

SoarBot: A Rule-Based System For Playing Poker

by

Robert I. Follek

Submitted in partial fulfillment
of the requirements for the degree of
Master of Science
in Computer Science

at

School of Computer Science and Information Systems

Pace University

May 2003

Thesis Signature (Approval) Page

I hereby certify that this thesis, submitted by Robert I. Follek, satisfies the thesis requirements for the degree of Master of Science and has been approved.

Dr. D. Paul Benjamin

Thesis Advisor

School of Computer Science and Information Systems

Pace University

Date

Abstract

SoarBot: A Rule-Based System For Playing Poker

Poker is a compelling topic for artificial intelligence research, different in many interesting ways from games like chess. In poker, multiple competing agents play a game of imperfect knowledge, where deception is legal and success or failure emerges only after thousands of hands.

In this thesis, I present SoarBot, a rule-based system for playing Texas Hold'em poker. SoarBot is an integration of the Soar rule-based system with the University of Alberta Computer Poker Research Group poker client framework. It uses a rule set of expert knowledge to make its betting decisions.

SoarBot competed in games against a mix of human and software players. It played much better than the worst human players, and much worse than the best human and software players. Overall, its play was mediocre.

Table Of Contents

<i>Abstract</i>	<i>iii</i>
<i>Table Of Contents</i>	<i>iv</i>
<i>List Of Tables</i>	<i>vii</i>
<i>List Of Abbreviations</i>	<i>viii</i>
1. Introduction	1
1.1 Why Poker?	1
General Application Problem	1
1.1.1 Imperfect Knowledge - Opponents' Hands Are Hidden	1
1.1.2 Multiple Competing Agents - Many Competing Players	2
1.1.3 Risk Management - Betting Strategies and Their Consequences	2
1.1.4 Agent Modeling - Identifying Patterns In Opponent's Play And Exploiting Them	3
1.1.5 Deception - Bluffing And Varying Style Of Play	3
1.1.6 Unreliable Information - Taking Into Account Your Opponents' Deceptive Plays	4
1.1.7 Conclusion: The Goal of Poker	4
1.2 Texas Hold'em	5
1.2.1 The Rules	5
1.2.2 The Betting	5
1.2.3 The Stages	6
1.2.4 The Button	6
1.2.5 The Blinds	6
1.2.6 Why Texas Hold'em?	7
1.3 The University of Alberta Computer Poker Research Group	8
1.3.1 The Online Poker Protocol	8
1.3.2 The CPRG Poker Client Framework	9
1.3.3 Quirks In The CPRG Framework	10
1.4 Poki, The CPRG Bot	11
1.4.1 Probabilistic Hand-strength and Hand-Potential Evaluations	11
1.4.2 Expert Systems	12
1.4.3 Simulations	12
1.4.4 Opponent Modeling	12
1.4.5 Interactions Between Technologies	13
1.5 Poker and Problems With Tree Search	13
1.6 Rule-Based Systems	14
1.6.1 Expert Systems	14
1.6.2 Rule-Based Systems	15
1.6.3 Rule-Based Systems vs. Conventional Programming Languages	16

1.6.4 Performance of Rule-Based Systems	17
1.7 Soar	18
1.7.1 Soar as a Unified Theory of Cognition	18
1.7.2 Soar as an Architecture for Intelligent Agents	19
1.7.3 Similarities Between Soar and a Typical Rule-Based System	20
1.7.4 Differences Between Soar and a Typical Rule-Based System	21
1.7.4.1 Operators	21
1.7.4.2 Soar's Execution Cycle	21
1.7.4.2.1 The Proposal Phase	22
1.7.4.2.2 The Decision Phase	23
1.7.4.2.3 The Application Phase	23
1.7.4.3 Impasses and Learning	24
1.7.5 Conclusion - Is Soar a Rule-Based System?	26
1.8 Software Integration Issues	27
1.8.1 Tcl	27
1.8.2 Soar and Tcl	28
1.8.2.1 The User Interface	28
1.8.2.2 The Input/Output Links	29
1.8.2.2.1 The Input/Output Links and Working Memory	29
1.8.2.2.2 The Input/Output Links and Tcl	30
1.8.3 The CPRG Poker Client Framework, Tcl, and Soar	31
1.8.3.1 Feather	31
1.8.3.2 SoarSession	32
1.8.3.3 Performance and Reliability	32
2. SoarBot	34
2.1 SoarBot.java	34
2.1.1 Key Concepts	34
2.1.1.1 Game State Data Structures	34
2.1.1.2 How SoarBot.java Moves Data to Tcl	34
2.1.1.3 Constructor	34
2.1.1.4 newGame() method	35
2.1.1.5 action() method	35
2.1.1.5.1 Check-Raise Bets	36
2.1.1.6 SoarBot.ini file	36
2.2 The Rule Set	36
2.2.1 Working Memory Structures	36
2.2.2 poker.soar	37
2.2.2.1 Subgoalng Based on the Stage of the Poker Hand	37
2.2.2.2 Selecting From Among Proposed Operators	38
2.2.2.3 Applying the Selected Operator	39
2.2.2.4 Removing the Selected Operator from the ^output-link	39
2.2.3 pokerIo.tcl	40
2.2.4 analysis.soar	40
2.2.5 defaultOps.soar	44

2.2.6 preflop_loose.soar	45
2.2.6.1 The ^maybe Attribute	47
2.2.7 flop.soar	48
2.2.7.1 Semibluffs	48
2.2.8 turn.soar	50
2.2.8.1 Pot Odds	51
2.2.9 river.soar	54
2.2.10 Log Files	56
2.2.11 Bluffing and Deception	57
3. Results	58
3.1 Mixed Human/Bot Games	58
3.1.1 Analysis	58
3.2 Bot-Only Games	60
3.2.1 Analysis	60
4. Conclusions	61
5. Opportunities for Further Development	62
5.1 Move Hand Analysis Out Of the Rule Set	62
5.2 Opponent Modeling	62
5.3 Learning	62
Appendix A: Downloading the Source Code	64
Bibliography	65

List Of Tables

Table 1-1	Characteristics of Intelligent Applications.....	1
Table 3-1	Results, Mixed Human/Bot Games.....	58
Table 3-2	Top Human Players.....	59
Table 3-3	Results, Bot-Only Games.....	60

List Of Abbreviations

API	Application Programming Interface
CPRG	University of Alberta Computer Poker Research Group
hold'em	Texas Hold'em
JNI	Java Native Interface
OPP	Online Poker Protocol

1. Introduction

1.1 Why Poker?

Poker is a compelling topic for artificial intelligence research, different in many interesting ways from games like chess. In their analysis of poker's suitability and challenges for research, Darse Billings and his co-authors from the University of Alberta Computer Poker Research Group (CPRG) list the characteristics of "applications that are perceived to require intelligent behaviour" (*Poker 1*), and show how these characteristics apply to poker:

General Application Problem	Problem Realization in Poker
imperfect knowledge	opponents' hands are hidden
multiple competing agents	many competing players
risk management	betting strategies and their consequences
agent modeling	identifying patterns in opponent's play and exploiting them
deception	bluffing and varying style of play
unreliable information	taking into account your opponents' deceptive plays

(*Poker 2*)

Billings et al note the correspondence between these characteristics and those found in "financial trading, business negotiations, and forecasting (from weather to politics)." (*Poker 1*). A software agent that can play world-class poker may be important beyond the world of the Las Vegas Strip.

1.1.1 Imperfect Knowledge - Opponents' Hands Are Hidden

Chess is a game of perfect knowledge or information, where "the world state is fully accessible...The computer representation of the game actually can be correct in every relevant detail - unlike the representation of fighting a war, for example." (Russell and Norvig 122). Both players in a game of chess have full and equal knowledge about the game pieces in play. Go and checkers are also games of perfect knowledge.

Poker is a game of imperfect knowledge, because each player has hidden cards in his hand that the other players can't see. The number of hidden cards varies depending on the specific poker variation. In five-card stud, each player has one hidden card. In Omaha, each player has four. In Texas Hold'em (hold'em), the game SoarBot plays, each player has two hidden cards. In all these games, the

player also uses a number of visible cards, in combination with his hidden cards, to build his hand. In draw poker, by contrast, each player's entire hand is hidden.

Imperfect knowledge means that a poker player maneuvers in an uncertain environment. He can rarely be *sure* that he's doing the right thing. Rather, he must think in terms of probabilities gleaned from mathematics and experience.

1.1.2 Multiple Competing Agents - Many Competing Players

Chess is a game for two players, but there are invariably several players in any worthwhile poker game. A typical casino hold'em game has as many as ten players. Collusion between players is illegal, so a player must simultaneously play against all the other players in the hand; to win the hand, he must beat all the other players. Poker is winner-take-all, and it's a poker truism that you can go broke finishing second.

The number of players in a hand also means that betting position can be critical. In hold'em, the dealer is the last to bet. He acts with the valuable knowledge of what all the other players betting before him chose to do.

1.1.3 Risk Management - Betting Strategies and Their Consequences

In a game of chess, each player wants to make the strongest possible moves. There's no reason to employ any other strategy. A sequence of strong moves is likely to win the game, and that's the goal. All wins have the same value, and there are no situations where a less-than-optimal move now may make strategic sense a hundred games later.

In poker, the notion of "strongest possible move" is a good deal subtler than simply betting the maximum when you have strong cards. A good poker player must have a betting strategy, a plan for "whether to fold, call, or raise in a given situation" (Schaeffer et al 2). The strongest move is the move that, over thousands of hands and a changing cast of opponents, wins the most money.

A good player may make bets that look odd out of context. For example, he may refrain from betting with very strong cards because he doesn't want to drive the other players out early. Or, with weak cards, he may bluff, hoping that he can scare the other players into folding better cards. Or, he may *bet* with strong cards, confident that the other players will stay in because they remember his earlier bluffs.

In poker, some wins have much more value than others, because some pots are much bigger than others. A good player adjusts his play to the size of the pot.

Given the same cards in two different hands, he may fold the first hand, but play the second, based purely on the amount of money in the middle of the table.

1.1.4 Agent Modeling - Identifying Patterns In Opponent's Play And Exploiting Them

In any game, a player has the opportunity to build historical knowledge of his opponents, a process known as opponent modeling. But a strong chess program can play very well without opponent modeling. It doesn't need to know whether it has played an opponent before, what openings the opponent prefers, or how the opponent reacts to time pressure. Only at the highest levels of competition do the programmers of chess software go through "extensive secret preparations" (Schaeffer).

