, this can be represented with the empty condition `()`. Note that each condition is checked every time, regardless of the outcomes of the others. It is often helpful to make the conditions mutually exclusive, resulting in only one specific set of actions enumerating each round.

3.5.2 Computer Moves

Computer move blocks, or `comp`, indicate actions which should be performed once by the computer each time they are encountered. They typically correspond to the game actions performed by a designated dealer. In contrast to `choice` blocks, actions in `comp` blocks should not contain the keyword `any`. Common tasks in these blocks include simple actions such as shuffling the deck or dealing cards to each player, or complex actions such as evaluating the cards in a location to determine a player's score or assigning a point to the player who is the owner of the card with the maximum score across selected players' card locations.

3.5.3 Stage

Stages are the foundational building block of control flow in RECYCLE. Stages are similar to while-loops in a procedural language, but they also naturally cycle the value of `(current player)` or `(current team)` depending on whether it is a `(stage player ...)` or `(stage team ...)`. Each stage has an end-condition, which is a Boolean expressions that can query some testable statement about the game state. As long as the end-condition is not met, the stage will run and then cycle to the next player or team before reevaluating the end-condition. Consider the following `stage`:

```
(stage player
  (end (== ((all player)
             iloc HAND) 0))
  (comp ()
    (move
      (top ((current player)
             iloc HAND))
      (top (game vloc DISCARD)))))
```

This stage will continue to execute until all players have moved the cards in their hands to a discard location. As long as the end condition is `false`, the code contained will run. When executing the `stage`, the value of `(current player)` changes each time, wrapping around from the final player back to the first. As this value changes, however, it does not affect the value of `(current player)` in surrounding contexts, and will return to its prior value when the `stage` exits.

Stages can also contain other stages. In Agram, an individual deal is over when all players' hands are empty. Within a deal, each trick is complete when every player has exactly one card on the table in front of them. This relationship is represented as a nested stage, with the deal as the outer-most stage, and the trick on the inside.

3.6 Player and Team Cycles

One of the most common elements of card games is a cyclical ordering of turns. In fact, stages incorporate this behaviour by default, cycling through the players or teams each iteration. The control flow becomes more complicated when the ordering is interrupted, so RECYCLE offers several functions to temporarily interrupt the ordering. These calls start with `cycle` and are followed by two parameters: when the change should take place and who should go instead.

The first parameter can have one of two values: `current` or `next`. Using `current` results in an immediate change, while `next` applies the change when the stage loops. The second parameter can be an integer, `current`, `next`, or `prev`.

If no changes are applied during the iteration following a change, the stage will proceed normally. Consider the following examples from the game Uno. To reverse the order, in the event of a reverse card being played, the following `cycle` action is performed:

```
(cycle next prev)
```

If a skip card is played, the following `cycle` action will skip the next player by setting the current player to be the next player:

```
(cycle current next)
```

3.7 Scoring

RECYCLE offers two functions to handle win conditions: `initialize points` and `scoring`. Together, these calls represent card precedence and the declaration of winners.

3.7.1 Precedence

In card games, precedence is the idea of a card having more *value* than another. For example, in Agram, the lead suit will have more value than any other suit. Also, aces have more value than tens, which in turn outrank nines, etc. This can be translated into a mathematical function which, given the rank and suit of a card, awards that card a number of points. For a given trick, the following RECYCLE code defines such a function, depending on the suit of the `LEAD` card:

```
(initialize points PRECEDENCE
  (
    (all (SUIT (cardatt SUIT
      (top (game mem LEAD)))) 100)
    (all (RANK (A)) 14)
    (all (RANK (10)) 10)
    (all (RANK (9)) 9)
    (all (RANK (8)) 8)
    (all (RANK (7)) 7)
    (all (RANK (6)) 6)
    (all (RANK (5)) 5)
    (all (RANK (4)) 4)
    (all (RANK (3)) 3)
  )
)
```

In this example, the function can be referred to by its name, `PRECEDENCE`. It will return an integer for each card evaluated against the rules. For example, if Hearts was the `LEAD` suit, the 3 of Hearts would be awarded 100 points for having the suit Hearts and 3 points for its rank, for a total score of 103. These functions need not map to all possible values or be one-to-one. If the function is not one-to-one, it is possible that the `max` or `min` functions will not be able to find a unique card, resulting in undefined behaviour.

