Card games as pointer structures: case studies in mobile CSP modelling

A.W. Roscoe

Oxford University Department of Computer Science

Abstract. The author has long enjoyed using the CSP refinement checker FDR to solve puzzles, as witnessed by examples in [7, 8]. Recent experiments have shown that a number of games of patience (card games for one) are now well within bounds. We discuss the modelling approaches used to tackle these and avoid symmetric states. For two such games we reveal much higher percentages of winnable games than was previously believed. The techniques developed for some of these card games – which employ various dynamic patterns of cards – suggest techniques for modelling pointer structures in CSP and FDR analogous to those used with the π-calculus. Most of these use CSP's ability to express mobile systems.

1 Introduction

The author has long enjoyed solving mathematical and combinatorial puzzles, so the advent of FDR in the early 1990's quickly led to him to try some of them out. The interested reader will find solutions to peg solitaire in [7, 8] and to Sudoku in [8]. The former has become perhaps the best known benchmark for different versions and implementations of FDR, for example in [1, 2]. These examples proved very useful for demonstrating how much better model checking is than humans at finding counter-examples in many situations, and for teaching students how to use FDR effectively. At the same time they have been a useful source of student exercises.[1]

At the same FDR has been used in a wide range of practical contexts, and the methods developed for efficiently coding these and puzzles frequently cross-fertilise.

The author had never, until recently, coded a game of cards in CSP. This was for a variety of reasons:

– The 52! different orderings of a pack of cards seems prohibitive in contemplating potential possibilities.

[1] The author gives an annual advanced course in the use of FDR and CSP, and often uses a puzzle as the first half of the take-home assignment. Previous years have included sudoku, magic squares, the unique non-trivial magic hexagon, reversible peg solitaire, knights exchange (also used as a benchmark in [1, 2]), and "jumping frogs" in multiple dimensions. The 2015 and 2016 ones were based respectively on on Freecell and Montana solitaire, two of the examples from the present paper.

– CSP's models and the way FDR works do not provide an efficient means for deciding the winner in games with multiple players trying to beat each other.[2]
– Most games of patience (i.e. card games for one, getting round the previous objection) have dynamic structures, with the cards being moved around into structures that change size or shape as the game progresses. FDR has generally been considered as much more suitable for static structures.

The first (state-space) objection can be addressed by observing that solving a *particular* deal of patience need not have anything like 52! states, though plainly the state space sizes of games with dynamic structure are potentially increased by this in a way not directly related to the number of deals. Also, the ever improving capabilities of FDR [1, 2] (for example parallelising efficiently on multi-core and cluster architectures: the 187M states of peg solitaire have been brought down to 23 seconds on our 32-core server and less than 2 seconds on a 1024 core cluster) mean that the sizes of problems that can be addressed routinely using FDR has increased greatly in the last few years. In fact virtually none of the patience game checks that the author has run from the examples quoted here exceed 10^9 states, a pretty routine level with the multi-core implementation of FDR at the time of writing, though the time taken for runs is a little more than implied by the solitaire figures above, because of the greater complexity of the CSP models. Few take more than 5 minutes exploring the state space on the server discussed here.

The second objection is addressed by considering patience games. The third is alleviated by the observation that CSP has been shown, for example in [9] and Chapter 20 of [8] to be capable of handling mobile structures.

In this paper we consider the issues of coding patience games in CSP, which includes building various dynamic structures and avoiding various ways in which symmetry can multiply state spaces. Some of these structures are in effect linked lists and other pointer-based constructions.

This work gives us insight into how to support the translation into CSP of more general programming models requiring similar structures, particularly ones based on pointers.

Most readers will be familiar with the card games called *patience* or *solitaire*, having played them via programs, apps, websites, or if they are (like the author) sufficiently old, with real cards.[3] We use the name patience in this paper, to provide a clear distinction from peg solitaire.

A game starts by shuffling one, or sometimes two packs of cards, then dealing some or all of them into a pre-set pattern. The player then manipulates the cards according to the rules of the particular game, usually aiming to arrange all the

[2] The received wisdom has been that they cannot do this at all, however in writing this paper I have discovered that there is a contrived way (that would not scale well to difficult problems) of doing it, illustrated the accompanying file `tictactoe.csp`, which analyses the winnability of the simple game *noughts and crosses* or *tic tac toe*. It shows how the algebra of CSP can calculate a general *minimax* over a tree.
[3] The latter (see figures) also proved useful in checking CSP descriptions.

cards into ordered suits in some way. A given deal is winnable if this is a possible outcome. In some cases the question of winnability is complicated by hidden cards or further nondeterminism beyond the initial shuffle. We will discuss this later.