A strong poker program depends on opponent modeling much more than a chess program does. A poker program can use historical data on betting patterns to try to deduce what an opponent's hidden cards are. A poker program can use historical data on an opponent's past bluffs to decide whether he's bluffing now. It can look for standard poker gambits, e.g. an opponent who always makes the first raise in late position in an attempt to "steal the ante".

1.1.5 Deception - Bluffing And Varying Style Of Play

According to Billings et al, "...the best risk management strategy in the world cannot compensate for a lack of deception, since human opponents are quick to exploit predictable players, no matter how strong they might otherwise be." (*Poker 2*) Correctly identifying a pattern goes a long way towards improving imperfect knowledge. If an opponent who plays only the best cards makes a bet when an ace shows up, you can fold your pair of kings without agonizing.

A good player must avoid falling into any patterns that the other players in the game can identify. Practical strategies for deception include bluffing - playing weak cards as if they're strong - and slowplaying very strong cards: "playing a hand weakly on one round of betting in order to suck people in for later bets" (Sklansky and Malmuth 49). Of course, there are deceptions within deceptions. A player with an established reputation as a bluffer will try to bet as if he's bluffing when he really has terrific cards.

Chess players certainly have styles of play - a player may be more or less aggressive, more or less comfortable in a defensive position, more or less likely to snatch a gambit pawn. But a discernable style in chess is of much less predictive value - the moves speak for themselves, with no guesswork involved. There's no hidden information to inadvertently betray, so there's little notion of deception.

1.1.6 Unreliable Information - Taking Into Account Your Opponents' Deceptive Plays

This is the flip side of deception. Each player is trying to play deceptively, and each player must consider the possibility that his opponents are playing deceptively as well. It is not unusual to run into a weak human player who both bluffs too frequently and folds too frequently, because he cannot conceive that anyone else in the game might ever bluff.

1.1.7 Conclusion: The Goal of Poker

Poker starts with the cards, and the cards are random. Take a sufficiently large sample and all players will get good cards and bad cards in roughly equal proportion. Why then does one player do any better in the long term than another?

It's not simply a matter of mathematics:

Knowing the mathematics of poker can certainly help you play a better game. However, mathematics is only a small part of poker logic, and while it is important, it is far less important than understanding and using the underlying concepts of poker. (Sklansky 5)

The long-term difference comes down to something very much like a winning investment strategy: good players win more with their good hands and lose less with their bad hands:

Claiming a pot when you have the best cards isn't enough to make you a winning player. Remember, there is no grand pay scale for holding the best hand. No one gives you a pile of money for drawing a royal straight flush. Some sucker has to pay you off. You have to maximize profits through guile and savvy, eke out every last dollar that your competition is willing to lose to you - and, when you don't have the winning cards, flee as fast as possible.

The key to minimizing losses when you have an inferior hand is recognizing the value of your cards relative to those of your competition. There is no predetermined fine for having a terrible hand. In fact, the stronger your losing cards are, the more money you are likely to part with. That's why the worst hand in poker is the second-best one at the table. (Bellin 20)

The goal of poker, then, is to maximize profits and minimize losses, hand after hand after hand, in a game world rife with complex problem characteristics. It's a tough goal, and a terrific challenge, for both players and programmers. A poker program that plays anywhere near the world-class level will be a stunning accomplishment. That's why poker.

1.2 Texas Hold'em

1.2.1 The Rules

Texas Hold'em is a seven-card poker game. Each player gets two hidden, or hole, cards. All the players also share the five board, or community, cards that are dealt face up in the middle of the table. Each player builds the best five-card hand from any combination of his two hole cards and the five board cards.

There's no obligation for a player to use his hole cards. For example, if the board is:

1. Ten of clubs
2. Jack of diamonds
3. Queen of spades
4. King of spades
5. Ace of clubs

then the board by itself forms an ace-high straight. No player in this hand will be able to make a better hand, no matter what his hole cards are, so all the players who stay in the hand until the end will split the pot.

1.2.2 The Betting

Each round of betting goes around the table in a circle, starting from the player to the dealer's left. In each round, the first player to act has a choice of betting or checking (not betting). This continues until someone bets. Once there's a bet, each player has a choice of folding (dropping out of the hand), calling (matching the bet), or raising (betting more). Most games have a rule limiting the number of raises in any round of betting to a maximum of three.

In fixed-limit hold'em, the game rules also fix the bet amount. Usually there's one amount for all bets in the early stages of a hand; it automatically doubles in the later stages. For example, a game of ten-twenty hold'em starts with \$10 bets and jumps to \$20. In pot-limit games, players can bet any amount up to the total

already in the pot. In no-limit games, players can bet all or any part of their money at any point in the hand.

1.2.3 The Stages

1. Preflop: At the start of a hold'em hand, each player gets two hole cards, and there's a round of betting. Typically, several players fold at this stage.
2. Flop: The dealer deals (flops) the first three board cards, and there's a round of betting.
3. Turn: The dealer deals the fourth board card, the bet amount doubles, and there's a round of betting.
4. River: The dealer deals the fifth and last board card, and there's a final round of betting.
5. Showdown: All active players (those who did not fold earlier in the hand) reveal their hole cards, and the dealer determines the winner. If there's a tie for best five-card hand, the players split the pot evenly - there's no recourse to a sixth card tiebreaker.

At any stage, if all the players but one fold, the dealer declares the last active player the winner of the hand and the hand is over. In this case, the winning player need not reveal his hole cards.

1.2.4 The Button

Poker is a positional game. It can be very helpful to see what the other players do - call, raise, or fold - before you make your decision. In stud poker, where each player has exposed cards, the player with the highest exposed cards makes the first bet. In hold'em, where the players all share the exposed cards, the player to the left of the dealer acts first. Since the deal rotates from player to player, the positional advantage works out evenly.

In casino poker, the casino provides the dealer, who is not a player in the game. But it's impractical for the dealer to keep changing seats as he deals for each player in turn. Instead, the dealer places a large plastic disc, the "button", in front of the player for whom he's currently dealing. The button moves around the table as the deal moves around the table.

1.2.5 The Blinds

In a typical home poker game, each player antes (puts a small fixed amount in the pot) before the start of each hand. This ensures that there's something in the pot.

A casino hold'em game starts differently. There's no ante, but there are two mandatory blind bets at the start of each hand, so called because the players who make the blind bets must do so before they see their hole cards.

The player to the dealer's left is called the small blind, because he must make the smaller blind bet. This bet is equal to half the bet in the game, e.g. \$5 in a ten-twenty game. The player to the small blind's left is the big blind. He must make a full bet, e.g. \$10 in a ten-twenty game.

Because these are bets, not antes, conventional betting in the preflop stage begins with the player to the big blind's left. And, because the big blind has already bet, no one else can check in the preflop stage. Each player must call the big blind's bet, raise it, or fold. (The small blind must at least call the half-bet difference between his half bet and the big blind's full bet.)

The blind betting structure gives players with poor cards a good reason to fold quickly, and ensures that players who want to stay in the hand will put some money in the pot. But, since each player will be the small blind and the big blind on a regular basis, as the deal rotates around the table, there's a steady, predictable cash drain. You have to risk some money and play your cards once in a while, or you'll go broke slowly. In fact, in hold'em tournaments, the size of the blind bets doubles periodically, so that the penalty for overly conservative play increases.

1.2.6 Why Texas Hold'em?

There are dozens of popular poker variations; a half-dozen different games show up regularly on the tables of casinos and cardrooms. The CPRG picked Texas Hold'em as their first variation to study because it "is considered to be the most strategically difficult poker variant that is widely played" (Billings et al, *Poker* 2), exhibiting all the characteristics described in sections 1.1.1-1.1.6 above. Hold'em is competitive poker's quintessence.

Many poker writers share this opinion. A. Alvarez notes that hold'em's "variations and subtleties are infinite" (36), and he quotes former world champion Johnny Moss's opinion that hold'em "is to stud and draw what chess is to checkers" (37). Sklansky and Malmuth call hold'em

probably the most complicated of poker games. The reason for this is that it is rare for it to be absolutely clear exactly how a hand should be played.

It is not uncommon to hear two expert players debate on the pros and cons of a certain strategy (1).

Texas Hold'em is also increasingly popular. The annual World Series of Poker, at Binion's Hotel and Casino in Las Vegas, uses hold'em for its final, championship round. The attendant publicity has spurred the growth of hold'em to the point where it "has overtaken 7-Stud as the most popular of the casino poker games" (Silberstang 79).

1.3 The University of Alberta Computer Poker Research Group

The University of Alberta Computer Poker Research Group (CPRG) is part of the University of Alberta GAMES Group. The GAMES Group "produces high-performance, real-time programs for strategic game-playing" ("What We Do" link). The CPRG has been active since 1995, and a succession of graduate students have contributed to the research.

As part of its ongoing efforts, the CPRG has developed a client-server poker system that runs games over the Internet. Their poker server deals Texas Hold'em, providing games where humans and software agents (the CPRG calls them bots) can play against each other for imaginary money.

1.3.1 The Online Poker Protocol

In a casino poker game, there are several players and one dealer. The dealer is not a player in the game. He deals the cards and manages the game, making sure that the players act in order, follow the betting rules, etc. When the hand is over, the dealer determines the winners and hands out the money.

This player-dealer structure maps well to a client-server architecture. Each client is a poker-playing agent (human or software), and the server is the dealer, running the game, with no financial interest in the outcome.

The CPRG has developed a protocol, the Online Poker Protocol (OPP), that defines a client-server poker system. The protocol "consists of simple messages being passed from server to client. The server hosts the game, the client acts as a player in a game" (par. 2).

The format of an OPP message is:

- message ID - integer;
- length of message data - integer;
- message data.

For example, the HOLE_CARD message has this format:

- integer - Position of the player whose hole cards these are;
- string - The hole cards, e.g. "Ah 5c" for ace of hearts, 5 of clubs.

The server uses this message to send each player his hole cards at the start of the hand, and, if the hand ends with a showdown, to send all players the hole cards revealed.