3.7.2 Winners

The last element of a game description in RECYCLE is always `scoring`. For convenience, `scoring` is implicitly a `stage player`, so references to `(current player)` within the call are valid and encouraged, as it would be atypical for a game to assign a score to a player without knowing which player was in question. The `scoring` block has two parameters: the first is either `min` or `max`, indicating whether the winner is the player with the lowest score or the highest, and the second is any block of RECYCLE which returns an integer and, as mentioned earlier, will likely include a reference to `(current player)`.

3.8 Agram Encoding

To illustrate how the previous elements can be combined to create a complete game, Listing 1 shows a full description of Agram in RECYCLE.

In lines 2 through 10, the number of players are defined, the teams are defined as individuals, indicating no alliances, and the deck is instantiated to the `STOCK` location. Because the rules for Agram dictate that there is no Ace of Spades, two separate `create deck` calls were necessary.

In lines 12 through 14, the `STOCK` location, now containing all of the requisite cards, is shuffled, and 6 cards are dealt to each player.

Lines 20 through 25 handle the case in which at least the first card of the trick has been played and the player is unable to follow suit, and are therefore allowed to play any card from their `HAND` to their `TRICK` location.

Lines 26 through 33 handle the case in which at least the first card of the trick has been played and the player can (and therefore must) play a card which follows suit.

Lines 34 through 38 give the first player the freedom to play any card, and subsequently `remember` that card to the `LEAD` location, ensuring that the following players will be forced into one of the two cases above.

The `comp` in lines 39 through 53 determines the winner of each trick using the scoring function defined in lines 41 through 46, clears the `LEAD` memory location, sets the next leader to be the player who holds the winning card, and clears the cards from the last trick. In the event all of the players' `HAND` locations are empty, indicating the game is over, it awards 1 point to the player who won the most recent trick.

Line 54 declares that the winner of the game is the player with the highest value in their `SCORE` storage bin. From the prior rules, we know that only one player will receive a point each game, making this a decidable and unique maximum value and owner.

```
(game
  (setup
    (create players 4)
    (create teams (0) (1) (2) (3))
    (create deck (game iloc STOCK) (deck (RANK (3, 4, 5, 6, 7, 8, 9, 10))
                 (COLOR (RED (SUIT (HEARTS, DIAMONDS)))
                        (BLACK (SUIT (CLUBS, SPADES))))))
    (create deck (game iloc STOCK) (deck (rank (A))
                 (COLOR (RED (SUIT (HEARTS, DIAMONDS)))
                        (BLACK (SUIT (CLUBS))))))
(comp (()
  (shuffle (game iloc STOCK))
  (move (top (game iloc STOCK))
        (top ((all player) iloc HAND)) 6)))
  (stage player
    (end (== (size ((all player) iloc HAND)) 0))
    (stage player
      (end (> (size ((all player) vloc TRICK)) 0))
        (choice
          ((and (== (size (game mem LEAD)) 1)
                (== (size ((current player) iloc HAND where
                  (all (== (cardatt SUIT each)
                    (cardatt SUIT (top (game mem LEAD))))))) 0))
            (move (any ((current player) iloc HAND))
                  (top ((current player) vloc TRICK))))
          ((and (== (size (game mem LEAD)) 1)
                (!= (size ((current player) iloc HAND where
                  (all (== (cardatt SUIT each)
                    (cardatt SUIT (top (game mem LEAD))))))) 0))
            (move (any ((current player) iloc HAND where
                  (all (== (cardatt SUIT each)
                    (cardatt SUIT (top (game mem LEAD)))))))
                  (top ((current player) vloc TRICK))))
          ((== (size (game mem LEAD)) 0)
            (move (any ((current player) iloc HAND))
                  (top ((current player) vloc TRICK)))
            (remember (top ((current player) vloc TRICK))
                  (top (game mem LEAD))))))
   (comp
     (()
       (initialize points PRECEDENCE (
         (all (SUIT (cardatt SUIT (top (game mem LEAD)))) 100)
         (all (RANK (A)) 14) (all (RANK (10)) 10)
         (all (RANK (9)) 9) (all (RANK (8)) 8) (all (RANK (7)) 7)
         (all (RANK (6)) 6) (all (RANK (5)) 5) (all (RANK (4)) 4)
         (all (RANK (3)) 3)))
       (forget (top (game mem LEAD)))
       (cycle next (owner (max (union ((all player) vloc TRICK))
         using PRECEDENCE)))
       (move (top ((all player) vloc TRICK))
             (top (game vloc DISCARD))))
     ((== (size ((all player) iloc HAND)) 0)
       (inc ((next player) sto SCORE) 1))))
  (scoring max (((current player) player) sto SCORE))
)
```

