DAT530
Discrete Simulation and Performance Analysis
Final Project
Solitaire game strategy

Racin W. Nygaard

Universitetet i Stavanger

Abstract. This project is such and such... +++

Table of Contents

List of Figures

List of Tables

Abbreviations

DP Draw Pile Module
FIFO First In First Out (Queue)
FP Foundation Pile Module
GUI Graphical User Interface
LIFO Last In First Out (Stack)
TP Tableau Pile Module

Nomenclature

card (In the Petri Net context) A token with a color which represents a card in the deck.
command A token with a color which represents a turn or movement command.

1 Introduction

This project aims to study the popular card game, Solitaire[Site]. Solitaire is bundled with most Windows[Site] installations, as well as being available for free on several sources. It is also easy to play the game with a physical card deck. A detailed explanation of the games rules can be found in the next chapter, Solitaire Rules[REF]
Since the game utilizes all 52 cards of the deck, the number of possible initial game states is 52!, which is a very high number. A large number of these inital game states can be merged, as they offer no difference in the difficulty to solve. Some of these initial states are unsolvable, but even given a solvable game state, one often find oneself in an unsolvable game state, due to certain actions in the game are non-reversible,. There has been attempts to find the distribution of solvable and unsolvable initial game states [ref]. This is roughly 75 percent are solvable, however the study also shows that only 35 percent of the games are won by an experienced player.
This project contains a complete model of the game, a GUI to play the game, and a basic bot to simulate user actions.

1.1 Solitaire Rules

Finite State Machine ?

2 Method and Design

2.1 Naming Policy

2.2 File structure

To reduce the number of files, most of the pre- and post-processor files of the FP and TP modules have been combined in one single file. An example of this can be shown in listing 1.1, which shows parts of `COMMON_PRE`

Listing 1.1. COMMON_PRE.m lines 1-5

```
function [fire, transition] = COMMON_PRE(transition)

if ismember(transition.name, {'tFPe_Clubs_Add', 'tFPe_Diamonds_Add', ...
        'tFPe_Hearts_Add', 'tFPe_Spades_Add'}),
    [fire, transition] = pre_tFPe_Add(transition);
```

By doing this it is possible to reduce the number of files required without overloading the `COMMON_PRE` and `COMMON_POST` files. It also makes it much easier to work and maintain the code as the logic is only located in one place, as opposed to four or seven places if each transition had their own file.

With this approach it is no longer possible to hard-code the names of the related transitions and places, so two additional functions; `get_tableau_from_transname` and `get_suit_from_transname` were developed. These functions take the name of the transition as input, and then return the unique identifier for which module it belongs to. The actual code is pretty simple, and parts of `get_suit_from_transname` is shown in listing 1.2. The reasoning behind not using the Matlab command `contains` is simply that it is not supported in older versions.

Listing 1.2. get_suit_from_transname.m lines 7-17

```
    if ~isempty(strfind(transitionname,'Clubs')),
        suit = 'Clubs';
    elseif ~isempty(strfind(transitionname,'Diamonds')),
        suit = 'Diamonds';
    elseif ~isempty(strfind(transitionname,'Hearts')),
        suit = 'Hearts';
    elseif ~isempty(strfind(transitionname,'Spades')),
        suit = 'Spades';
    else,
        suit = 0; % Invalid suit.
    end
```

2.3 Overall Design

The model developed is pretty large, and contains 94 transition and 42 places. It is developed using the modualar approach, and encompasses 6 different modules. Some of the modules are duplicated, with the only difference being the names of the transitions and places.