Currently, the only poker variant that the OPP supports is fixed-limit Texas Hold'em. The CPRG plans to extend the OPP to provide support for pot-limit and no-limit hold'em games in the future.

1.3.2 The CPRG Poker Client Framework

The CPRG has also developed and released C and Java versions of a framework for developing OPP poker clients. The framework handles low-level details like socket communications and OPP message parsing. It lets developers of poker bots focus on poker.

Both versions are full programs ready for extension - by editing a function in the C version, and by extending a class in the Java version. Pseudocode for the basic client program looks like:

```
connect to game
loop
    wait for message
    parse message
    update game data
    if it's this client's turn to bet
        call decision function/method
        send decision back to server
    end if
end loop
```

The client program maintains information on the state of the current game. In the C version, this game state information is in global variables. The Java version maintains the game state in classes like GameInfo and PlayerInfo.

I started my work with the C version of the framework because it was the simplest way to integrate the poker client with the other tools I was using. But as soon as I started working with the game state information, I realized that the C version was relatively primitive. For example, the C version stores the hole cards as a raw string straight from the socket message, e.g. "Ah 5c", while the Java version parses the card string into a Card class with getRank() and getSuit() methods. I checked with Aaron Davidson, one of the CPRG researchers, and he told me that

The C code may be a little out of date. I don't use it, and I only clean it up every now and then...I only use the Java stuff for my bots, so the java has undergone far more testing. (*Re: Poker Server*)

The Java version also has some useful javadocs, while the C version has no formal documentation - just comments in the code. To take advantage of its ongoing development, additional features, and greater reliability, I shifted my efforts to the Java version of the framework.

In the Java version, the mechanism for developing new poker clients is simple: extend the Player abstract class and implement the necessary methods. The Player class has five abstract methods, but there are only two methods that require more than an empty body: newGame() and action().

The framework invokes newGame() whenever a new game starts, passing in objects that store the new game context. The method is an opportunity for the client software to finish up any processing of the previous game, and to prepare for the new one. The client doesn't have to make any decision at this point - newGame() returns void.

When the framework invokes action(), the client has an opportunity to do its analysis and make a decision about which action to take. The CPRG provides sample code for some simple clients that do little or no analysis. For example, AlwaysCallPlayer is a working poker bot that always calls when it has to act. Its action() method is one line:

```
public int action() {
    return Holdem.CALL;
}
```

1.3.3 Quirks In The CPRG Framework

The AlwaysCallPlayer action() method works in all situations because the OPP doesn't distinguish a check from a call. In fact, the CALL and CHECK constants defined in Holdem.java have the same value:

```
public final static int CHECK = 1;
public final static int CALL = 1;
```

Check means you don't want to bet. You can only do this if no else has bet yet. Call means that another player made a bet and you're willing to match it. They're different, but there's no ambiguity, because the check and call actions are mutually exclusive. The interpretation is clear from the game state - if another

player made a bet, it's a call, else it's a check. The OPP takes the same fault-tolerant approach to bet and raise actions - they also share a message code.

The CPRG explains this message overloading: "A *check* and a *call* are logically equivalent, in that the betting level is not increased. The term *check* is used when the current betting level is zero, and *call* when there has been a wager in the current betting round" (Billings et al, *The Challenge* 5).

That's true as far as it goes. But there's a big difference between not having to add any money to the pot - a check - versus having to add as much as \$80 - a late-round call after a bet and three raises. A player will check even if he has terrible cards, because it doesn't cost him anything, but he'll call only if he thinks his winning chances are worth the money he must put in the pot.

I think this message overloading is a suboptimal design decision, because it may mask bugs in client software. I would prefer a protocol that fully distinguishes the actions and rejects invalid ones. In practice, though, it was easy enough to test the game state and make the distinctions in my own code. I didn't have to change the CPRG framework for this.

But I did run into another limitation that forced me to rework the framework a bit. The Player class does not provide a shutdown() method, a hook for the client to do any necessary cleanup. I added a shutdown() method to the CPRG framework by extending two classes:

- poker.PlayerEx.java extends poker.Player.java and adds an abstract shutdown() method;
- poker.online.BotPlayerEx.java extends poker.online.BotPlayer.java. It overrides BotPlayer's shutdown() method to first invoke PlayerEx.shutdown() before it invokes BotPlayer.shutdown().

And that did the trick. I was able to introduce new functionality by extending the existing CPRG classes instead of editing them - another advantage of the Java framework versus the C version.

1.4 Poki, The CPRG Bot

Poki is the CPRG's latest and greatest poker bot. It has evolved over time, incorporating the technologies the CPRG used and refined in their earlier bots with the innovations that new team members bring to the table. At this point, Poki does not rely on any one approach exclusively. Instead, it uses a combination of artificial intelligence and game-playing technologies.

1.4.1 Probabilistic Hand-strength and Hand-Potential Evaluations

Poki uses a set of mathematical evaluation functions that calculate probabilities and answer questions like:

- How good is a given hand compared to other hands?
- What is a hand's potential to improve?
- What is a hand's potential to worsen?

1.4.2 Expert Systems

Poki uses a set of expert rules, adapted from Sklansky and Malmuth, to make decisions for its play in the preflop stage. In later stages, Poki uses rules developed by Darse Billings, a CPRG member, Poki developer, and professional poker player.

1.4.3 Simulations

Poki does real-time simulations to estimate the expected value of potential betting actions. The simulations use a selective sampling technique based on "the assignment of probable hands to each opponent" (Billings et al, *The Challenge* 23).

1.4.4 Opponent Modeling

The CPRG points out that opponent modeling is important for maximizing wins:

Two opponents can make opposite kinds of errors - both can be exploited, but it requires a different response for each. For example, one opponent may bluff too much, the other too little. We adjust by calling more frequently against the former, and less frequently against the latter. To simply call with the optimal frequency would decline an opportunity for increased profit, which is how the game is scored (Billings et al, *The Challenge* 25).

Poki collects statistical data on each opponent's "personal history of actions" (Billings et al, *The Challenge* 27) and builds a table of likely opponent actions in different situations. The CPRG uses a neural network to analyze the data and guide Poki's statistical opponent modeling - the neural network identifies "new features to focus on when modeling common opponents" (Billings et al, *The Challenge* 29).

1.4.5 Interactions Between Technologies

I've presented the technologies that Poki uses as if they're separate pieces. In reality, they work together, with significant synergy. For example, accurate opponent modeling makes for accurate simulations; accurate evaluation functions provide accurate input data for the expert systems.

1.5 Poker and Problems With Tree Search

Poki's simulation is selective, not exhaustive, because "Enumerating all possible opponent hands and future community cards would be analogous to exhaustive game tree search, and is impractical for poker" (Billings et al, *The Challenge* 23).

In their game-theoretic analysis of imperfect knowledge games, including poker, Koller and Pfeffer confirm that

While we can now solve games with tens of thousands of nodes, we are nowhere close to being able to solve huge games such as full-scale poker, and it is unlikely that we will ever be able to do so. A game tree for five-card draw poker, for example, where players are allowed to exchange cards, has over 10^{25} different nodes...

Nevertheless, chess-playing programs are very successful in spite of the fact that we currently cannot solve full-scale chess. Can the standard game-playing techniques be applied to imperfect information games? We believe that, in principle, the answer is yes, but the issue is nontrivial (39).

They identify some of the difficulties that the imperfect knowledge causes. Tree pruning is much less effective than it is for a perfect knowledge game because

...one cannot eliminate nodes that the opponent considers possible, even if one knows for a fact that they are not. Similarly, nodes that neither player considers possible, but that one player thinks the other player considers possible, can also have an effect on a player's strategy. The only nodes that can be completely eliminated from consideration are those for which it is *common knowledge* that they cannot be reached. (40)

This difficulty is also a factor in the evaluation function:

This function must take into account the knowledge of the players in a state...For example, a naive evaluation function for poker might estimate that the player with the better hand will win the current stake, but this fails to differentiate between the states where a player knows she has the better hand, and one where she does not, and this knowledge affects the value of the state. (40)

Part of the problem is that the number of cards and players makes for a large branching factor:

Even if we were to circumvent all of these obstacles, the bounded-depth-search approach is not a general solution. While it may work when the game tree is deep but not too wide, some game trees (full scale poker, for example) cannot even be expanded to depth 1. (40-41).

Koller and Pfeffer see potential in an abstract game tree approach, where “Many similar game states are mapped to the same abstract state, resulting in an abstract game tree much smaller than the original tree” (41). For example, a player starting a hold'em hand with an ace of spades and a small card will play out most hands the same way whether his small card is the two of diamonds, the two of hearts, or the three of spades. As more researchers turn their efforts to imperfect knowledge games, it will be interesting to see if abstract game trees bear fruit.

1.6 Rule-Based Systems

1.6.1 Expert Systems

SoarBot, my poker program, is, at its core, a rule-based system. A rule-based system is a type of expert system, and an expert system

is a computer-based system that uses knowledge, facts, and reasoning techniques to solve problems that normally require the abilities of human experts. (Martin and Oxman 14).

Expert systems have been around for more than thirty years. Systems exist for dozens of subject domains, from medical diagnosis to computer hardware configuration to tax law.

1.6.2 Rule-Based Systems

Rule-based systems are also known as production rule systems and production systems. The terms are used interchangeably in the literature. Whatever the term, the essential characteristic is the same: such systems

```
encode the knowledge critical for decision making in
the form of individualized reasoning rules. Storage of
the knowledge is in the form of simple if-then or
condition-action rules (Martin and Oxman 6).
```

The rules are also known as productions. The if-clause conditions of a rule are sometimes called its left-hand-side. The then-clause actions are called its right-hand-side. When the rule interpreter executes the right-hand-side actions of a rule, it is said to fire, or trigger, the rule.

For example, here's a rule from MYCIN, a 1970's-era rule-based system for diagnosing blood infections:

```
IF stain is gram positive
AND morphology is coccus
AND growth is chains
THEN suggestive evidence that organism is streptococcus
(Addis 204)
```

MYCIN is written in Lisp, and the rules are stored internally as Lisp code. But MYCIN can translate its rules into English for display.

Rule-based systems have a basic three-part architecture:

1. The rule set - the knowledgebase of expertise in the form of if-then rules;
2. Working memory - a dynamic scratchpad region for data, goals, and intermediate results;
3. The rule interpreter - The engine that applies rules and modifies working memory (Jackson 31).