In the next section we give background information about CSP, FDR and a few games of patience. Section 3 discusses how to code them, together with our results.

All the CSP programs discussed in this paper can be downloaded from the author's publications web page, along with this paper.

2 Background

2.1 CSP

In this paper we will make extensive use of the extension of CSP by functional programming constructs, known as CSP$_M$, and described in detail in [7]. This allows to create data types, use inbuilt structures such as tuples, lists and sets, and to write concise programs which lay out large networks of component processes indexed by such structures. Thus we can model the various components of a card game at the programming level, and then, for example, lay out the initial configuration of a game as a network of one process per card. For this reason the notation given below is that of CSP$_M$ rather than the blackboard language seen in most books and papers. The main CSP operators used in this paper are:

- `STOP` The process that does nothing.
- `a -> P` The process that communicates the event `a` and then behaves like `P`. The single event `a` can be replaced by various choice constructs such as `c!e?x:D`. This particular one assumes that `c` is a channel with two data components. This construct fixes the first to be the value of the expression `e` and allows the second to vary over `D`, assumed to be a subset of the type of the second component. If `D` is absent it varies over the whole of this type. In cases with such an input, the successor program `P` can include the input identifier `x`.
- `P [] Q` is the external choice operator, giving the choice of the initial visible events of `P` and `Q`, and then continuing to behave like the one chosen. Like a number of other CSP operators, CSP$_M$ has both the binary and an indexed form `[] x:A @ P(x)`, meaning the external choice over all the processes `P(x)` as `x` varies over the set `D`. The effect of the blackboard CSP $?x : D \to P(x)$ (which has no literal translation in CSP$_M$) can be achieved via `[] x:D @ x -> P`.
  CSP$_M$ also has `P |~| Q` as its internal, or nondeterministic choice operator, which allows the process to choose between the arguments. However this operator will not feature in this paper.
- The final constructs we will use for building sequential processes allow us to put two processes in sequence: `SKIP` is a process that terminates successfully and `P;Q` lets `P` run until it terminates successfully and then runs `Q`. Successful

termination is represented by the process communicating the special event $\checkmark$, and is distinct from `STOP`. Thus `STOP;P = STOP` and `SKIP;P = P`.

– CSP has a number of parallel operators. The ones we shall use here are *alphabetised* parallel and *generalised parallel*. Alphabetised parallel is written `P [A||B] Q`, where `P` and `Q` are respectively communicating in their alphabets `A` and `B`, with them forced to synchronise on actions in the intersection of their alphabets. Most of our uses of this operator will be in its indexed form `|| i:I@[A(i)]P(i)` where `A(i)` is the alphabet of `P(i)`, and an action in the union of these alphabets occurs just when all the `P(i)` with it in their alphabet perform it. Generalised parallel `P [|A|] Q` specifies just the interface `A` that `P` and `Q` must synchronise on, with them being free to communicate anything else independently.
– Two important operators are commonly used to structure parallel networks: *hiding* `P\X` allows internal or irrelevant communications `X` to be hidden, becoming invisible actions τ. The other is *renaming* `P[[R]]` where `R` specifies a collection of pairs of events written `a <- b`. For each such pair, every time `P` offers `a`, `P[[R]]` offers `b` leading to the corresponding state of `P`, still renamed. The relational renaming implicitly maps any event of `P` not otherwise renamed to itself. It is permissible to map one event of `P` to several different ones (which become alternatives) or several events of `P` to the same target.
– Some of our uses of hiding and renaming will have the effect of creating a model which can easily be related to a natural specification. Another operator we will use for this purpose is *throw*: `P [|A|> Q`, in which `P` progresses except that if it ever communicates an event in `A` it passes control to `Q`. Thus, like sequential composition, it represents a way in which `P` can hand control over to `Q`.

2.2 Some games of patience

There are a number of books on patience, for example [6], and many web-sites offering implementations of it. A number of standard terms are used. Quoting in part from from Wikipedia[4], the following terms describe collections of cards which exist in some games:

– *Stock* Typically squared and face-down. These cards can be turned over into the waste, usually one-by-one, but sometimes in groups of two or three (depending on individual game rules), whenever the player wishes.
– *Waste* or *Discard* or *Heap* The area where the cards from the stock go when they are brought into play. The following are typically true: The pile is squared and face-up. In most games, only cards from the stock can be played to the waste. Only the topmost card is available for play.
– *Foundation* This usually consists of the cards build up in individual suits, face up and in order. The objective of most games is to move all cards here.