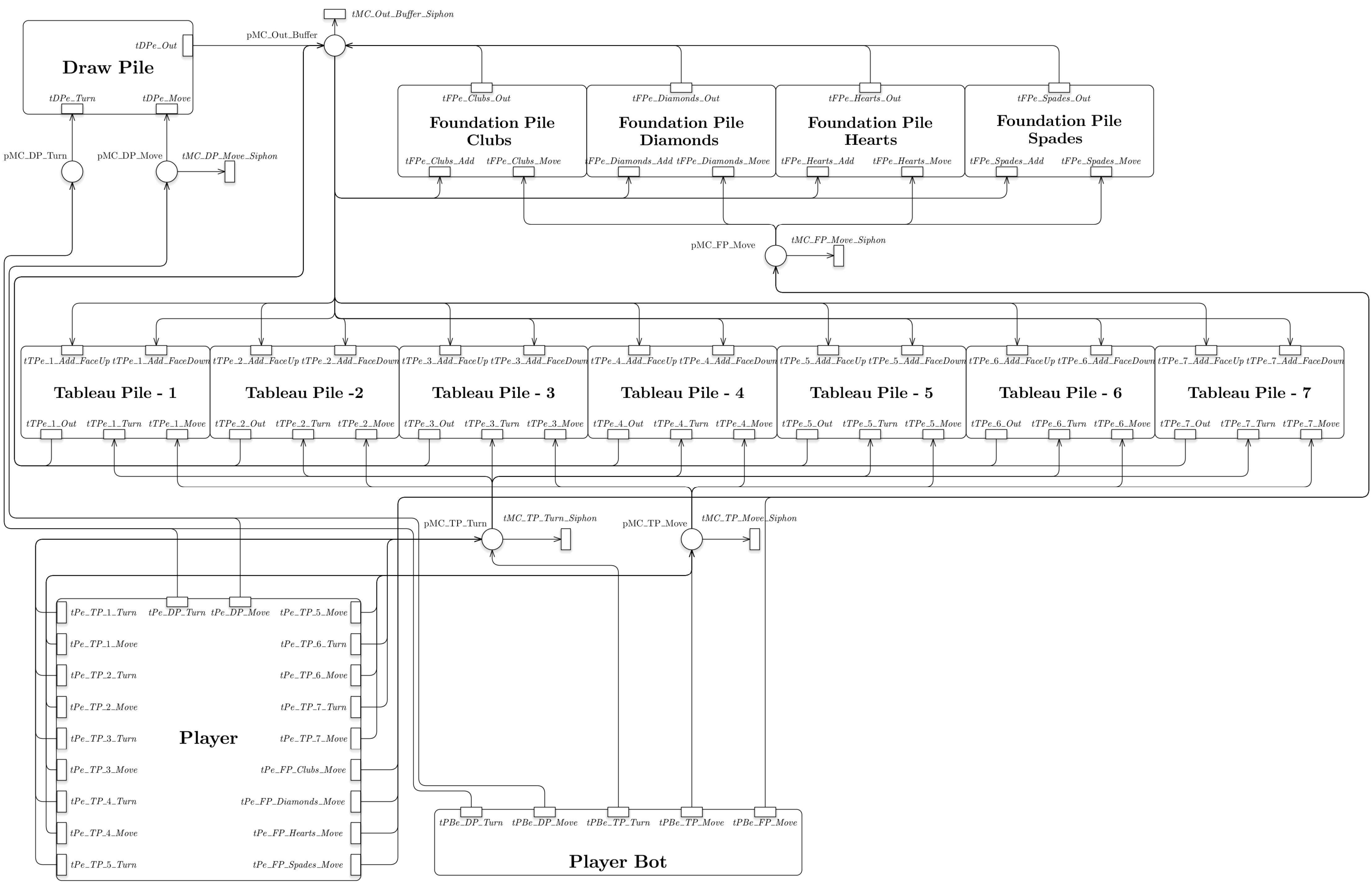

Fig. 1. The complete model - Without the internal components of the modules.