Listing 1. Rules for Agram in RECYCLE.

4 CardStock

CardStock is a run-time and analytics engine for Recycle games. It is written in C# with Mono and Xamarin[2] for cross-platform support, while relying on ANTLR,[3] a parsing library, to generate the parse tree. The source code for CardStock is available from the online code repository site GitHub.[4]

In order to make our language as locally interpretable as possible, we perform run-time existence checks on all card locations and integer storage names. The first time a location is referenced, it is instantiated as an empty location, and the first time a storage is referenced, it is instantiated with the value 0.

CardStock runs simulations for a given game by iterating over the nodes in the parse tree. This iteration allows the game to be interrupted every time a choice block is encountered. This is a powerful stopping point for two reasons: the state, as well as the iterator, can be cloned with either perfect or imperfect knowledge to allow hypothetical play-throughs given the current state, and the state can be tracked for the purpose of detecting game state repetition.

To perform experiments in CardStock, a user specifies the game file to be loaded, the number of simulations to be run, and the types of players involved in each game. When run in *debug* mode, a majority of the actions performed in the game, as well as some useful state information, are reported to the user. When run in *release* mode, only the resulting scores from each game are displayed, drastically speeding up the execution time.

When the simulations have been completed, the win percentages for each player are displayed, along with the number of cycles that were detected and the average number of decisions made per game, which can be used to estimate game time. Additionally, the number of available choices at every decision, or the branching factor, is also logged throughout.

5 Players

CardStock's default players choose an action to take from a given list of possible decisions using a discrete uniform probability distribution. Simulating a game with these random players can give designers a general insight as to whether the game is behaving as expected, somewhat analogous to fuzz testing in software development [14]. This testing can also detect if the end conditions are always met or if instead the game can become stuck in an infinite loop. But if the game requires competent strategy to proceed and terminate, then random players may not provide the best information about these types of games.

To automatically incorporate intelligent players, we implemented PIPMC players, a combination of Perfect Information Monte Carlo and Pure Monte Carlo players [15, 16]. These players can clone the game state each time they are asked to make a decision, with the caveat that they randomly assign possible cards to locations that are invisible to the player. This ensures that the player's search for good moves relies on the information visible to them as well as a large number of random assignments of the other cards, without explicitly showing them information that human players would not have.

PIPMC players are not guaranteed to make optimal decisions through this *averaging over clairvoyance* approach. Within cloned game states, the card assignment will appear fixed, thus the players could fail to choose moves that would add or hide information about the game state. However, our goal is to only indicate the existence of potential for strategy [17].

For each potential action in their decision list, our PIPMC players clone twenty hypothetical games, simulate a completed game for each clone consisting of only random players, and record the outcome. PIPMC will then choose the action that earned the best average score, accounting for the minimisation or maximisation nature of the game in play.