Typically, the rule interpreter uses forward chaining:

```
...inference rules are applied to the knowledge base,
yielding new assertions. This process repeats forever,
or until some stopping criterion is met (Russell and
Norvig 313).
```

A two-rule example of a new assertion in a hypothetical medical diagnosis system might look like this:

```
R1.
```

```
IF patient P exhibits symptoms X AND Y
THEN patient P suffers from disease type A
```

R2.

```
IF patient P suffers from disease type A
AND patient P exhibits symptom Z
THEN patient P suffers from disease type B
(Addis 204-5)
```

If the conditions for the first rule, R1, are satisfied, the rule interpreter will fire R1 and update working memory with the datum that patient P suffers from disease type A. Then, if working memory also shows that patient P exhibits symptom Z, the conditions for the second rule, R2, are satisfied. The rule interpreter will fire R2, updating working memory with the datum that patient P suffers from disease type B.

The rule interpreter runs in a simple loop. Each trip through the loop is called a cycle:

1. Match rule left-hand-sides against data in working memory;
2. If a rule matches, fire it. This may change working memory;
3. Go to step 1.

There may be more than one matching rule available to the rule interpreter at step 2. If so, the interpreter will use its “conflict resolution” mechanism. Some strategies for conflict resolution are:

- Don't allow a rule to fire more than once on the same data;
- Prefer rules that match on more recent data;
- Prefer rules that match on a greater number of conditions.

A rule interpreter may use a combination of these strategies (Jackson 34).

Left-hand-side conditions can use both constants and variables, with the usual equality/inequality operators. Variable declaration is automatic. Variables created in a rule's left-hand-side conditions are available in the rule's right-hand-side actions. The right-hand-side actions let rules add, change, or delete data in working memory. Actions may also call external functions, letting rules interact with external systems. The rule interpreter may also define special areas of working memory that act as automatic input and output links to external systems.

1.6.3 Rule-Based Systems vs. Conventional Programming Languages

Because the rule interpreter provides the rule-matching loop and conflict resolution strategies, the rule language can be much simpler than a conventional programming language: no variable declarations, no loops, no subroutines - just

condition-action clauses. And, unlike statements in a conventional programming language, the rules in a rule set can appear in any order.

This simplicity makes it easy to add or remove rules. Consider the two-rule medical diagnosis example above. Adding a new rule is as easy as:

```
IF patient P exhibits symptoms M AND N
THEN patient P suffers from disease type B
```

This rule provides a new way for the rule-based system to diagnose disease type B. It doesn't have any connection to the two original rules; either of those rules could vanish without affecting this rule. Because each rule represents an independent, discrete piece of knowledge, changes can "be made incrementally without the necessity of understanding the entire control structure." (Martin and Oxman 6). In a rule-based system with hundreds, or even thousands, of rules, this is invaluable.

Some rule-based system project teams include a knowledge engineer who acts "as an intermediary between the knowledge-base and its author" (Martin and Oxman 3). The knowledge engineer interviews the expert and writes the rules.

Because the rule language is simple, the knowledge engineer does not need the full skill set of a programmer, and he can focus on the representation of expertise as a rule set:

```
Knowledge represented as if-then rules is easily
understandable. Most people are comfortable reading
rules, in contrast to knowledge represented in
predicate logic (Bigus and Bigus 83).
```

There is also evidence that the simple rules of rule-based systems correspond well with the way experts work:

```
The process of rendering the knowledge explicit in a
piecemeal fashion seemed to be more in tune with the
way that experts store and apply their knowledge. In
response to requests as to how they do their job, few
experts will provide a well-articulated sequence of
steps that is guaranteed to terminate with success in
all situations (Jackson 8-9).
```

In other words, experts don't think like programmers; a language to capture their expertise need not resemble a programming language.

1.6.4 Performance of Rule-Based Systems

A rule interpreter does a great deal of pattern matching between rule left-hand-sides and working memory. This “many-pattern/many-object” matching can be computationally expensive: “...early production systems spend over 90 per cent of their time doing matching of this kind” (Jackson 128).

In 1982, Charles Forgy published the rete (rhymes with “treaty”) algorithm. The algorithm takes advantage of two characteristics of rule sets:

- Rule left-hand-sides often share conditions;
- There are relatively few working memory changes in one cycle.

The algorithm first compiles the rule set into a network (“rete” is Latin for “net”). This eliminates rule duplication, and records initial matches between left-hand-sides and working memory. Then, the algorithm updates the network as working memory changes. The result is fast, efficient pattern-matching. Rather than retest all rule left-hand-sides in each cycle, the algorithm retests only those rules whose left-hand-sides are dependent on the previous cycle’s working memory changes.

The rete algorithm has emerged as the classic algorithm for rule-based systems. The venerable OPS5 system was the first rule-based system to use rete. Second-generation systems like Clips and Soar both use it, and more recent rule-based systems like Jess and Lisa continue the tradition.

1.7 Soar

Soar is the rule-based system behind SoarBot. But Soar has goals and capabilities beyond those of a typical rule-based system. Soar’s developers describe Soar as “a theory, implemented as a software architecture, that seeks to describe and realize the fundamental, functional components of intelligence” (Soar Technology, *Soar: A Functional Approach* 1).

1.7.1 Soar as a Unified Theory of Cognition

Soar’s developers began their work on Soar in an attempt to define a unified theory of cognition, to reconcile and unify the “microtheories” (Lehman, Laird, and Rosenbloom 2) of individual disciplines within the field of cognitive science. From this perspective, Soar is a theory - their candidate unified theory of cognition.

The Soar cognitive theory says that cognitive behavior

- is goal oriented;
- reflects a rich, complex, detailed environment;
- requires a large amount of knowledge;

- requires the use of symbols and abstractions;
- is flexible, and a function of the environment;
- requires learning from the environment and experience (Lehman, Laird, and Rosenbloom 5-6).

The Soar developers explain that, because of their backgrounds “in both artificial intelligence and psychology”, they chose to state their theory

both narratively and as a working computer program. So, for us at least, working on a unified theory of cognition means trying to find a set of computationally-realizable mechanisms and structures that can answer all the questions we might want to ask about cognitive behavior (Lehman, Laird, and Rosenbloom 2-3).

The “working computer program” is Soar, and the intentions behind it are plainly more ambitious than a typical rule-based system.

1.7.2 Soar as an Architecture for Intelligent Agents

Soar is also an architecture for intelligent agents, where an agent is “anything that can be viewed as **perceiving** its environment through **sensors** and **acting** upon that environment through **effectors**” (Russell and Norvig 31), and its intelligence is “the ability of the agent to capture and apply application domain-specific knowledge and processing to solve problems” (Bigus and Bigus 10).

Soar’s developers explain that their goal “is for Soar to support all the capabilities required of a general intelligent agent,” and they describe Soar as

an architecture that can...be used to build systems that work on the full range of tasks expected of an intelligent agent, from highly routine to extremely difficult, open-ended problems (Laird, Congdon, and Coulter 1)...

This intelligent agents view of Soar is not at all at odds with the cognitive theory view of Soar - rather, it’s a rephrasing of the same ideas. Both views grow out of the definition of Soar as “an architecture for constructing general intelligent systems” (Laird, Congdon, and Coulter 1). Support for intelligent agents is explicit in both the software and its user community:

- The first window that appears in the interactive Soar environment is the Agent Control Panel;
- Soar researchers often refer to their rule sets as agents.

1.7.3 Similarities Between Soar and a Typical Rule-Based System

Notwithstanding the cognitive architecture and intelligent agents views of Soar, there are striking similarities between Soar and a typical rule-based system. Soar follows the basic three-part architecture of a rule-based system: rule set, working memory, and rule interpreter. In the intelligent agents view, the rule set defines an agent. In the cognitive architecture view, the rule set represents long-term knowledge, and working memory represents short-term knowledge of the current situation.

Whatever the point of view, the rules in a Soar rule set strongly resemble the rules in many rule-based systems. For example, here are Soar versions of the medical diagnosis rules R1 and R2 from section 1.6.2 above:

```
#pound sign starts a comment.
#R1.
#IF patient P exhibits symptoms X AND Y
#THEN patient P suffers from disease type A

sp {diagnose*A
    (state <s> ^patient <p>)
    (<p> ^symptom X
      ^symptom Y)
-->
    (<p> ^disease A)
}

#R2.
#IF patient P suffers from disease type A
#AND patient P exhibits symptom Z
#THEN patient P suffers from disease type B

sp {diagnose*B
    (state <s> ^patient <p>)
    (<p> ^disease A
      ^symptom Z)
-->
    (<p> ^disease B)
}
```

Without going into the details of Soar's rule language, it's worth noting some of the syntax features in these rules:

1. Ordinary strings are constants;
2. Strings that start with ^carats are attributes;
3. Strings in <angle brackets> are variables;

4. The --> arrow separates the left-hand-side conditions from the right-hand-side actions.

These same syntax features show up in the OPS5 rule-based system written in the late 1970's; anyone familiar with OPS5 would have little trouble reading Soar rules. Soar also uses the terminology of rule-based systems: the "sp" that starts each rule is an abbreviation for "Soar production", and the Soar documentation refers to rules and productions.

1.7.4 Differences Between Soar and a Typical Rule-Based System

1.7.4.1 Operators

Soar's architecture includes the concept of an operator, an "abstract type" (Soar Technology, *Soar: A Comparison* 1) that "transforms the current state by changing some of its features and values" (Lehman, Laird, and Rosenbloom 12). Operators are special attributes of the problem state that "perform actions, either in the world or internally..." (Laird 20).

Operators are the decisions that Soar makes based on its rule firings. They answer the question, "What should I do next?" Operators may trigger an action in, or send a message to, external software. For example, in SoarBot, the operators correspond to betting options that the CPRG poker framework recognizes, like fold or call. For games or puzzles represented completely within Soar, operators will make the next move: in a Soar solution to the 8-queens puzzle, the operators move queens around the board.

1.7.4.2 Soar's Execution Cycle

As per section 1.6.2 above, a typical rule-based system has a rule interpreter that runs in a simple loop of cycles. On each cycle, it looks for a rule whose left-hand-side conditions match working memory, and fires the matching rule. If more than one rule matches, the rule interpreter uses its conflict resolution algorithm to select one rule to fire. In this approach, one cycle equals one rule fired. The medical diagnosis example in section 1.6.2 above would take two cycles to diagnose disease B.