[4] `https://en.wikipedia.org/wiki/Glossary_of_patience_terms`

- *Tableau* The tableau consists of a number of tableau piles of cards. Cards can be moved from one pile or area to another, under varying rules. Some allow stacks of cards that match the building requirements to be moved, others only allow the top card to be moved, yet others allow any stack to be moved.
- *Reserve* A group or pile(s) of cards where building is usually not permitted. These cards are dealt out at the beginning, and used, commonly one card at a time, during the play.
- *Cell* Common to "FreeCell" type games, cells allow only one card to be placed in them. Any card can be put in a cell. These act as manoeuvring space.

When one is writing a program to test to see if a given deal of a game is soluble, there are two important issues:

- In some games all the cards are laid face up on the table, whereas in others some are face up and some are face down or are hidden in the stock. In the former sort the player can take everything into consideration before deciding which move to play next, while in the latter there are things which would have affected the play, but the player does not have full information without cheating and taking a look.
- In most games the cards are shuffled before the start but beyond that what happens to the cards is completely in the hands of the player. However in some there is re-shuffling or some other element of randomness introduced after the start. A good example of this is the game Montana, in which the 52 cards are dealt out in four rows of 13 and the aces removed. The game is based on the spaces this removal creates. See Figure 1.

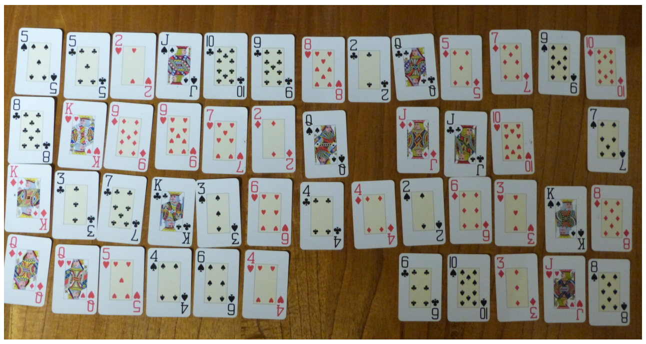

Fig. 1. A Montana deal after the aces are removed

A move in Montana is either to place a 2 in a space that appears in the first column or, in any other column, to place the card which follows (in the same suit) the card immediately to the left of the space. Thus the space

to the right of a king cannot be filled and the game becomes stuck when all four spaces lie to the right of a king with no other card in between. In the position shown in Figure 1 there are three moves possible, namely $K\clubsuit$ moved to after $Q\clubsuit$, $J\heartsuit$ after $10\heartsuit$ and $5\heartsuit$ after $4\heartsuit$.
The objective is to get the 2-to-king of the four suits in (respectively) the four rows, in order, leaving a single gap at the end of each row. Since this is difficult, the game provides that when stuck you can pick up all the cards which are not already in place, plus the aces, shuffle them, and lay them out in the vacant positions. The aces are once more removed and the game re-starts. Usually three of these re-deals are allowed.
In real life it is unknowable how these re-deals will behave, and so impossible to condition one's strategy before one of them on how they will turn out. Note that in this case a re-deal might put all the cards into exactly the right spots or place each of the aces to the immediate right of a king (so solving or immediately blocking the game).
We will not consider games of this type, and so in particular will only consider Montana in the case where there are no re-deals.

In the case of games with hidden cards, in seeking a solution to a given deal we will be finding out if the player could have performed a series of moves to win the game, which in some cases, where cards are hidden, might be lucky guesses.

So searching for a solution on a tool like FDR only models what an optimal player ought definitely to be able to achieve where there are no hidden cards and no randomisation after the initial shuffle. Aside from no-redeal Montana, two other such games are Freecell and King Albert, as described here:

– Freecell is a very popular and well understood game thanks to Microsoft including it in Windows complete with a method of generating numbered pseudo-random deals from a seed[5]. The cards are initially laid out in eight columns with seven or six cards as shown in Figure 2.
The objective is to build them up in the foundation as four suits, starting with the aces. In the basic rules cards are moved around singly[6] between the columns and the four cells, each of which can hold one card. A card can either be moved into the foundations (preserving suit order), into a cell or onto the top of a column if the top card has one value higher and the opposite colour. There are no restrictions on what can be moved into an empty column. Only the top card of each column can be moved.

[5] `https://en.wikipedia.org/wiki/Microsoft_FreeCell`

[6] Most computer implementation allow the moving of stacks of cards with descending alternate colour from one stack to another *but only provided such a move is possible given the number of cells and empty columns available.* One could produce versions of the CSP coding that allowed such extended moves. This could be done (as seems to be done in most implementations) by involving heuristics about how many cells and empty columns there are) or potentially by invoking CSP coding to carry out the moves coupled with hiding and the FDR `chase` operator that eliminates the resulting τ actions.