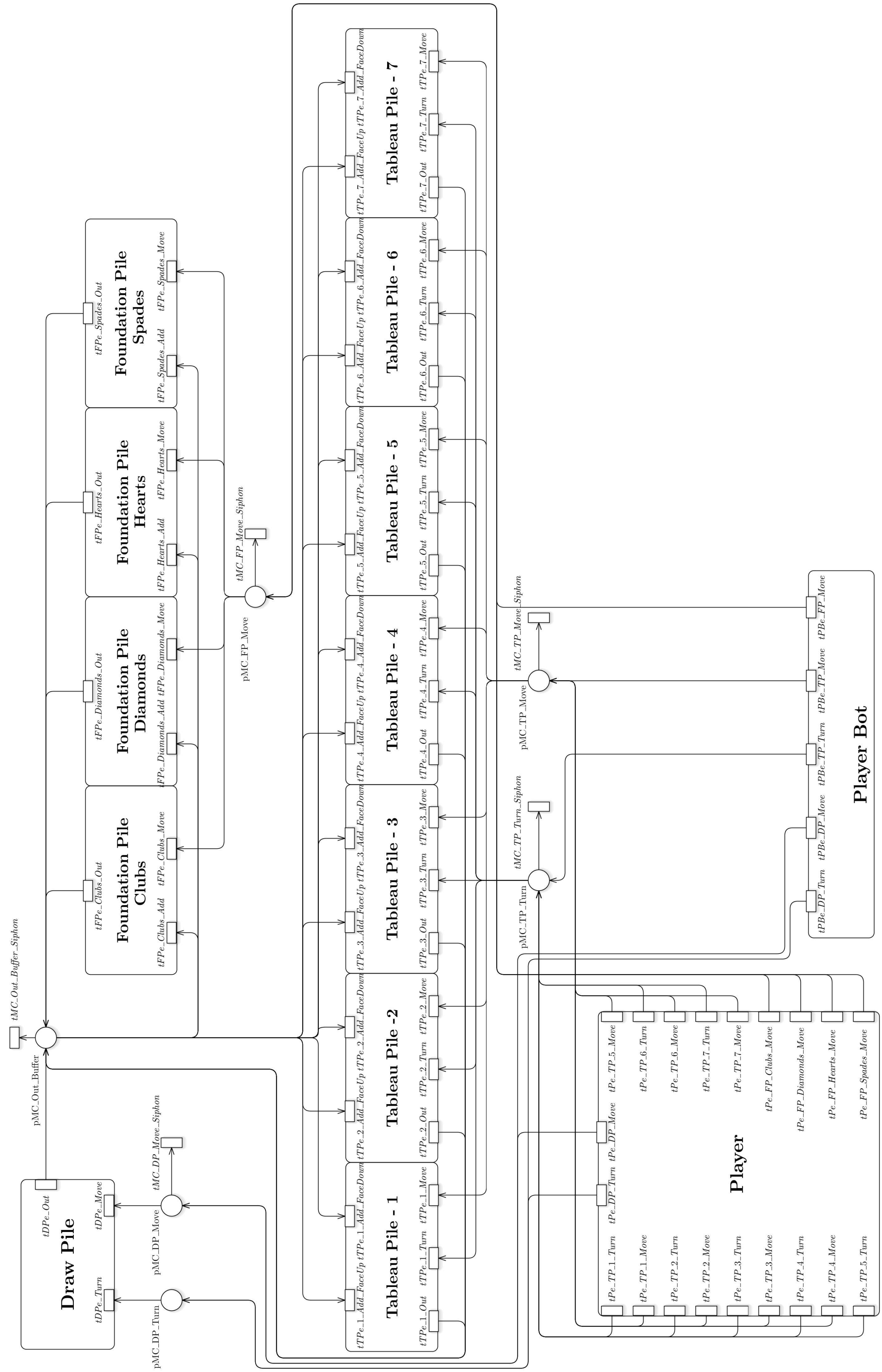

Fig. 2. The complete model - Without the internal components of the modules.