6 Results

To date, we have encoded the following games in Recycle: Agram, GOPS, Hearts, Lost Cities, Pairs, Spades, Uno, War and Whist.[5] Here, we highlight three of these games – Agram, Pairs and War – using CardStock to explore the game design properties of turn order advantage, game length, average branching factor, and the potential for strategy.

The following experiments emulate the decision process that could have been used to discern the given game design from among competing variants. For each game and variant, we conducted 10 epochs of 100 simulations, for a total of 1,000 simulations per variant. The following results show the pooled means and standard deviations of these experiments.

[2]https://www.xamarin.com/
[3]http://www.antlr.org/
[4]https://github.com/mgoadric/cardstock
[5]https://www.pagat.com/whist/whist.html

6.1 Agram

As described earlier, Agram is a small trick-taking game. With only six cards in a player's hand and six tricks to be won, we asked: *Is there a balance between player choice and the potential for strategy?*

Figure 2 shows the average player decision branching factor with default random players. The lead player in each trick is shown separately from the three aggregated results of the three later players. We can see the effect of being forced to follow suit when possible. The lead player can always play whatever card they desire, but following players are then limited to approximately 2.5 card choices on average for the first three tricks and tapering off thereafter. There is a definite advantage to being the lead player in terms of player choice.

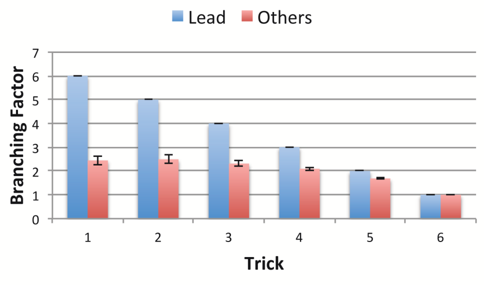


Figure 2. Branching factor for lead versus following players in Agram.

To investigate the potential for strategy in Agram, we ran simulations for two through five players, using one PIPMC with the remaining players random. We report in Figure 3 the win percentage for the PIPMC player in comparison to the expected probability of winning for a random player, given the assumption that the game is balanced.

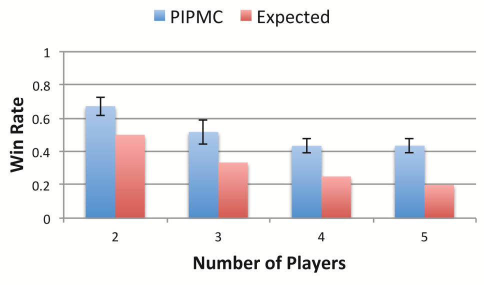


Figure 3. Gain of PIPMC player over expected value of random player in Agram.

PIPMC players are able to control their fate, outperforming the expected value by approximately 20 percentage points across all player sizes. However, there is still enough randomness in the game to confound their ability to win.

Given the above, we next investigated: *What is the smallest number of tricks that still allows for a fair game?* We can easily explore variants of Agram by changing one number in the RECYCLE description. Our first set of variants altered the number of cards dealt to each player from one to six, while fixing the number of players at two. Figure 4 shows the results for each hand size using random players.

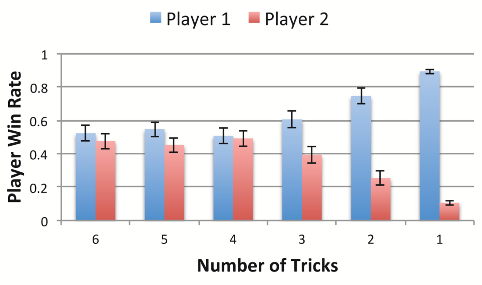


Figure 4. Fairness of turn order in two-player Agram when limiting number of cards dealt.