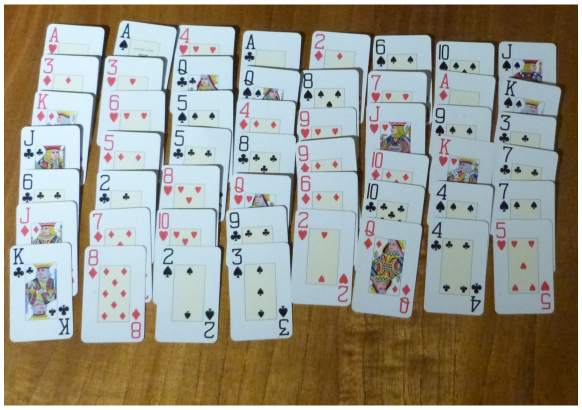

Fig. 2. Freecell (cells and foundation empty and not shown): not quite the impossible deal 11982 (see text)

Thanks to its popularity and the de facto standardisation of deal numbering, Freecell has been well studied with solver programs[7], and a good deal is known about it including that only 8 of the first million deals are not solvable. The deal shown in Figure 2 was the result of the author trying to photograph the first of these, deal 11982. However there is a mistake: the deal 11982 has 10♡ and 8♡ swapped from their true positions in that deal. It is interesting that this small change actually makes the deal solvable (needing the full 4 cells), just as deal 11982 is soluble with 5. The distributed file `freecelldeal.csp` illustrates these facts.

– King Albert (named after a king of Belgium) is less well known, and indeed was unknown to the author until he was working on this paper. In this the cards are dealt out into nine columns as shown in Figure 3, leaving seven cards as the "Belgian Reserve". This game is played exactly like Freecell except that there are no cells, and cards can only be moved from the reserve, never into it. The author has found no evidence of prior computer analysis of this game, merely a frequently repeated statement that about 10% of games are solvable (we will see later that this is wrong).

Nor has the author found any evidence for computer analysis of Montana, though one does often find the statement that about 1 in 20 games or 1 in 10 games is soluble when using the three redeals. We will see later that this is also wrong.

The final game we consider is perhaps the best-known one, Klondike [8]: see Figure 4 Unlike the ones above this does have hidden cards (initially only eight are visible, the tops of the seven columns and the top card of the 24-card stock. It is played like King Albert except that a card in one of the columns is only

[7] For example `http://fc-solve.shlomifish.org/`

[8] http://en.wikipedia.org/wiki/Klondike_(solitaire)

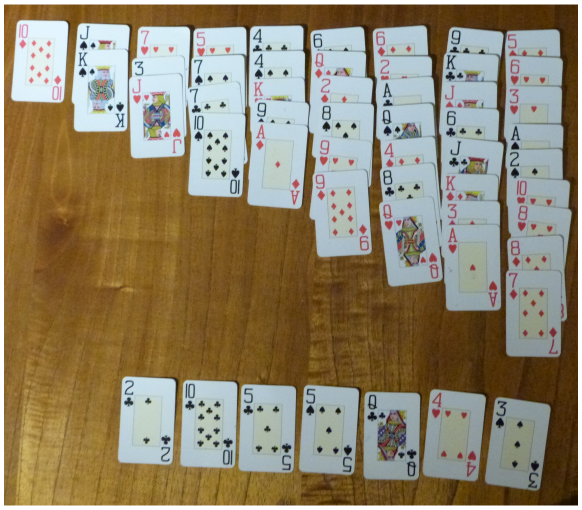

Fig. 3. A deal of King Albert, showing reserve cards below and foundation missing and empty

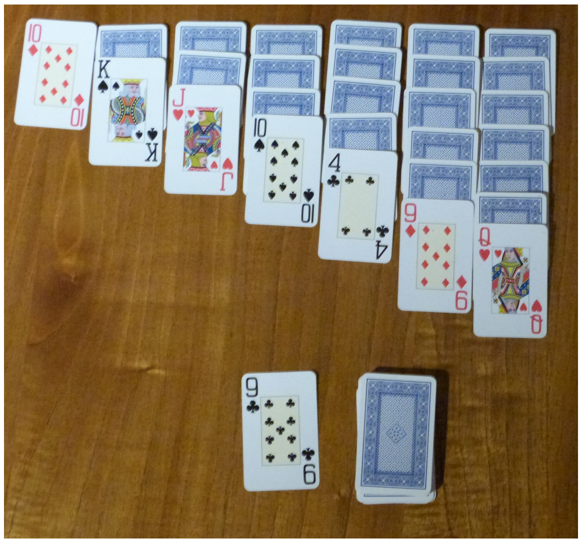

Fig. 4. Deal of Klondike, with stock and waste below, foundation missing and empty

turned face-up when it is at the top, stacks of alternating descending cards can be moved freely between columns provided the alternating descending property in maintained, but only a stack with a king at the bottom can be moved into an empty stack. Usually (unlike Freecell and most descriptions of King Albert) cards can be moved from the foundations to the columns, preserving the alternating descending property.