2.4 Draw Pile Module

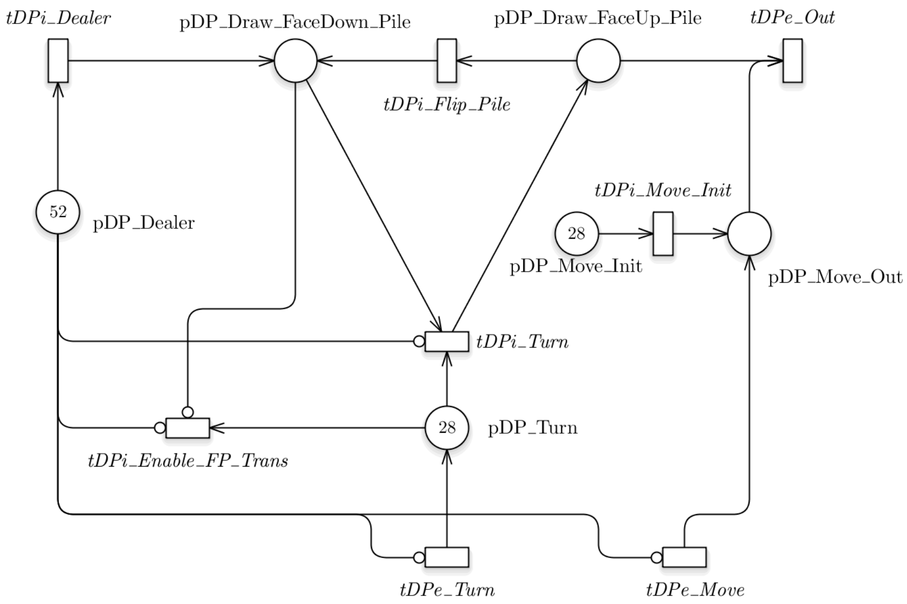

Fig. 3. Draw Pile Module

The Draw Pile module is depicted in figure 3, and has several key responsibilities, once of which is to do the initial dealing of cards. In order to preserve the correctness of the gameplay, external input is not allowed during this phase. When first running the model, all the initial tokens of `pDP_Dealer` will be sent to `tDPi_Dealer`. This transition will give each token a color which represents a card in the deck. Possible colors are initially stored in the cell `global_info.DECK`. If `global_info.RANDOM_DECK` is set, a random permutation of the colors will be given to the tokens. By having `global_info.RANDOM_DECK` set to false, it is possible to run analytics which require that the cards are dealt equally each time.

After all tokens are given a color, `tDPi_Turn` will be enabled. This transition will move cards from the pile which represents face-down cards, `pDP_Draw_FaceDown_Pile` to the one representing face-up cards, `pDP_Draw_FaceUp_Pile`. This transition will fire as many times as the length of `global_info.INITIAL_DEAL_MOVE`, which is 28 in a normal game. This is not something that would be done if the game where played with physical cards, as they would just be dealt without turning them. In this model however, this is required so that existing logic could be re-used.

Concurrently to the firing of `tDPi_Turn`, the transition `tDPi_Move_Init` will fire an equal amount of times. The transition will give each of the tokens in `pDP_Move_Init` a color which represents to which tableau pile the card should be moved to. The color given to each token is augmented by the cell, `global_info.INITIAL_DEAL_MOVE`. An example of a color given is *Move:TP1:DP* which means; *Moving a card from source DP to destination TP1.* Every time a card reaches its destined tableau pile, the variable `global_info.CARDS_DEALT` will be incremented by one in `COMMON_POST`. Once it becomes equal to the length of `global_info.INITIAL_DEAL_MOVE`, the initial dealing phase is over, and the normal phase starts.

During the normal phase, external input is allowed. The first input of the Draw Pile Module is `tDPe_Move`. This transition has an pre-processor file, which makes it only fire if there are tokens in `pDP_Draw_FaceUp_Pile`. Additionally, the Player and Player Bot modules ensures that the enabling token has color on the format *Move:(destination):DP*.

Listing 1.3. tDPe_Move_pre.m

```
function [fire, transition] = tDPe_Move_pre(transition)

fire = 0;
if ~isempty(tokIDs('pDP_Draw_FaceUp_Pile')),
    fire = 1;
end
```

The second input, `tDPe_Turn` is used to used to simply move cards from the face-down pile to the face-up pile during the normal phase. An interesting thing about this is that once all the cards are in the face-up pile, the next time one attempts to turn a card, all cards should be moved back to the face-down pile in LIFO style, just as they would if you simply flip the deck of cards around in real-life.
This is accomplished by the transitions `tDPi_Flip_Pile` and `tDPi_Enable_DP_Trans`. The `tDPi_Enable_FP_Trans` is actually an siphon, and becomes enabled once `pDP_Draw_FaceDown_Pile` is empty, and there is an active turn action on-going so that `pDP_Turn` has at least one token. The transition has one post-processor file, shown in listing 1.4. Given that there are actually any tokens left in `pDP_Draw_FaceUp_Pile` it will set the global flag, `global_info.DP_Flip_Pile_Running` to `true`, if there are no tokens in the face-up pile, it will simply release the `playerAction` resource. The use of resources is discussed further in chapter 3.5. The reason for not having an arc directly from the face-up pile is due to this transition being a siphon, so the card would be removed from the game if it fired.

Listing 1.4. tDPi_Enable_FP_Trans_post.m

```
function [] = tDPi_Enable_FP_Trans_post(transition)

global global_info;
if ~isempty(tokIDs('pDP_Draw_FaceUp_Pile')),
    global_info.DP_Flip_Pile_Running = true;
else,
```

```
    % Release playerAction resource to allow for another player action.
    release(global_info.last_command_source);
end;
```

Once `global_info.DP_Flip_Pile_Running` is set to `true` and there are tokens in `pDP_Draw_FaceUp_Pile`, the transition `tDPi_Flip_Pile` will start firing. The pre-processor file is listed in 1.5, and will keep selecting the latest arrived card from `pDP_Draw_FaceUp_Pile` and fire. In the post-processor file, listed in 1.6, it will check for the length of the face-up pile, once it becomes empty it will set the flag `global_info.DP_Flip_Pile_Running` to `false`, and the cards have been successfully turned around.