In contrast, Soar's execution cycle has three phases, built around its handling of operators:

1. Proposal
2. Decision
3. Application

1.7.4.2.1 The Proposal Phase

In the proposal phase, Soar matches and fires as many rules as possible. And, because firing one rule may change working memory so that the left-hand-side conditions of another rule now match, Soar continues to match and fire rules until the rule set reaches quiescence.

Conceptually, the rules fire simultaneously, so there are no timing issues around which rules fire first. Soar keeps working memory consistent: if a later rule fires and changes working memory in a way that invalidates the left-hand-side conditions of a rule that fired earlier, Soar automatically retracts the firing of the earlier rule. (A common Soar newbie mistake is a rule whose right-hand-side invalidates the match of its own left-hand-side. The rule loops: it fires because the left-hand-side matches, then retracts because the right-hand-side changes working memory so that the left-hand-side no longer matches, then fires again because the left-hand-side matches again after the right-hand side working memory changes are retracted, then retracts again because the left-hand-side no longer matches...)

Because Soar fires all possible rules, there's no need for the conflict resolution that a typical rule-based system performs. Given the medical diagnosis rule set in section 1.6.2 above, Soar would match and fire both rules in one cycle's proposal phase.

In Soar terms, the rules that match and fire during the proposal phase fall into two categories:

1. Elaborations - Rules that make changes in working memory, but don't propose operators;
2. Proposals - Rules that propose operators.

Syntactically, proposals look much like other rules, with the addition of the special `^operator` attribute:

```
sp {propose*hello-world
    (state <s> ^type state)
-->
    (<s> ^operator <o> +)
    (<o> ^name hello-world)
}
# (Laird 21)
```

Because Soar fires all possible rules in this phase, multiple operator proposals may fire. This is okay. Proposal rules can assign preferences to the operators

they propose; Soar uses these preferences to choose among the multiple operators in its decision phase.

1.7.4.2.2 The Decision Phase

Soar applies one operator per execution cycle. In the decision phase of the cycle, Soar selects the one operator to apply. It considers all the currently proposed operators, and bases its selection on the preferences attached to the operators when they were proposed.

Preferences can be binary, e.g. operator A is preferable to operator B, or unary, e.g. operator M is best. Preferences may also be expressed through separate rules in the rule set.

1.7.4.2.3 The Application Phase

In the application phase, Soar applies the operator that it selected in the decision phase. To do this, Soar adds an `^operator` attribute to the problem state. Then, it looks for and fires any rules whose left-hand-sides now match because of the new `^operator` attribute. For example, this rule matches if and only if the decision phase selects an operator named `hello-world`:

```
sp {apply*hello-world
    (state <s> ^operator <o>)
    (<o> ^name hello-world)
-->
    (write |Hello world|)
    (halt)
}
```

The actual application of the `hello-world` operator happens when Soar fires this rule's right-hand-side actions. These actions may be changes to working memory, commands relayed to external systems, or both. Changes to working memory may mean that some elaborations are now ready to fire because their left-hand-side conditions are all met. As it did in the proposal phase, Soar will match and fire all possible elaboration rules until the rule set reaches quiescence. Then Soar returns to the proposal phase and begins the next execution cycle.

It's worth emphasizing that the operator proposal and application phases are just special cases of the general rule-based system approach: match left-hand-side conditions, perform right-hand-side actions. Soar has no intrinsic knowledge of how to apply the selected operator - it depends on rules provided by the knowledge engineer or rule programmer for that.

1.7.4.3 Impasses and Learning

When the Soar execution cycle is in the decision phase, it may not be able to make a decision about which of the proposed operators it should select. For example,

...it is possible that the preferences are either incomplete or inconsistent. The preferences can be incomplete in that no **acceptable** operators are suggested, or that there are insufficient preferences to distinguish among **acceptable** operators. The preferences can be inconsistent if, for instance, operator A is preferred to operator B, and operator B is preferred to operator A. Since preferences are generated independently, from different production instantiations, there is no guarantee that they will be inconsistent. (Laird, Congdon, and Coulter 24).

In Soar terms, this is an impasse. Soar recognizes several impasse types. The circumstances vary, but the essence is that, when Soar reaches an impasse, it does not know what to do next. "An impasse is what happens when the decision cycle can't decide" (Lehman, Laird, and Rosenbloom 23).

When Soar hits an impasse, it subgoals: it automatically creates

a new state in which the goal of the problem solving is to resolve the impasse. Thus, in the substate, operators will be selected and applied in an attempt either to discover which o[f] the tied operators should be selected, or to apply the selected operator piece by piece. The substate is often called a subgoal because it exists to resolve the impasse, but is sometimes called a substate because the representation of the subgoal in Soar is as a state. (Laird, Congdon, and Coulter 27).

So, in the course of solving a problem, Soar may create a hierarchy of superstates and substates in working memory. When Soar creates a substate, it begins its execution cycle in the substate. The idea is that the proposal/decision/application phases in the substate will resolve the impasse in the superstate.

Soar provides little more than a naked substate. As with operator proposal and application, it's up to the knowledge engineer/rule programmer to write elaboration rules that give the substate a name and copy relevant attributes

down from the superstate to the substate. But this is not difficult - there are standard, well-documented Soar rule idioms for substate elaboration.

This multiplicity of states is why all Soar rules “must match against a state in working memory” (Laird, Congdon, and Coulter 44), and why rule left-hand-sides always begin with a state <s> condition. The top state can always be distinguished because it has no ^superstate attribute:

```
(state <s> ^superstate nil)
```

The knowledge engineer will generally also use elaboration rules to give the top state a name descriptive of the problem as a whole, and the substates names descriptive of their roles.

One advantage of this state/substate approach is that it provides “a way of partitioning knowledge and limiting the operators to be considered in searching for a goal’s desired state” (Lehman, Laird, and Rosenbloom 23). For example, I separated SoarBot’s rules into states based on the stages of a poker hand. At each stage of the hand, the top state proposes an operator named for the stage, i.e. “preflop”, “flop”, “turn”, or “river”, without an operator preference value. This means that Soar can’t pick an operator, so it subgoals and creates a substate. I have a rule that gives the substate the name of the operator that triggered it, e.g. a top-state operator named “preflop” triggers a substate named “preflop”. Then, rules specific to the substate fire, proposing operators with preferences. This resolves the impasse and Soar makes a betting decision.

Soar’s subgoaling mechanism also makes it possible for Soar to learn:

When an operator impasse is resolved, it means that Soar has, through problem solving, gained access to knowledge that was not readily available before. Therefore, when an impasse is resolved, Soar has an opportunity to learn, by summarizing and generalizing the processing in the substate.

Soar’s learning mechanism is called chunking; it attempts to create a new production called a chunk. The conditions of the chunk are the elements of the state that (through some chain of production firings) allowed the impasse to be resolved; the action of the production is the working memory element or preference that resolved the impasse (the result of the impasse). The conditions and action are variablized so that this new production may match in a similar situation in the future and prevent an impasse from arising. (Laird, Congdon, and Coulter 33)

In other words, when Soar resolves an impasse, it can remember the left-hand-side conditions that it matched and the right-hand-side actions that it took, through however many levels of subgoaling were necessary. In effect, it can add a new rule to its rule set. Going forward, Soar will not have to subgoal in the same situation again, because it has learned what to do.

This ability to learn new rules is what most strongly distinguishes Soar from a typical rule-based system. The chunking learning mechanism can “speed up behavior by compiling many steps through many subspaces into a single step in the pre-impasse problem space” (Lehman, Laird, and Rosenbloom 35). In problem spaces where Soar solves the problem by search, it typically subgoals deeply, and chunking can yield a visible performance improvement. Intriguingly, the body of Soar research includes sophisticated Soar applications whose learning rates mirror human learning rates in the same problem spaces (Lehman, Laird, and Rosenbloom 36).

1.7.5 Conclusion - Is Soar a Rule-Based System?

My answer is yes, but it's worth noting that this is a touchy question within the Soar community. Soar Technology has even issued a brief white paper that addresses the question, delineating the differences. Their inconclusive conclusion is that

it might be appropriate simply to say that Soar is *not* a rule-based system. The alternative argument (historically made by Soar researchers) is that rule-based representations can be used in much more flexible ways than they typically have been (Soar: A Comparison 2).

Ultimately, it doesn't much matter how we classify Soar. Clearly, it looks and acts like a rule-based system in many ways. Also clearly, it has goals and capabilities beyond the scope of most rule-based systems.

In practice, we answer the question when we use Soar. We can use it as a conventional rule-based system, or as a tool to explore machine learning. This decision may in turn depend on how readily we can use learning in a problem space. It can be difficult, because the speed-up that chunking provides is not always useful.

For example, with SoarBot, I used Soar as a powerful rule-based system with bells and whistles that helped me manage the rule set. But I didn't use Soar's learning features. The speed-up from chunking was insignificant, because SoarBot has no search, just a simple two-stage problem space. Its speed without chunking was fine.

Using chunking effectively may involve “building other styles of learning as content theories that rest on top of the architecture” (Lehman, Laird, and Rosenbloom 31), developing metarules that work together with chunking to build new rules for the problem space. I did not attempt this with SoarBot.

1.8 Software Integration Issues

SoarBot uses Soar version 8.3. In version 8.3, the Soar kernel is written in C and the external interface is written in Tcl 8.0. SoarBot also uses the Java version of the CPRG poker framework. Integrating Soar and Tcl with the Java framework presented some challenges.

1.8.1 Tcl

Tcl is an open-source interpreted scripting language, similar to Perl and Python. Such scripting languages typically:

1. Have no separate compilation step;
2. Use dynamic typing for variables instead of static typing;
3. Include high-level data structures like vectors and hash tables as built-in language elements;
4. Provide an interactive interpreter.

Tcl works well as a glue language, integrating readily with other applications and tools. In fact, its name is an acronym for “Tool Command Language” (Welch xiv). Standard Tcl releases include Tk, a cross-platform GUI toolkit.

Tcl provides two interpreter executables:

- tclsh, a command line;
- wish, a GUI built with Tk.