There are many variants on this game, mainly based on how the stock cards are dealt (typically singly or in threes, with various numbers of times through the stock). There are figures for the solubility of this game (based on perfect knowledge) of several variants[9]. In this paper we will concentrate on the case where cards are dealt singly, with no limit on the number of times a card can be turned over.

3 Modelling patience in CSP

In CSP, particularly when this is designed for input into FDR, we want models to have a fairly high degree of parallelism, since the tool is typically more efficient for these. We also want to avoid creating any inessential states, which in most games means states that do not relate directly to positions where real moves have been completed. For if we model a single move by several, especially if we allow these to interleave, the state space can grow alarmingly.

All of the games we have described have symmetries which mean there can, depending on how they are coded, be large classes of states which are completely equivalent in behaviour. In the three examples based on columns of cards, it is clear that two states with the same set of columns (though in different positions) are completely equivalent as regards solubility, as are two positions obtainable from each other by permuting the rows in Montana.

Aside from Montana, which has an essentially fixed format throughout (four rows of 13 cards or spaces), all the others change in pattern dynamically through the game as various piles and stacks of cards grow, shrink and are moved around.

There are two obvious approaches to modelling these games in CSP. One is to make each card a process, which at any time knows enough about its position to allow it to determine its ability to move or to be moved to. These cards, together with one or two other processes to monitor things like empty cells or columns, can make up the CSP model of a game. The other is to model each component of the game such as a column, an individual suit in the foundation, the stock or the waste cards. In the case of Montana it makes sense to model each position in the 4×13 grid as a separate process that is either a card or is a space (this state changing as moves are played). Each of these has its advantages and disadvantages, as revealed in the following discussion.

In the card-based solution, each card needs to know when it is at the top of a column or otherwise available for moving. It therefore has to know not only when it is moved, but also whether a card moves from or to being on top of it.

[9] `https://en.wikipedia.org/wiki/Klondike_(solitaire)`

This means that each card needs to know what card is above it and/or which is below it, noting that when a card is at the top of a column there will be none above, and none below when on the bottom.

The author dealt with this by using the first of the following data types:

```
datatype links = Crd.card | Top | Bottom

datatype suits = Hearts | Clubs | Diamonds | Spades

datatype card = Card.suits.{1..13}
```

making each column into a linked list of processes. There are arguments for making this doubly linked (each card knowing the identities of the cards above and below it), though in many circumstances one can get away with a singly linked list. However in the latter case it is still for a card to know if it is at the top of a column, and probably necessary for it to know if it is at the bottom of one, so that moving it creates an empty column.

He used actions such as `stacktostack.c1.c2` meaning that `c1` moves from the top of a column to the top of another[10], where the current top card is `c2`. Clearly the processes representing `c1` and `c2` respectively need to agree to this move, with `c2` agreeing only if it is at the top of a column and `c1` agreeing only when it is at the top or in the sorted portion in the case of Klondike. Such moves will only be possible when `c1` is of the opposite colour and one point lower than `c2`.

The the cards now at the top of the two stacks will have to know this. It follows that whatever card was below `c1` will need to synchronise on the event also or otherwise be told about its new status.

Because a card keeps the same alphabet at all times, but the card on top of it changes, its alphabet (in the case where it synchronises) will have to contain all moves of the form `stacktostack.c1.c2` where `c1` is any card which might ever be on top of it. Furthermore it will have to agree to such events: in the case where the card `c` we are defining is `c2`, it will only agree to this event when it is on top of a stack. If it is not `c2` but `c1` is on top of it, `c` knows that after the event it is on top. However (wherever it is in the game) `c` should never block the event `stacktostack.c1.c2` when it is neither `c1` nor `c2`.