Listing 1.5. tDPi_Flip_Pile_pre.m

```
function [fire, transition] = tDPi_Flip_Pile_pre(transition)

global global_info;
fire = 0;
if global_info.DP_Flip_Pile_Running == true,
    transition.selected_tokens = tokenArrivedLate('pDP_Draw_FaceUp_Pile',1);
    fire = 1;
end
```

Listing 1.6. tDPi_Flip_Pile_post.m

```
function [] = tDPi_Flip_Pile_post(transition)

global global_info;
if isempty(tokIDs('pDP_Draw_FaceUp_Pile')),
    global_info.DP_Flip_Pile_Running = false;
    global_info.SCORE = max(global_info.SCORE - 100, 0);
    % Release playerAction resource to allow for another player action.
    release(global_info.last_command_source);
end;
```

Lastly, there is the `tDPe_Out` transition. This is the only external output of the module. When enabled, its pre-processor will take the lastest card arrived at `pDP_Draw_FaceUp_Pile`, but the earliest command arrived at `pDP_Move_Out` when firing. By taking the earliest command arrived in a FIFO manner, we ensure that the inital dealing will be correct. If we were to take the latest command, we would have to add additional logic such as alternating firing to make certain the ordering of cards would be correct. The code is shown in lising 1.7

Listing 1.7. tDPe_Out_pre.m

```
function [fire, transition] = tDPe_Out_pre(transition)

% Want to make sure that we get the earliest move-token, and the latest
% card. This is so that we can have a natural ordering of the cards during
% the initial dealing.
moveToken = tokenArrivedEarly('pDP_Move_Out', 1);
% Explicitly sure to get the card at the top of the stack.
cardToken = tokenArrivedLate('pDP_Draw_FaceUp_Pile', 1);

transition.selected_tokens = [moveToken cardToken];
fire = 1;
```

2.5 Foundation Pile Module

The Foundation Pile module is duplicated four times, once for every suit, clubs, diamonds, hearts, and spades. The only difference between these modules is the

Table 1. Transitions used in Draw Pile

	Name	Description
1	tDPe_Move	External input for the move-command
2	tDPe_Out	External output
3	tDPe_Turn	External input for the turn-command
4	tDPi_Dealer	Gives every token a color to represent a card in the deck.
5	tDPi_Enable_FP_Trans	Used to facilitate the flipping of the face-up pile.
6	tDPi_Flip_Pile	Moves cards from face-up pile to face-down pile in a LIFO manner.
7	tDPi_Move_Init	Generates initial move-commands to facilitate initial dealing of the cards.
8	tDPi_Turn	Moves a card from the face-down pile to the face-up pile.

Table 2. Places used in Draw Pile

	Name	Description
1	pDP_Dealer	Holds the initial tokens which will become cards.
2	pDP_Draw_FaceDown_Pile	Holds the face-down cards. These are not visible to the player.
3	pDP_Draw_FaceUp_Pile	Holde the face-up cards. Only the top card is visible to the player.
4	pDP_Move_Init	Holds initial tokens used for generating move-commands.
5	pDP_Move_Out	Buffer for move-commands.
6	pDP_Turn	Buffer for turn-commands.

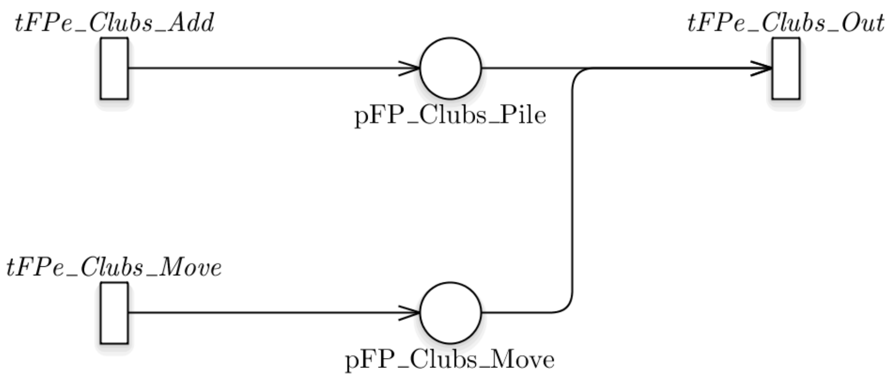

Fig. 4. Foundation Pile Module

names of their respective transitions and names, so the description given for clubs will count for the other duplicates as well. All the pre- and post-processor files are shared with other suits.