Both tclsh and wish provide a read-eval-print loop:

- The user enters a command;
- The Tcl interpreter evaluates the command;
- The interpreter prints the result of the evaluation.

Both tclsh and wish support Tcl’s two different approaches to integration with a foreign application:

1. Extend Tcl so that the foreign application’s API functions are available as Tcl commands, or

2. Embed a Tcl interpreter within the foreign application, so that it can execute Tcl scripts.

The extension approach adds new commands to Tcl. The new commands are wrappers around the foreign application's C API functions. The foreign application's API functions are usually packaged as a shared dynamic library that's automatically loaded as part of the Tcl shell environment. Tcl still provides its basic command line and GUI shells, with the new commands available. This approach is a good fit when the goal is to provide scripting capabilities for the functions of an existing API. It's the approach that Soar uses to expose its C kernel.

The embedding approach is essentially the opposite of the extension approach. Instead of adding the foreign application to Tcl, it adds Tcl to the foreign application. The Tcl API includes functions to create a Tcl interpreter, send it commands, and retrieve the results. With this approach, there's no Tcl command line; the foreign application is in charge. This approach works when the foreign application is a full-blown program rather than a library of API functions.

1.8.2 Soar and Tcl

1.8.2.1 The User Interface

The specific command used to start Soar varies from platform to platform, but the command's effects are the same. It:

- Runs wish, Tcl's GUI interpreter;
- Has wish load and execute a Tcl script, start-soar.tcl.

The result is a GUI interpreter for Soar. The Tcl wish executable provides the GUI Tcl interpreter, and start-soar.tcl extends it with the Soar command set. start-soar.tcl also adds Soar-specific features to the GUI.

The process is seamless enough that a Soar user may never realize, and need never think about the fact, that Soar is running on top of Tcl/Tk. The GUI provides menus for the common Soar commands as well as a command line that accepts all Soar and native Tcl commands.

For example, a Soar user might enter the command

```
source foo.soar
```

to load the foo agent into the Soar environment, where foo.soar is a text file that contains a Soar rule set. But **source** is actually a native Tcl command.

After loading the file, the Soar user might enter the

```
run
```

command to execute the foo agent. This is a Soar command that Soar added to Tcl, not a native Tcl command. The wish executable without start-soar.tcl loaded returns an “invalid command name” error if you ask it to evaluate **run**. But, from the Soar user’s point of view, the difference between the two commands is invisible.

1.8.2.2 *The Input/Output Links*

In section 1.7.4.2, I described the phases of Soar’s execution cycle as:

1. Proposal
2. Decision
3. Application

A more complete description looks like:

1. Input
2. Proposal
3. Decision
4. Application
5. Output (Laird, Congdon, and Coulter 23-24)

The input and output phases are Soar’s links to the outside world. In the input phase, “New sensory data comes into working memory.” In the output phase, “Output commands are sent to the external environment” (Laird, Congdon, and Coulter 23-24).

1.8.2.2.1 The Input/Output Links and Working Memory

In working memory, every top state starts with an `^io` attribute, and the `^io` attribute in turn has `^input-link` and `^output-link` attributes. Soar automatically populates the `^input-link` with attributes that store the results of the input phase; in the output phase, Soar automatically propagates the attributes of the `^output-link` out to the world. The particular `^input-link` and `^output-link` attributes, both names and values, depend on the specifics of the external applications involved.

For the knowledge engineer writing rules, the input link and output link are much like any other working memory data. Rule left-hand-sides can match against the `^input-link`. For example, this operator proposal rule is from a Soar agent written

for Eaters, a Pacman-like video game (Laird 5). The agent matches nearby cells to see if they contain food.

```
sp {propose*move-to-food
    (state <s> ^io.input-link.my-location.<dir>.content
        << normalfood bonusfood >>)
-->
    (<s> ^operator <o> + =)
    (<o> ^name move-to-food
        ^direction <dir>)
} # (move-to-food.soar)
```

Rule right-hand-sides can add data to the ^output-link. This Eaters operator application rule adds a ^move attribute to the ^output link, and a ^direction attribute, with the value “north”, to the ^move attribute.

```
sp {apply*move-north
    (state <s> ^operator <o>
        ^io <io>)
    (<io> ^output-link <ol>)
    (<o> ^name move-north)
-->
    (<ol> ^move <move>)
    (<move> ^direction north)
} # (move-north.soar)
```

1.8.2.2.2 The Input/Output Links and Tcl

Soar provides hooks for custom Tcl input and output procedures; Soar automatically runs these procedures during its input and output phases. The programmer integrating Soar with an external application can use these procedures to connect to the external application, and to move data into and out of working memory. SoarBot’s input procedure is iProc(); the output procedure is oProc(). Both procedures are in the pokerlo.tcl file.

Soar includes some commands that help with the esoterica of working memory updates, e.g.

- add-wme - Add a working memory element;
- remove-wme - Remove a working memory element.

Each working memory element has a unique timetag; add-wme returns the timetag of the element it adds. The input/output procedures can save timetags in Tcl global variables, so that working memory elements are accessible for later update or deletion. The syntactic specifics are somewhat tortured, and the

documentation (Congdon and Schwamb, and the `soar-io.tcl` file in the Soar demos directory) is sketchy. Here's a sample:

```
# Add the clock wme to the input link. Save the timetag of
# the clock wme so we can update during normal input cycles.
scan [add-wme $g_inputLinkId ^clock $g_clock] "%d"
    g_timeTags(clock) # (pokerlo.tcl)
```

This code puts a `^clock` attribute under the `^input-link` attribute in working memory. The new `^clock` attribute gets the value of the `g_clock` Tcl global variable, and its timetag is saved in `g_timeTags(clock)`, a Tcl global hash table.

Tcl global variables are also a convenient way for the input/output procedures and the external application to exchange data. Hash tables are helpful to keep the number of global variables under control. For example, in `SoarBot`, a single global hash named `"g_gameInfo"` stores several items of data, via its keys:

```
g_gameInfo(numActivePlayers)
g_gameInfo(numBets)
g_gameInfo(pot)
// etc.(SoarBot.java)
```

1.8.3 The CPRG Poker Client Framework, Tcl, and Soar

As I discussed in section 1.3.2 above, `SoarBot` uses the Java version of the CPRG's poker client framework. But `Soar` runs on top of Tcl. So, in order to use `Soar` with the Java framework, I had to find a way to use the framework with Tcl.

As I discussed in section 1.8.1 above, Tcl offers two approaches to integration: extension or embedding. Because the Java framework is a complete, ready-to-run program, the first thing I looked for was a way to embed Tcl within it.

1.8.3.1 Feather

I found exactly what I needed: Feather is Alden Dima's "...package that allows a Java application to embed native Tcl interpreters within the same process as the Java virtual machine" (par. 1).

Feather is a thin Java Native Interface (JNI) wrapper around the Tcl C API functions for creating and using a Tcl interpreter, e.g. `Tcl_CreateInterp()`, `Tcl_Eval()` (Tcl Library Manual). Feather's C component is fewer than 200 lines of code, the Java component fewer than 100 lines.

Feather makes a Tcl interpreter available as an object in a Java program. Once you've instantiated a Tcl interpreter object, you can invoke its methods to

evaluate a Tcl command, or file of commands, and retrieve the results as a Java String:

```
TclInterpreter interp = new TclInterpreter();
try {
    // sourcing a tcl script
    result = interp.evalFile(new File("test.tcl"));
    // calling an individual tcl command
    result = interp.eval("set a 1");
}
catch (TclEvalException e) {...}
// (Feather 0.1 Homepage)
```

I cobbled together a Linux build for Feather, did some simple tests, and it worked fine.

1.8.3.2 SoarSession

With Feather in place, I had Java and Tcl working together. The final piece was Soar.

To get Soar to run via Java and Feather, I had to dive into Soar's Tcl initialization scripts and make some changes. I ended up writing a wrapper class, `SoarSession.java`, that instantiates a Feather object and handles the Soar-specific initialization details. To simplify distribution and reuse, `SoarSession` stores the necessary Soar initialization scripts as string constants. It feeds them to Tcl via Feather.

`SoarSession` also adds a layer of Soar-friendly methods. For example, to start Soar, load a simple agent, and run it, all you have to do is:

```
SoarSession ss = new SoarSession(tclDir, soarDir);
ss.loadAgent(agentFile);
ss.run();
ss.stop();
```

1.8.3.3 Performance and Reliability

I had some concerns about how well all the pieces would hang together in practice:

- Feather is a 0.1 release, with no updates since October 1999;
- With two virtual machines (Tcl and Java) in the mix, I was leery of speed problems;

- The Java <=> Feather <=> SoarSession <=> Tcl <=> Soar combination has a lot of moving parts, and these polygamous software marriages are sometimes turbulent.

To test it all, I wrote a minimal Java program that used SoarSession to run a minimal Soar poker agent. The poker agent always called the current bet, a Soar version of AlwaysCallPlayer in section 1.3.2. I put everything together and turned it loose on the CPRG poker server.

The test agent played a few hundred games over a few hours, thoroughly exercising the full server-to-client-to-server round-trip, up and down all the software layers on the client side. Everything went smoothly. The combination was robust, and speed was not a problem. It was time to write a poker agent.

2. SoarBot

2.1 SoarBot.java

SoarBot is my poker agent; SoarBot.java is its Java layer. SoarBot.java uses the CPRG client framework by extending the generic PlayerEx abstract class (§ 1.3.3).

2.1.1 Key Concepts

2.1.1.1 Game State Data Structures

SoarBot.java gets game state information from two of the CPRG framework classes:

- GameInfo - “Stores all of the info defining a single game of poker.”
- PlayerInfo - “Stores all of the information for a player during a poker game” (Davidson, *Generated Documentation*).

The two classes work together; the GameInfo class has methods that return PlayerInfo objects.

2.1.1.2 How SoarBot.java Moves Data to Tcl

SoarBot.java moves data to the Tcl layer by setting TCL global variables. To do this, it prepares strings that are valid TCL commands, e.g.:

```
set g_gameInfo(position) 5;
```

is a Tcl command that sets the “position” slot in the g_gameInfo global hash table to the value 5.

Then SoarBot.java invokes SoarSession’s eval() method to execute the Tcl command string, propagating the global to the Tcl layer.