This is similar to the techniques used in Chapter 20 of [8] to model mobility, where processes move around and change which ones they communicate with. There the author showed that this could be simulated in CSP by putting every event that ever be available to a process into its simulation alphabet, and having the simulation of each process that does not for the time being have an event a in its mobile alphabet agree to that event. The key way of explaining this is the CSP identity

[10] Where there is only one card that `c1` can move on top of, as when it is moved to the top of a suit, there is no need for the parameter `c2`, and where there is a restricted range of targets for `c1` we can again reduce the range of `c2`. In the common case where cards are only moved on top of one of the opposite colour and one point higher, we can replace `c2` by a suit.

```
P [A||B] Q  =  (P|||RUN(B'))[|Events|](Q|||RUN(A'))
```

where `A'` are the members of `A` not in `B` and vice-versa, provided `P` and `Q` do not terminate with ✓ and only communicate within `A` and `B` respectively. Thus the parallel composition on the left hand side, with its fixed alphabets, is equivalent to the one on the right where a pair of processes always synchronise on everything after being padded out by the two `RUN(X)` processes, which are always willing to communicate anything in the parameter set of events. By building something like a dynamic version of `RUN(X)` which changes as its accompanying process's alphabet changes, we can create the effect of `P [A||B] Q` where `A` and `B` can change as the execution progresses. That is precisely the approach set out in [8], whereas here we take the approach of building the process representing each card so that it allows the extra events required for mobility itself by allowing but ignoring events that would be relevant only if the moving card is in a different position.

So the state of a card when it is in one of the eight stacks of Freecell, the first game we consider here, is

```
CStack(c,below,above) = above==Top & stacktosuit.c?_ ->
  (NewTop(c,below); CSuit(c))
[] above==Top & stacktostack?c':skabove(c)!c -> CStack(c,below,Crd.c')
[] above==Top & stacktofc!c -> (NewTop(c,below);CFC(c))
[] above==Top & fctostack?c':skabove(c)!c -> CStack(c,below,Crd.c')
[] above==Top & stacktostack!c?c':skbelow(c) ->
                    (NewTop(c,below);CStack(c,Crd.c',above))
[] IsCard(below) and above==Top & stacktoestack.c ->
                  (NewTop(c,below);CStack(c,Bottom,above))
[] above!=Top & maketop.cardof(above)!c -> CStack(c,below,Top)

NewTop(c,c') = if c'!=Bottom then maketop.c.cardof(c') -> SKIP
                              else freestack.c -> SKIP


LSbelow(Card.S.n) = if n==Ace then Bottom else Crd.Card.S.n-1
```

Note that there are separate events for the possible places for moves to and from a stack such as a cell or suit. The separate event for moving to an *empty* stack is because this is governed by different rules to moves to a non-empty stack. There is a separate processes which monitor the numbers of empty stacks and only allows a card to move into one when one is available.

It is important to realise that a card has no idea which of the eight stacks it is in. That is because it really does not matter: all that matters is that it is in a stack and which cards are above and below it. Note that if we did have an index for which stack a card is in, this might increase the state space by a factor of nearly 8!, because any permutation of the stacks would now produce a different state except for swapping two or more empty ones.[11] This is perhaps the

[11] In practice one cannot swap stacks that are not completely sorted into decreasing alternating order, but in soluble games there are many possible permutations of all-

strongest reason why we code stacks as clusters of mobile card processes rather than a single process that represents a given stack, although another strong one is that such processes would have large state spaces.[12]

Similarly there is a very simple state for a card in a cell, with cards entering these when allowed by a separate process that monitors how many are currently filled. Just as with stacks, there is good reason not to let each card know *which* cell it is in.

```
CFC(c) =
fctosuit.c?_ -> CSuit(c)
[] fctostack.c?c':skbelow(c) -> CStack(c,Crd.c',Top)
[] fctoestack.c ->  CStack(c,Bottom,Top)
```

There is similarly a process which monitors the progression of cards into the four suits, ensuring that each suit is built up in order and also raising a flag when all four are complete. It has a further, more subtle purpose. If a card is available to put in a suit then it is not always optimal to put it there, because it may be needed to hold a lower card of the opposite colour on top of it in a stack. However if both the cards of opposite colour with value one lower are in the suits *or* both cards of opposite colour with value two lower *and* the card with value three lower and the same colour (but different suit) as c are already there then there can be no point in not moving c to its suit.[13]

The Boolean flag which is part of the event moving each card to its suit is calculated by the `Suits` process: it is true is it is safe to force the move according to the above criteria.

We force such actions to happen when they can. By itself this considerably reduces the state space, but we have another trick to play as well. We hide forced events and apply the FDR *chase* operator which forces *tau* events to happen without counting them as events in the breadth-first search. This has the effect of bringing forward states in which a lot of moves to suits have been forced, meaning that FDR typically looks at a lot less other states before finding one of these than it would without *chase*. In fact if the forcing is made optimistic: forcing all enabled moves to suits, this usually results in finding solutions faster, though in theory it could fail to find any solution when there is in fact one.