We can see that with four, five or six cards, the game appears fair, however, a clear bias for the first player emerges as the number of cards is reduced. We believe that because the lead suit becomes the highest precedence, it is very unlikely in such limited games that the following player is able to follow suit and thus is doomed to failure.

These results track with the known variants of Agram. In particular, the version in which players are dealt only five cards instead of six is known as Sink-Sink.[6] There are no established variants of smaller size, perhaps due to a human player's refusal to repeatedly play a game that is unfair.

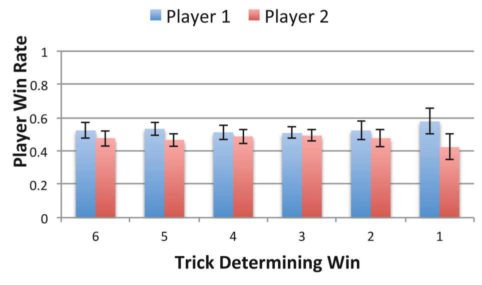


Figure 5. Fairness of turn order in Agram when varying which trick will decide the winner.

Our second set of variants retains the deal of six cards, but changes the number of tricks played

[6]Rules for Sink-Sink can be found on the Pagat page for Agram: https://www.pagat.com/last/agram.html

before determining the winner. As above, we fix the number of players at two and run simulations with random players. In Figure 5, we see that most variants are relatively balanced games, except for when the game is decided after playing only one trick.

6.2 Pairs

Pairs is a press-your-luck card game for two to six players, described by designers James Ernest and Paul Peterson as 'a new classic pub game'.[7]

Pairs uses a custom deck of 55 cards, containing one card of value 1, two cards of value 2, etc., up to ten cards of value 10. Each round players are dealt one card to a face-up hand, and then players cycle in turn order to either end the round by scoring the minimum value card in play or draw another card into their hand. If the drawn card is the same value as a card currently in their hand, the player scores that many points and the round is over. The first player to score a set number of points over multiple rounds is the loser, so players strive to minimise their points.

First, we examined the question: *how can turn order be manipulated to create a fair game?* We focussed on games with four random players. In the four-person version, the game is over when one player earns at least 16 points. Initially, we found that all players have an expected value of slightly over 9 points. This points to a fair game, however, we found evidence of turn order bias within an individual round.

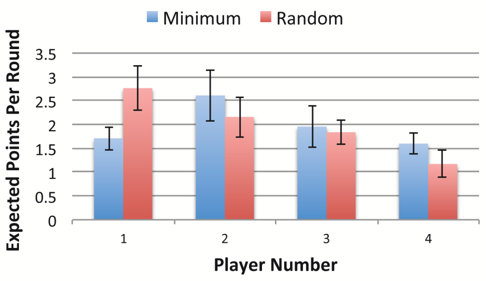


Figure 6. Unfairness of turn order in Pairs.

In these experiments, players were not given the option of stopping early, to isolate the effect of drawing another card. Figure 6 shows the expected value for each player in one individual round with two different rules to determine the first player. The rules of Pairs designate the player with the minimum valued card as the first player. This led to an unbalanced game, in which the second player has the worst expected value, as seen by the blue (left) bars.

However, we can see the benefit of this approach in comparison to a variant in which the first player is chosen randomly. For this variant, there is a strong disadvantage in being the first player, with an expected value of 2.75 points, as opposed to being the fourth player, with an expected value of 1.37. While both are unbalanced, the published rules adhere more closely to a uniform chance of winning for all players.

The designers of Pairs propose a continuous variant, where the round is not over when a player bows out or draws a pair. Instead, only the current player's hand is discarded, and on their next turn, they must take a card from the stock. This variant can be captured in RECYCLE with a single stage and much simpler structure. In simulated games with four players, we found this one-round variant to be well balanced. We also noted another property of the continuous variant; the expected value for each player rose from 9 points to 11.5 points. This variant could therefore have more tension, as all players are closer to the threshold for losing the game.