2.1.1.3 Constructor

This is where SoarBot.java starts Soar. SoarBot.java instantiates a SoarSession object; the SoarSession object creates a Tcl interpreter and runs the necessary

Soar initialization scripts. Then SoarBot.java uses the SoarSession object's loadAgent() method to load the poker agent, poker.soar.

2.1.1.4 newGame() method

The CPRG framework invokes SoarBot.java's newGame() method whenever a new game starts. The method parameters include the GameInfo object for the new game. SoarBot.java stores the parameters into member variables, and sets the necessary Tcl globals.

2.1.1.5 action() method

The CPRG framework invokes SoarBot.java's action() method whenever it's SoarBot's turn to bet. This is where the layers of software interact, and where the SoarBot agent plays poker. There are several steps:

1. SoarBot.java updates its member variables, and the corresponding Tcl globals, with current game state information. Then, it invokes the SoarSession runTillOutput() method.
2. SoarSession's runTillOutput() method invokes Soar's "run out" command, starting the Soar execution cycle (§ 1.8.2.2) on the poker.soar rule set.
3. The Soar execution cycle enters its input phase and runs iProc(), the Tcl input procedure (§ 1.8.2.2.2) in pokerlo.tcl. iProc() reads the Tcl globals that SoarBot.java populated, and moves the game state information into working memory elements on the ^input-link (§ 1.8.2.2.1). This makes the current values available for poker.soar rules to match against.
4. As the name runTillOutput() suggests, Soar runs its execution cycle repeatedly until the cycle produces output. In Soar terms, "output" means that Soar applies an operator that adds an attribute to the ^output-link (§ 1.8.2.2.1).

In particular, the rules in poker.soar set the output-link attribute ^bet.action to fold, call, raise, or check-raise.

5. Once there's an attribute on the ^output-link, the Soar execution cycle enters its output phase (§ 1.8.2.2.) and runs oProc(), the Tcl output procedure (§ 1.8.2.2.2) in pokerlo.tcl. oProc() transfers the working memory elements on the ^output-link (§ 1.8.2.2.1) into Tcl globals.
6. Soar stops running. SoarSession's runTillOutput() method returns, and SoarBot.java's action() method resumes execution. It reads the Tcl globals to see which betting action the poker.soar rule set selected, and it returns the action to

the CPRG client framework. The client framework sends a socket message off to the CPRG poker server. SoarBot has made its bet.

2.1.1.5.1 Check-Raise Bets

There's one special case that works differently: a check-raise bet. A check-raise is a two-stage bet where a player with a very strong hand checks (makes no bet), in the hope that someone else acting behind him will bet and he'll have a chance to raise. It's a risk - if no one else bets, the player has lost a chance to bet on a strong hand.

In SoarBot.java, a check-raise starts like any other bet, following the six steps above. However, if the ploy works out and SoarBot gets the chance to raise, SoarBot.java's action() method short-circuits the usual flow and simply returns the raise action. It bypasses the SoarSession and Soar layers, because there's no new decision to make.

Also, if SoarBot check-raises successfully once in a hand, SoarBot.java sets a flag so that SoarBot will not try it again later in the same hand. A check-raise makes a strong impression on the other players, and it's unlikely to work twice in a hand.

2.1.1.6 SoarBot.ini file

SoarBot.java gets run-time configuration information from SoarBot.ini, a Java Properties file in the current directory. SoarBot.ini has properties for the SoarBot version number, log file names, Tcl and Soar directories, etc.

2.2 The Rule Set

SoarBot's rule set consists of about 150 Soar rules in nine .soar files, and about 500 lines of supporting Tcl code in 3 .tcl files. This section covers the major files and some of their rules. For the reader without Soar experience, the rules may appear somewhat baffling, but they provide at least the flavor of Soar rule programming.

2.2.1 Working Memory Structures

For what follows, it's helpful to have a sense of the working memory structures that SoarBot uses. Caveat: This is a simplified snapshot - working memory changes constantly, and I've omitted many elements. This is an overview of the structures, not the details.

Each level of indentation shows a subattribute relationship, e.g. ^input-link is a subattribute of ^io, ^game is a subattribute of ^input-link, etc.

```
^io
  ^input-link
    ^game
      ^card
        ^num 1 ^rank a ^rank-num 14 ^suit c ^type hole
        ^next-rank 2
      ^card
        ^num 2 ^rank 4 ^rank-num 4 ^suit c ^type hole
        ^next-rank 5
      ^card
        ^num 3 ^rank a ^rank-num 14 ^suit s ^type board
        ^next-rank 2
      ...
      ^maybe yes
      ^analysis
      ^bets-to-call 1.5
      ^top-board-rank-num 14
      ^hutchinson-num 24
```

The input procedure, iProc() (§ 1.8.2.2.2), creates and updates a ^game structure on the input link as data flow down from SoarBot.java. The ^game structure has a ^card structure for each card in the hand. The ^type attribute distinguishes hole cards from board cards.

2.2.2 poker.soar

poker.soar is the main rule set file. It's the file that SoarBot.java loads into Soar. poker.soar in turn incorporates the other .soar and .tcl files, via the Tcl **source** command: "source *filename*" loads the contents of *filename* into the Tcl interpreter; it's loosely analogous to a C #include directive.

The rules in poker.soar itself do four things:

- Trigger a subgoal based on the stage of the poker hand;
- Select from among proposed operators;
- Apply the selected operator;
- Remove the selected operator from the ^output-link.

2.2.2.1 Subgoaling Based on the Stage of the Poker Hand

During the preflop stage of the hand (§ 1.2.3), poker.soar uses this rule to propose a preflop operator:

```

sp  {propose*preflop
    (state <s> ^name poker
        ^io.input-link <io>)
    (<io> ^game <g>)
    (<g> ^stage preflop)
-->
    (<s> ^operator <o>)
    (<o> ^name preflop)
}

```

poker.soar has similar rules for the flop, turn, and river stages of the hand.

These rules create operators with the same name as the stage. But the rules intentionally don't mark the operators as acceptable choices. This creates an operator no-change impasse, and Soar subgoals (§ 1.7.4.3).

I added this mechanism to poker.soar to force subgoaling, because subgoaling is a requirement for Soar's learning mechanism. As it happens, I never used learning, so the subgoaling serves no real purpose. But it doesn't hurt.

2.2.2.2 *Selecting From Among Proposed Operators*

Several of the rules in the other .soar files propose acceptable operators; in fact, it's normal for the rule set to propose multiple operators. poker.soar has rules that sort out all the proposed operators by preferring one operator over another (§ 1.7.4.2.2).

In SoarBot, each acceptable operator has a ^score attribute, and this rule simply prefers operators with higher scores:

```

sp  {select*prefer*higher*score
    (state <s> ^operator <o1> +
        ^operator { <> <o1> <o2> } +)
    (<o1> ^score <s1>)
    (<o2> ^score { > <s1> <s2> })
-->
    (<s> ^operator <o2> > <o1>)
}

```

This rule matches operators with the same score, and gives them an indifferent preference with respect to each other:

```

sp  {select*indifferent*tied*score
    (state <s> ^operator <o1> +
        ^operator { <> <o1> <o2> } +)
    (<o1> ^score <s1>)
}

```

```

    (<o2> ^score <s1>)
-->
    (<s> ^operator <o2> = <o1>)
}

```

This means that, if several rules have the same highest score, Soar is free to select any one of them. These two rules work together to let Soar select just one operator to apply from any number of proposed operators.

2.2.2.3 Applying the Selected Operator

poker.soar uses this rule to apply the selected operator:

```

sp {apply*bet*operator
    (state <s> ^operator <o>
        ^io <io>)
    (<io> ^output-link <ol>)
    (<o> ^bet.action <a>)
-->
    (<ol> ^bet.action <a>)
}

```

The rule simply copies the ^bet.action attribute from the operator to the ^output-link. Because Soar is running till output (§ 2.1.1.5), this change to the ^output-link ends Soar's execution cycle, and the selected ^bet.action flows back up the layers of software.

2.2.2.4 Removing the Selected Operator from the ^output-link

After the Tcl output procedure, oProc() (§ 1.8.2.2.2), retrieves the ^bet.action from the ^output-link and uses it to set a Tcl global, it adds a ^status attribute to ^bet, with a value of "complete". poker.soar has a rule that matches this and removes the completed ^bet from the ^output-link:

```

sp {remove*bet
    (state <s> ^operator <o>
        ^io.output-link <ol>)
    (<ol> ^bet <b>)
    (<b> ^status complete)
-->
    (<ol> ^bet <b> -)
}

```

The effect of this rule is to clear the ^output-link for the next execution cycle.

2.2.3 pokerlo.tcl

pokerlo.tcl is a file of Tcl code. It contains the input and output procedures (§ 1.8.2.2.2), iProc() and oProc(), and the utility functions they need. It handles the one-time registration of iProc() and oProc(), so that Soar automatically invokes them in its input and output phases.

2.2.4 analysis.soar

analysis.soar consists of 33 Soar rules that analyze a hold'em poker hand to figure out what it is - a pair, a flush, a full house, etc. The rules are useful for the flop, turn, and river stages of a hold'em game (§ 1.2.3) - all the stages *except* the initial preflop stage.

In Soar terms, these analysis rules are elaborations: they create working memory elements, but they don't propose operators.

Section 2.2.1 shows an ^analysis attribute on the ^game structure. The iProc() input procedure (§ 1.8.2.2.2) automatically clears the ^analysis attribute whenever a new hand starts, and at each new stage of the hand. The left-hand-sides of the rules in analysis.soar match the card patterns, and the right-hand-sides add attributes to the ^analysis attribute.

For example, this rule matches if SoarBot has the highest pair possible at any point in the hand:

```
sp {elaborate*analysis*top-pair
  (state <s> ^io.input-link.game <g>)
  (<g> ^analysis <an>
    ^card <c1>
    ^card <c2>
    ^top-board-rank-num <tbrn>)
  # Use num to make sure we're looking at different cards.
  # Use rank-num instead of rank so that we can compare.
  (<c1> ^rank-num { <rn1> >= <tbrn> } ^num <num1> ^type hole)
  (<c2> ^rank-num <rn1> ^num { <num2> > <num1> })
  # But don't set this if we've also set...
  (<an> -^top-2-pair yes
    -^trips-with-hole-card yes
    -^full-house-with-hole-card yes
    -^4-of-a-kind-with-hole-card yes)
-->
  (<an> ^top-pair yes)
}
```

This rule matches two cards that have the same rank. The rank must be equal to or greater than the rank of the highest-ranking board card, and at least one of the matching cards must be a hole card. If these conditions all match, then SoarBot has a pair at least as high as the highest possible pair any player can make from the board cards. When this rule fires, it adds a ^top-pair attribute, with a value of “yes”, to the ^analysis attribute.