The events `freestack.c` indicating that a stack has become empty are also hidden and *chased*, because it is safe to do so and also reduces the state space by eliminating interleaving.

sorted ones. Not multiply counting these makes a huge difference to the state space of soluble games.

[12] Given that aces can be assumed never to be in a sorted stack, there are exactly $32,712 = 2^{12+3} - 12 * 4 - 8$ sorted ones.This does not take account of the unsorted cards that may be in the stack above these.

[13] The first of these cases is obvious: the only two cards c' that c might hold are already in the suits and cannot move back. The second case follows because under these circumstances it is possible and safe to move any such c' to the suits: there is no need for c to hold it.

There are several files implemented using these ideas available with this paper on the author's website. These include ones with a random deal generator that requires a seed, one that cannot be solved with four cells, and one that uses suits rather than cards as the second argument of `stacktostack` as discussed above. At the time of writing all 52-card deals attempted were within the range of the author's dual-core laptop if not offered more cells than they need, most taking a few minutes[14]

A feature common to all our card-game models is that FDR's preliminary compilation phase in which it reduces a CSPM script to the supercombinator form the checking run in, takes significant time for each deal irrespective of how many states that deal has. In the case of these Freecell models, it takes about 20s per script on the author's laptop, but this would have been considerably more if we had used processes representing multiple cards (like whole stacks) or had use a larger alphabet (for example by parameterising `stacktostack` with three cards (the card moving and the from and to ones), which is very tempting.

3.1 Other games

The coding of King Albert is very similar to that of Freecell: the only differences are that there is now one more stack, that these have different sizes, and that the seven-card reserve (to which cards cannot move) replaces the cells. This presents no challenge. The main point of interest in this game is the results obtained. Despite the statements on the web to the effect that about 10% of games are soluble, the author found that 72 of the first 100 random deals were solved, mostly fairly quickly, by FDR.

Klondike is similar to King Albert. The fact that some of the cards in columns are initially face down does not affect the coding, only the interpretation of what a discovered solution means – as discussed earlier this now depends on some luck as well as careful planning. The reserve of King Albert has now become a 24-card stock that are initially face down and turned over in groups of 1 or 3 into the waste. There is a sense that a stock that is turned over in 1s as often as the player chooses is equivalent, in terms of solvability, to a reserve: each card remaining in the stock can always be reached. On the other hand this would move away from the spirit of the game and not be adaptable to more limited access to the stock. The author therefore implemented the combination of stock and waste as a doubly linked circular list: the links in one direction enable us to turn over the cards from the stock in the correct order, and those in the other enable us to treat the waste as a stack. A header process was included in the list to detect the top and bottom of the stock, handle the process of starting a new pass through the list, and handle the case where the stock as become empty.

[14] To illustrate this, for an arbitrarily chosen random deal the author tried, 0 cells resulted in a model with only a few states and no solution, 1 cell led to a solution on ply 59 of the FDR search and 227K states, 2 cells found a solution in 40 plies but used 58M states, 3 used 34 plies and 257M states, while 4 used 31 plies and 734M states. The decrease in the number of plies with more cells is because these give the player more move options.

Cards are never added to this circular list, but we have to allow for a card c being removed, meaning that the two adjacent processes need to have the pointers which formerly pointed to c being altered. Thus the event(s) constituting the move of c to a column or suit need to tell the two adjacent cards what c's former links were. It would be preferable to to do this in a single action, but that would mean that the events would need to contain at least three cards as data fields, meaning that the alphabet size and consequent compilation time would become problematic. Therefore the author actually coded these interactions via a series of events, using a separate process `Linker` to avoid having three-card events. The events are

1. The main move of a waste card `c` to a column or suit, that card then
2. communicates its previous `below` link to `Linker`,
3. followed by its previous `above` link.
4. `Linker` then tells `above` and `below` to link to each other in place of `c`.

The last three events are hidden and the FDR function `chase` applied. This forces the three τs created from the `Linker` events to happen immediately after the main move, thereby making the four events effectively into one. This cuts down on potential interleavings and eliminates any possibility of error arising from the series of events relating to different game moves overlapping.

It would be easy to adapt the resulting CSP model to any of the usual variations in how the stock is used.

The solvability of Klondike has already been well studied (see Wikipedia article), so the author did not seek to repeat these experiments.