This module is inactive during the initial phase, and only becomes interactable once the normal phase starts. It has two external inputs, the first of which is `tFPe_Clubs_Add`. This transition has a shared pre-processor file, . More about in the move command can be read in chapter 3.3.1.

Table 3. Transitions used in Foundation Piles

	Name	Description
9	tFPe_Clubs_Add	
10	tFPe_Clubs_Move	
11	tFPe_Clubs_Out	
12	tFPe_Diamonds_Add	
13	tFPe_Diamonds_Move	
14	tFPe_Diamonds_Out	
15	tFPe_Hearts_Add	
16	tFPe_Hearts_Move	
17	tFPe_Hearts_Out	
18	tFPe_Spades_Add	
19	tFPe_Spades_Move	
20	tFPe_Spades_Out	

Table 4. Places used in Foundation Piles

	Name	Description
7	pFP_Clubs_Move	
8	pFP_Clubs_Pile	
9	pFP_Diamonds_Move	
10	pFP_Diamonds_Pile	
11	pFP_Hearts_Move	
12	pFP_Hearts_Pile	
13	pFP_Spades_Move	
14	pFP_Spades_Pile	

Table 5. Transitions used in Tableau Piles

	Name	Description
53	tTPe_1_Add_FaceDown	
54	tTPe_1_Add_FaceUp	
55	tTPe_1_Move	
56	tTPe_1_Out	
57	tTPe_1_Turn	
58	tTPe_2_Add_FaceDown	
59	tTPe_2_Add_FaceUp	
60	tTPe_2_Move	
61	tTPe_2_Out	
62	tTPe_2_Turn	
63	tTPe_3_Add_FaceDown	
64	tTPe_3_Add_FaceUp	
65	tTPe_3_Move	
66	tTPe_3_Out	
67	tTPe_3_Turn	
68	tTPe_4_Add_FaceDown	
69	tTPe_4_Add_FaceUp	
70	tTPe_4_Move	
71	tTPe_4_Out	
72	tTPe_4_Turn	
73	tTPe_5_Add_FaceDown	
74	tTPe_5_Add_FaceUp	
75	tTPe_5_Move	
76	tTPe_5_Out	
77	tTPe_5_Turn	
78	tTPe_6_Add_FaceDown	
79	tTPe_6_Add_FaceUp	
80	tTPe_6_Move	
81	tTPe_6_Out	
82	tTPe_6_Turn	
83	tTPe_7_Add_FaceDown	
84	tTPe_7_Add_FaceUp	
85	tTPe_7_Move	
86	tTPe_7_Out	
87	tTPe_7_Turn	
88	tTPi_1_Move_Multiple	
89	tTPi_2_Move_Multiple	
90	tTPi_3_Move_Multiple	
91	tTPi_4_Move_Multiple	
92	tTPi_5_Move_Multiple	
93	tTPi_6_Move_Multiple	
94	tTPi_7_Move_Multiple	