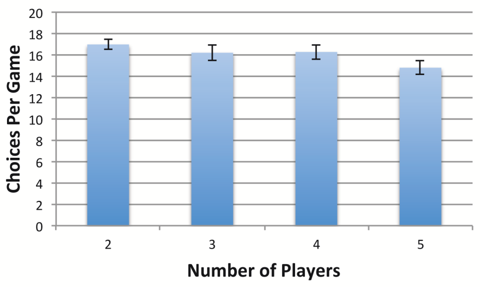


Figure 7. Game length as a function of number of players in Pairs.

Second, we investigated: *what methods can be used to keep game length constant while varying the number of players?*. The advertised game length for Pairs is 15 minutes. The rules scale the number of points necessary to lose a game by the number of players, using the formula: $(60/numPlayers) + 1$. To explore how this simple rule affects the length of the game, we simulated games with only random players for two through five players. We forego estimating the clock time for each decision, and instead report in Figure 7 the average number of calls to a `choice` block for a player decision.

We found that there is a very consistent correspondence between the number of players and the length of the game, demonstrating that the scaling is having the desired effect. A different

[7]https://boardgamegeek.com/boardgame/152237/pairs

rule could be implemented to provide perfect consistency; however, the loss of simplicity would not be worth the complication.

Finally, we asked: *with only two choices per turn, does Pairs retain the potential for strategy?* We ran simulations for 2 through 5 players, using one PIPMC and leaving the remaining players random. Since the goal in Pairs is to not lose as opposed to win, we show in Figure 8 the non-loss percentage for the PIPMC player in comparison to the expected probability of not losing for a random player, given the assumption that the game is fair. We can see the PIPMC is drastically better than the uniform random players, quickly approaching a non-loss probability of 1.

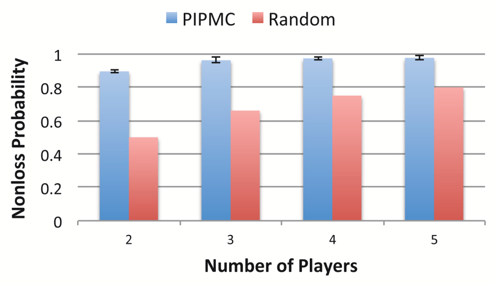


Figure 8. Gain of PIPMC player over expected value of random player in Pairs.

6.3 War

Finally, we examined War, a two-player simultaneous trick-taking game.[8]

In **War**, half of a shuffled standard deck is dealt to each player. Each round, players play the top card from their hand into the trick. The winner of the trick adds the won cards to the bottom of their hand. The goal of the game is to collect all the cards in the deck.

There is no potential for strategy in War, due to the player having only one option on each turn. However, we include this game in our analysis because RECYCLE can also detect when a game has the potential to enter an infinite cycle. In particular, we confirm through Monte Carlo experiments the results of Lakshtanov and Roshcina that certain implementations of the rules can lead to endless loops [18].

In our first implementation, we placed the cards at the bottom of both players decks in a deterministic order: the first player's card followed by the second player's card. This enabled cycles to occur in the game state. We found that 16.8% of simulated games were cyclical. After revising our RECYCLE code for War to shuffle the cards before moving them to the bottom of the deck, no cycles occurred.

6.4 Performance

As noted above, for all of our experiments, we ran each configuration 1,000 times. While execution times may vary depending on the complexity and branching factor of the game and the hardware used, the results shown in Table 1 are indicative of the times taken to conduct 1,000 simulated games of Pairs and Agram. PIPMC players can be orders of magnitude slower than random players.

Game	Random	PIPMC
Pairs	12s	14m
Agram	25s	75m

Table 1. Timings for 1,000 simulated games.

The run-time for random players reflects the average game length, but the PIPMC execution time is influenced by both game length and branching factor. The random runs for Agram were ≈ 2 times longer than for Pairs, but, because Agram has a higher branching factor than Pairs, the PIPMC runs for Agram were ≈ 5 times longer than for Pairs.