Unlike the three games we have discussed above, Montana maintains a rather static structure throughout the game: four rows made up of 13 cards and spaces each, with four spaces and 48 cards overall. Therefore we now have the option of maintaining a process for each of these 52 slots (which is at all times a named card or an unnamed space) as well as the option of having a mobile process for each card (and perhaps space too). While the mobile cards approach has the potential benefit of eliminating any symmetries which arise from permuting the rows, provided one does not create more symmetries by giving the four spaces separate identities, the disadvantages mean it is not practical. In summary these are:

– A process for a slot in the array only needs 49 basic states (one for each card that might be in the slot and one for a space. However to get the linked list approach to work the list needs to be doubly linked (i.e. each card has a pointer to both left and right). It also needs to know how many spaces (0–4) are on its right, making $48 \times 48 \times 5$ basic states for a card process, and each of these states needs to participate in a large number of events. This is significantly more expensive than in the other examples we have seen of doubly linked lists, because for example the only cards that can be involved in the stock and waste in Klondike are those initially in the stock.
– To maintain the state of these pointers and counters at the sites both where the card is moved from and to needs multiple communications and large

alphabets. Multiple communications lead to a large number of intermediate states.

These mean that the time taken to compile, run and debug the checks are substantially greater than with the array model. Despite using tricks such as factoring the card process into three and using a lot of renaming to avoid symmetric compilations, the author did not succeed in getting the mobile card model to run for more than a 20 card pack (once the aces had been removed), which is the smallest possible with four suits where one cannot get a complete row of spaces.

On the other hand the array model was very successful and demonstrated that Montana (not allowing any redeals) is far more frequently soluble than appears to have been believed before. 63 of the first 100 random deals were soluble with no redeals, whereas several web sites quote the figure of about 10% being soluble by a skilled player with three redeals.

4 Conclusions

We entered on this study to illustrate the expressive power of CSP, in particular in describing mobile systems, and the power of FDR to decide things about them. It is clear that these 52-card games of patience are within the scope of what can be handled by these technologies, but are sufficiently close to the boundary that we have to be careful about how to describe each game, taking careful consideration of the nature of each system from the points of view of alphabet size, compilation time and running time.

We have also illustrated a style of describing mobile systems in CSP that is different from that described in Chapter 20 of [8], where it was shown how to achieve the effect of communicating channels along channels. The approach used here is more explicit: we simply tell each process (i.e. card) who it is next to, keeping this information up to date, and the processes duly communicate with their neighbours. Thus processes are told the topology of their local network explicitly rather than implicitly.

Many of the models we created included linked structures of CSP processes strongly suggestive of the pointer structures in heap used in programming. We will develop this idea by showing in a subsequent paper how such programming styles can be supported by CSP and FDR, particularly when new *symmetry reduction* functionality of FDR is used to eliminate the differences in states created by different choices of heap cells for the same roles. That issue is essentially the same as the reason it was so much better to use the card-based linked model for games like Freecell rather than to park cards in particular named columns or cells. The same symmetry reduction technology would successfully eliminate these symmetries in card models too, but the models presented here do not require it.

The main reason for this is that each member of the linked structures we build is tied to an immutable and distinct value, namely its value as a card, whereas cells in a heap are identical.

One could well imagine producing a compiler from a suitable language for describing games of patience to CSP, if one were wanted. However one might well need to input some human judgement about what style of model (say array or mobile) would work best.

The CSP files relevant to this paper can be found in an accompanying zip file on the author's web site. Note that the one for Klondike incorporates a trick not discussed in this paper: this uses the ability of FDR to compute multiple counter-examples to determine, in a single run, what the highest score attainable with a given deal it. The style of programming for this exactly parallels one discussed for metering timing problems in Chapter 14 of [8].

References

1. T. Gibson-Robinson, P. Armstrong, A.Boulgakov and A.W. Roscoe. FDR3 – A modern refinement checker for CSP. In *Tools and Algorithms for the Construction and Analysis of Systems* (pp. 187-201). Springer 2014.
2. T. Gibson-Robinson and A. W. Roscoe. FDR into the cloud. *Communicating Process Architectures* (2014).
3. Bernard Helmstetter and Tristan Cazenave. Searching with analysis of dependencies in a solitaire card game. *Advances in Computer Games.* Springer US, 2004. 343-360.
4. C.A.R. Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985.
5. Graham Kendall, Andrew J. Parkes, and Kristian Spoerer. A Survey of NP-Complete Puzzles. ICGA Journal 31.1 (2008): 13-34.
6. Albert H. Morehead. *The Complete Book of Solitaire and Patience Games.* Read Books Ltd, 2014.
7. A.W. Roscoe, *The theory and practice of concurrency*, Prentice-Hall International, 1998.
8. A.W. Roscoe, *Understanding concurrent systems*, Springer 2010.
9. A.W. Roscoe, *CSP is expressive enough for* π, Reflections on the work of C.A.R. Hoare, Springer 2010.