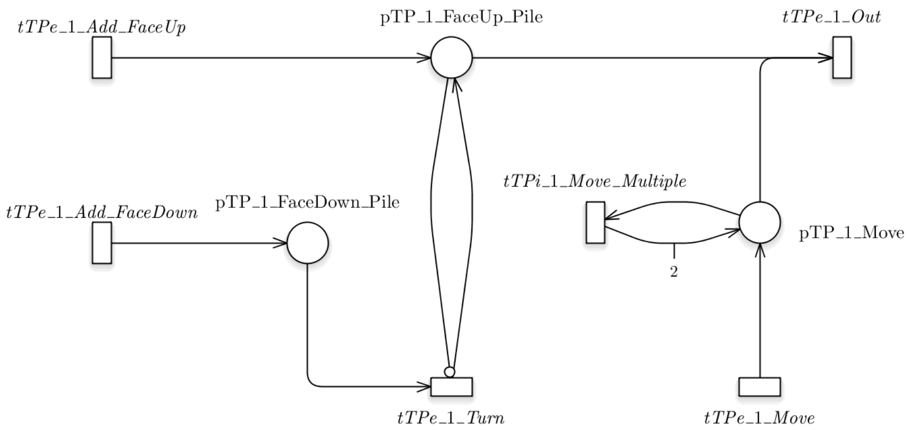

Fig. 5. Tableau Pile Module

Table 6. Places used in Tableau Piles

	Name	Description
22	pTP_1_FaceDown_Pile	
23	pTP_1_FaceUp_Pile	
24	pTP_1_Move	
25	pTP_2_FaceDown_Pile	
26	pTP_2_FaceUp_Pile	
27	pTP_2_Move	
28	pTP_3_FaceDown_Pile	
29	pTP_3_FaceUp_Pile	
30	pTP_3_Move	
31	pTP_4_FaceDown_Pile	
32	pTP_4_FaceUp_Pile	
33	pTP_4_Move	
34	pTP_5_FaceDown_Pile	
35	pTP_5_FaceUp_Pile	
36	pTP_5_Move	
37	pTP_6_FaceDown_Pile	
38	pTP_6_FaceUp_Pile	
39	pTP_6_Move	
40	pTP_7_FaceDown_Pile	
41	pTP_7_FaceUp_Pile	
42	pTP_7_Move	

2.6 Tableau Pile Module

2.7 Module Connector Module

2.8 Player Module

2.9 Player Bot Module

3 Implementation

3.1 GUI

3.2 Algorithms

3.2.1 Atomicity

3.3 Commands

3.3.1 Move Command

The move command is contains four parts; the command, destination, source and amount. Each part is concatenated together, with colon as a separator. An example of a move command would be: *Move:TP1:TP5:3*, which means *Move 3 cards from TP1 to TP5*. If amount is not given, it will assume one card to be moved.

In order to make sure that only valid commands are issued, the function `checkCommand_Move` has been developed.

Listing 1.8. checkCommand_Move.m

```
function [ doCommand, cmdDest, card, cmdSource ] = ...
    checkCommand_Move( command, destination, source, handle_err)

global global_info;
[moveCmd, card] = splitCommand(command);
cmdDest = moveCmd{2};
cmdSource = moveCmd{3};

doCommand = false;
if length(cmdDest) < 3,
    set_handle(handle_err,'String','INCOMPLETE_COMMAND');
    return;
elseif ~ismember(cmdDest, global_info.FP_TP_PILES),
    set_handle(handle_err,'String','INVALID_MOVE_COMMAND');
    return;
end

% Foundation Piles
if ismember(cmdDest, global_info.FP_PILES),
    if ~isempty(destination) && destination(1) ~= cmdDest(3),
        return;
    end;
    movedCard_split = strsplit(card,'_');
    moved_suit = movedCard_split(1);
    moved_rank = movedCard_split(2);
    if(~isfield(global_info.SUITS,cmdDest(3))),
        set_handle(handle_err,'String','INVALID_SUIT');
        return;
    end;
    if moved_suit{1} ~= cmdDest(3),
```

```
        set_handle(handle_err,'String','INVALID_LOCATION');
        return;
    end;
    global_suit = global_info.SUITS.(cmdDest(3));
    fp_Pile = strcat('pFP_',global_suit(1),'_Pile');
    if(iscell(fp_Pile)),
        fp_Pile = fp_Pile{1};
    end;
    dest_topCard_Id = tokenArrivedLate(fp_Pile,1);
    moved_rank_value = global_info.CARDVALUE_MAP(moved_rank{1});
    if dest_topCard_Id,
        dest_topCard_Color = get_color(fp_Pile,dest_topCard_Id);
        dest_topCard_split = strsplit(dest_topCard_Color{1},'_');
        dest_topCard_Rank = dest_topCard_split(2);
        diffRank = moved_rank_value - global_info.CARDVALUE_MAP(dest_topCard_Rank{1});
        if(diffRank ~= 1), % Added card must be 1 value higher than the current card.
            set_handle(handle_err,'String','INVALID_CARD_VALUE');
            return;
        end;
    elseif moved_rank_value ~= 1,
        set_handle(handle_err,'String','FIRST_CARD_MUST_BE_ACE');
        return;
    end;
elseif ismember(cmdDest, global_info.TP_PILES),
    tableau_dest = cmdDest(3);

    if ~isempty(destination) == 1 && destination(1) ~= tableau_dest,
        return;
    end;
    movedCard_split = strsplit(card,'_');
    moved_suit = movedCard_split(1);
    moved_rank = movedCard_split(2);

    tp_FU_Pile_Dest = strcat('pTP_',tableau_dest,'_FaceUp_Pile');

    % Can not add to tableau piles where face up is empty and there exist
    % cards in face down pile.
    if ~isempty(tokIDs(strcat('pTP_',tableau_dest,'_FaceDown_Pile'))) && ...
            isempty(tokIDs(tp_FU_Pile_Dest)),
        set_handle(handle_err,'String','FACE_DOWN_PILE_MUST_BE_EMPTY');
        return;
    end

    if(iscell(tp_FU_Pile_Dest)),
        tp_FU_Pile_Dest = tp_FU_Pile_Dest{1};
    end;

    % Do not check amount once the command has reached it's destination.
    if length(moveCmd) >= 4 && ~isempty(source),
        if ismember(moveCmd{3}, global_info.TP_PILES),
            tableau_src = moveCmd{3};
            tableau_src = tableau_src(3);
            tp_Pile_Src = strcat('pTP_',tableau_src,'_FaceUp_Pile');
            if(iscell(tp_Pile_Src)),
                tp_Pile_Src = tp_Pile_Src{1};
            end;
        else,
            set_handle(handle_err,'String','INVALID_MOVE_COMMAND');
            return;
        end
        amount = str2double(moveCmd{4});
        if amount > length(tokIDs(tp_Pile_Src)) || amount < 1,
            set_handle(handle_err,'String','INVALID_AMOUNT');
            return;
        end;
    end
    % Check against the latest (lowest) card at destination.
    dest_topCard_Id = tokenArrivedLate(tp_FU_Pile_Dest,1);
    moved_rank_value = global_info.CARDVALUE_MAP(moved_rank{1});
    if dest_topCard_Id,
        dest_topCard_Color = get_color(tp_FU_Pile_Dest,dest_topCard_Id);
        dest_topCard_split = strsplit(dest_topCard_Color{1},'_');
        dest_topCard_Suit = dest_topCard_split(1);
        dest_topCard_Rank = dest_topCard_split(2);

        moved_global_suit = global_info.SUITS.(moved_suit{1});
```

```
        dest_global_suit = global_info.SUITS.(dest_topCard_Suit{1});

        diffRank = moved_rank_value - global_info.CARDVALUE_MAP(
            dest_topCard_Rank{1});
        % Added card must be 1 value lower than the current card.
        if(diffRank ~= -1),
            set_handle(handle_err,'String','INVALID_CARD_VALUE');
            return;
        end;
        % Moved and current suit color must be different (red/black).
        if(strcmp(moved_global_suit{2},dest_global_suit{2})),
            set_handle(handle_err,'String','SUIT_COLOR_MUST_BE_ALTERNATING');
            return;
        end;
    elseif moved_rank_value ~= 13,
        set_handle(handle_err,'String','FIRST_CARD_MUST_BE_KING');
        return;
    end;
else,
    set_handle(handle_err,'String','INVALID_PILE');
    return
end;

if ~isempty(source),
    global_info.last_command_source = source;
end;

set_handle(handle_err,'String','');
doCommand = true;
```