7 Conclusion

We developed a new language, RECYCLE, for describing card game states and mechanisms, and built a system, CARDSTOCK, to run automated tests on encoded games. Upon encoding games in RECYCLE, one can readily find elements to modify and potentially improve the game. Using this system, we evaluated various properties of card game designs to understand the effects of subtle rule changes within Agram, Pairs and War.

7.1 Limitations

While a wide variety of card games can be encoded in RECYCLE, there currently remain limitations to be addressed.

- Games cannot reference actual physical relationships between card locations. For example, in the Golf variant Crazy Nines,[9] players arrange cards in a 3×3 grid and earn points based on patterns in rows, columns, diagonals, or blocks.

[8]https://www.pagat.com/war/war.html
[9]https://www.pagat.com/draw/golf.html

- All game actions must be sequential. While RECYCLE can emulate simultaneous moves such as blind-bidding through some sleight-of-hand with `comp` blocks, it cannot capture the concept of an out-of-turn-order interrupt found in many 'take-that' card games, such as Munchkin.[10]

- The visibility rules apply only to card locations, not cards themselves. Individual cards in a location cannot be exposed to other players. Also, a player can always see their own locations without regard to visibility, which prevents us from encoding games such as Hanabi,[11] in which players cannot see their own hand.

7.2 Future Work

The approach taken above can be applied to many other card games. We first plan to compare Hearts, Spades and Whist, along with other members of the trick-taking family to understand their similarities and differences. Also, once a large enough library of games are encoded in RECYCLE, it will be possible to attempt to apply a distance metric, to cluster games based on their mechanisms and perhaps find new families and relationships.

Currently, our intelligent players are a baseline for detecting if player choices in a game matter. We plan to incorporate more powerful strategies such as Information Set Monte Carlo Tree Search [19], with multiple players of various strengths playing against each other, in an attempt to determine if a game is suitable for novice, intermediate, and expert players. We also hope to significantly decrease the run-time of our code so that we can efficiently examine larger games.

In a similar vein, showing that intelligent players can defeat random players in a game is not necessarily sufficient to support a claim of high game quality. It is also desirable to have multiple paths to victory, so that players cannot exploit one particular overly strong strategy. This could be accomplished by artificially limiting the choices of players to a subset of options and examining whether distinct strategies emerge.

We have begun implementing a web interface to CARDSTOCK, that will eventually allow designers to graphically encode games in RECYCLE, submit their designs for execution by our experimental setup to generate multiple runs, and then explore display graphs and statistics based on these runs.

Finally, we plan to develop a system to automatically generate games using the RECYCLE language. We believe that the properties described above, and others, can be quantified more precisely, such as divergence from fair turn order and percentage gain of PIPMC players, in order to provide clear metrics of game quality [20]. Using these statistics, we hope that an evolutionary search through the space of games possible in RECYCLE could result in the creation of novel mechanisms and interesting game designs [21].

[10]https://boardgamegeek.com/boardgame/1927/munchkin
[11]https://boardgamegeek.com/boardgame/98778/hanabi

Acknowledgements

Thanks to Brent Yorgey, Ian Shrum, and Mario Muscedere for excellent lunch-time card game discussions, and to Gabe Ferrer, Chris Camfield, John McAvey, Collin Shaddox, Laura Goadrich, and the anonymous reviewers for their feedback on this article.

References

[1] Pulsipher, L., 'The Essence of Euro-Style Games', *The Games Journal*, February 2006. http://www.thegamesjournal.com/articles/Essence.shtml

[2] Schwartz, B., *The Paradox of Choice*, New York, Ecco, 2004.

[3] Morris, D. and Rollings, A., *Game Architecture and Design*, Indianapolis, The Coriolis Group, 2000.

[4] Isbister, K., *How Games Move Us: Emotion by Design*, Massachusetts, MIT Press, 2016.