3.4 Initial Dealing

3.5 Resources

3.6 Moving Multiple Cards

3.7 Scoring

3.8 Possible improvments

A major drawback of the siphon `tMC_Out_Buffer_Siphon` is that if it fires, the card will actually be removed from the game, and the game becomes unsolvable. This transition will fire if the move-command of the token has an invalid destination. Due to how the Player and Player Bot modules are set up, this will never happen as they will check the validity of the move command before actually issuing the command. Still, I think it would be an improvement add an additional transition to the Draw Pile module which would accept cards from `tMC_Out_Buffer_Siphon`, instead of totally discarding them.

Another improvement would be to refactor the codebase by moving more of the validity check of the commands from the Player and Player Bot modules to the destination transitions. The Player Bot modules uses roughly 200 lines of code to always issue valid commands, I think this could be drastically reduced. By doing this it would be easier to create additional modules which could interface with the game, for example a hardware-based module.

```
def mapper_from_to(self, key, email):
    if 'to' in email.keys() and 'from' in email.keys() and 'body_count' in email.key
```

4 Testing, Analysis and Results

4.1 Algorithms

4.1.1 Atomicity In order to preventdd

4.2 Initial Dealing

4.3 Resources

4.4 Moving Multiple Cards

5 Discussion

References

1. Wikipedia article on Tf-idf. `https://en.wikipedia.org/wiki/Tf?idf`
2. Tom White, Hadoop: The Definitive Guide, 2015, *ISBN: 978-1-491-90163-2*
3. Docker API Docs, `https://docs.docker.com`
4. Slides from DAT630, Krisztian Balog
5. Kaggle. The Enron Email Dataset. `https://www.kaggle.com/wcukierski/enron-email-dataset`
6. Data Intensive Systems Compendium, Tomasz Wiktorski et al.