[5] Faidutti, B., 'Themes & Mechanics 4.0: A Question of Balance', *The Games Journal*, September 2005. http://www.thegamesjournal.com/articles/ThemesMechanics4.shtml

[6] Holland, J., 'Emergence', *Philosophica*, vol. 59, no. 1, 1997, pp. 11–40.

[7] Fullerton, T., 'Game Design Workshop: A Playcentric Approach to Creating Innovative Games, Third Edition', A K Peters, CRC Press, 2014.

[8] Parlett, D., *The Penguin Book of Card Games*, London, Penguin, 2009.

[9] Thielscher, M., 'A General Game Description Language for Incomplete Information Games', *AAAI*, vol. 10, 2010, pp. 994–999.

[10] Font, J. M., Mahlmann, T., Manrique, D. and Togelius, J., 'A Card Game Description Language', in Esparcia-Alcázar, A. (ed.), *Applications of Evolutionary Computation*, Vienna, Springer, LNCS 7835, 2013, pp. 254–263.

[11] Font, J. M., Fernandez, D., Manrique, D., Mahlmann, T. and Togelius, J., 'Towards the Automatic Generation of Card Games Through Grammar-Guided Genetic Programming', in Yannakakis, G., Aarseth, E., Jørgensen, K. and Lester, J. (ed.) *International Conference on the Foundations of Digital Games*, Crete, SASDG, 2013, pp. 360–363.

[12] Kowalski, J., 'Embedding a Card Game Language into a General Game Playing Language' in Endriss, U. and Leite, J. (ed.), *Proceedings of the 7th European Starting AI Researcher Symposium (STAIRS 2014)*, Amsterdam, IO Press, pp. 161–170.

[13] Steele, G., *Common LISP: The Language*, New York, Elsevier, 1990.

[14] Miller, B., Koski, D., Lee, C., Maganty, V., Murthy, R., Natarajan, A. and Steidl, J., 'Fuzz Revisited: A Re-Examination of the Reliability of UNIX Utilities and Services', University of Wisconsin-Madison, Computer Sciences Department, 1995

[15] Kupferschmid, S. and Helmert, M., 'A Skat Player Based on Monte-Carlo Simulation,' in van den Herik, J. H., Ciancarini, P. and Donkers, H. H. L. M. (ed.), *Computers and Games (CG 2006)*, Turin, Springer, LNCS 4630, 2007, pp. 135–147.

[16] Althöfer, I. and Hartisch, M., 'On the Effects of Biasing Win Conditions', *Game and Puzzle Design*, vol. 1, no. 1, 2015, pp. 50–55.

[17] Russell, S. and Norvig, P., *Artificial Intelligence: A Modern Approach*, third edition, New York, Prentice-Hall, 2010.

[18] Lakshtanov, E. and Roshchina, V., 'On Finiteness in the Card Game of War', *The American Mathematical Monthly*, vol. 119, no. 4, 2012, pp. 318–323.

[19] Cowling, P., Powley, E. and Whitehouse, D., 'Information Set Monte Carlo Tree Search.' *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 4, no. 2, 2012, pp. 120–143.

[20] Browne, C., *Evolutionary Game Design*, Berlin, Springer, 2011.

[21] Ryan, C., Collins, J. J. and O'Neill, M, 'Grammatical Evolution: Evolving Programs for an Arbitrary Language', *Genetic Programming*, Berlin, Springer, 1998, pp. 83–96.

Connor Bell is an undergraduate computer science major and Software Engineer: Tools and Infrastructure at Google.
Address: 1600 Washington Ave, Hendrix College, Conway, AR, USA. **Email:** bellca@hendrix.edu

Mark Goadrich is an Associate Professor of computer science at Hendrix College and designer of the card games Nanuk and Gene Pool.
Address: 1600 Washington Ave, Hendrix College, Conway, AR, USA. **Email:** goadrich@hendrix.edu

Shakashaka Challenge #6

Half-colour empty cells with triangles, as per the rules on p. 13. Challenge by Guten © Nikoli.