In these lines, the rule checks to make sure other analysis.soar rules haven't found a higher hand:

```
(<an> -^top-2-pair yes
      -^trips-with-hole-card yes
      -^full-house-with-hole-card yes
      -^4-of-a-kind-with-hole-card yes)
```

The minus sign in front of the attribute is a negation; for the rule to match, the working memory element must *not* exist. So, if the ^analysis attribute has any of the higher hands, this rule won't match, and ^top-pair won't be added. Since all the rules fire simultaneously (§ 1.7.4.2.1), there's no problem if this rule fires, then retracts automatically when another rule sets a higher ^analysis attribute.

It's up to the knowledge engineer to decide whether attributes should be mutually exclusive or not. In some cases, there's no conflict between multiple ^analysis attributes. For example, whenever SoarBot has a pair using a hole card, the elaborate*analysis*pair rule adds a ^pair attribute to ^analysis. If SoarBot's pair is also the top pair, the ^analysis attribute will end up with both ^pair and ^top-pair attributes. That's fine - the distinction between ^pair and ^top-pair is a matter of degree, and there may be other rules that look for one value of the other.

Another analysis.soar example: This rule matches whenever SoarBot has a flush, five cards of the same suit, using at least one of its hole cards:

```
sp {elaborate*analysis*flush-with-hole-card
    (state <s> ^io.input-link.game <g>)
    (<g> ^analysis <an>
      ^card <c1>
      ^card <c2>
      ^card <c3>
      ^card <c4>
      ^card <c5>)
  # Use num to make sure we're looking at different cards.
  (<c1> ^num <num1> ^suit <s1> ^type hole)
  (<c2> ^num { <num2> > <num1> } ^suit <s1>)
  (<c3> ^num { <num3> > <num2> } ^suit <s1>)
  (<c4> ^num { <num4> > <num3> } ^suit <s1>)
  (<c5> ^num { <num5> > <num4> } ^suit <s1>)
-->
```

```

    (<an> ^flush-with-hole-card yes)
}

```

The first hole card always has a ^num value of 1, the second hole card is ^num 2, and the board cards are ^num 3 through 7. This left-hand-side pattern will match whenever one or both hole cards are part of the flush.

Because the rule programming language is limited to just if-then statements, with no looping constructs, I was wary of how far I would get with poker hand analysis before I ran into problems. I hit a bump when I tried to find a pattern for a straight, a sequence of cards in order by rank, e.g. 7-8-9-10-jack. Here's the rule I used:

```

sp {elaborate*analysis*straight
  (state <s> ^io.input-link.game <g>)
  (<g> ^analysis <an>
    ^card <c1>
    ^card <c2>
    ^card <c3>
    ^card <c4>
    ^card <c5>)
  (<c1> ^next-rank <nr1>)
  # Middle ranks can't be ace
  (<c2> ^rank { <nr1> <> a } ^next-rank <nr2>)
  (<c3> ^rank { <nr2> <> a } ^next-rank <nr3>)
  (<c4> ^rank { <nr3> <> a } ^next-rank <nr4>)
  (<c5> ^rank <nr4>)
-->
  (<an> ^straight yes)
}

```

The problem is that this rule doesn't specifically match a hole card. It's possible that the five board cards in a hold'em hand will form a straight; this rule will match when that happens, as well as when a hole card is part of the straight. In the flush rule above, the order of the cards doesn't matter - what matters is that they have a common suit. So it's simple to match a ^type value of "hole" on one of the cards. For a straight, card order is critical. Any of the five cards could be the hole card. What's needed is a pattern that says, "Make sure <c1> is a hole card, or <c2>, or <c3>, or <c4>, or <c5>." But Soar's rule syntax doesn't support this sort of "or" construct.

I could have worked around this with five separate rules, each one matching a different card as the hole card. But that's ugly. Instead, as a workaround, I added a rule, elaborate*analysis*straight-on-board, that specifically tests for a straight using the five board cards and adds a ^straight-on-board attribute to ^analysis. Then, if ^analysis has the ^straight attribute and not the ^straight-on-board attribute, I know that SoarBot has a straight that depends on a hole card.

Unfortunately, if both attributes are set, they may both be signaling the same straight, or they may be signaling two different straights: one using the board cards, and another using one or both hole cards.

Another shortcoming that I didn't tackle is that my hand analysis rule set isn't granular enough to handle tiebreaker cards, i.e. the rule set recognizes a flush, but it doesn't distinguish between flushes. The broad distinctions are straightforward: a flush beats a straight, a straight beats three of a kind, etc. But, within each hand type, higher cards win, e.g. if the two best hands in the game are both flushes, the flush with the highest card wins. If both flushes have the same high card, the flush with the second-highest card wins, and so on.

Because of the five shared board cards in hold'em, this sort of analysis can sometimes go to several levels. Suppose the five board cards in a hand are:

- Ace of spades;
- King of spades;
- 8 of hearts;
- 7 of spades;
- 2 of spades

Each player has just two hole cards, so the best possible hand is a spade flush. Because there are four spades on board, it's possible that more than one player will have the flush. With these board cards, a jack of spades is a significantly more valuable card than a four of spades. Both cards make the flush, but the jack makes a flush that loses only to the queen, while the four is very weak - it loses to almost any other flush. And any player who has the queen of spades has the "nuts" - the guaranteed winning hand.

This is easy for a human player to see. It's also fairly easy to code up in a conventional programming language with loops. But Soar rules to handle this example would be unwieldy, and would probably depend on help from the Java or Tcl layers.

In practice, tiebreaker hands don't come up all that frequently, but they do come up, and the small details can add up to the difference between a winning poker agent and a loser. In retrospect, I think I made a mistake by doing the hand analysis as a suite of Soar rules, for several reasons:

- Problems analyzing straights and tiebreakers;
- The negated attributes that check for higher hands are repetitive and hard to maintain;
- It's easy for syntax errors to pass unnoticed, e.g. errors in variable names;

- Running unit tests on Soar rules to make sure they're matching what they should, and not matching what they shouldn't, requires a lot of setup work and is hard to automate.

If I had coded the hand analysis logic in a more conventional programming language and passed the results down to Soar on the ^input-link, these problems would either be more manageable or disappear completely. Soar rules aren't a good fit for some problems; using Soar rules successfully includes knowing when *not* to use them.

2.2.5 defaultOps.soar

The rules in defaultOps.soar propose operators for actions that SoarBot considers any time it has to act. For example, SoarBot always considers folding its cards and quitting the hand. This rule proposes folding in all possible substates:

```
sp {propose*fold*default
  (state <s> ^name << preflop flop turn river >>
    ^io <io>)
  (<io> ^input-link <il>
    ^output-link <ol>)
  (<il> ^clock <cl>) # line 6
-->
  (<s> ^operator <o> +)
  (<o> ^name fold*default
    ^bet.action fold
    ^score 10)
}
```

The ^operator score attribute has a low value, 10, so that Soar will prefer other operators with higher scores.

The ^clock attribute in line 6 ensures that this proposal retracts and fires automatically on each execution cycle as the ^clock value changes. This is a handy Soar idiom that I used on all SoarBot's proposals.

Another default rule is that SoarBot should never fold if it can stay in the hand for free:

```
sp {propose*check*no-cost
  (state <s> ^name << preflop flop turn river >>
    ^io <io>)
  (<io> ^input-link <il>
    ^output-link <ol>)
  (<il> ^clock <cl>)
```

```

        ^game <g>)
    (<g> ^bets-to-call 0.0)
-->
    (<s> ^operator <o> +)
    (<o> ^name check*no-cost
        ^bet.action call
        ^score 20)
}

```

The key match on the left-hand-side is ^bets-to-call. If this is zero, it won't cost SoarBot anything to stay in the hand. And, because the check*no-cost operator has a ^score of 20, Soar will always prefer it to the fold*default operator when the rule set proposes both.

2.2.6 preflop_loose.soar

preflop_loose.soar is the rule set for the preflop stage of a hold'em hand (§ 1.2.3). This is the first stage of the hand, where each player gets two hole cards and no board cards have been dealt yet. Thus, a player's preflop play is largely a function of his hole cards and position at the table.

The rule set begins with eleven elaboration rules, similar to the analysis rules in analysis.soar, except that these rules are specialized for the preflop stage - they match against just the two hole cards.

Here's the rule that matches a pair of aces, the best starting hand:

```

sp {preflop*elaborate*preflop-strength*pair-aces
    (state <s>          ^name preflop
                        ^io.input-link.game <g>)
    (<g> ^card <hc1>
        ^card <hc2>)
    (<hc1> ^type hole ^num 1 ^rank a)
    (<hc2> ^type hole ^num 2 ^rank a)
-->
    (<g> ^preflop-strength 1)
}

```

Here are a couple of rules that match high cards of the same suit:

```

sp {preflop*elaborate*preflop-strength*ace-king-suited
    (state <s> ^name preflop
              ^io.input-link.game <g>)
    (<g> ^card <hc1>
        ^card <hc2>)
    (<hc1> ^type hole ^suit <s1> ^rank a)
}

```

```

    (<hc2> ^type hole ^suit <s1> ^rank k)
-->
    (<g> ^preflop-strength 1)
}

sp {preflop*elaborate*preflop-strength*2-high-cards-suited
    (state <s> ^name preflop
        ^io.input-link.game <g>)
    (<g> ^card <hc1>
        ^card <hc2>)
    (<hc1> ^type hole ^suit <s1>
        ^rank { << a k q j >> <r1> } )    # line 7
    (<hc2> ^type hole ^suit <s1>
        ^rank { << a k q j >> <> <r1> }) # line 9
-->
    (<g> ^preflop-strength 2)
}

```

In the second rule, the << a k q j >> syntax in line 7 matches the ^rank attribute against any of the bracketed values; if there's a match, the matching value ends up in the <r1> variable. In line 9, the << a k q j >> <> <r1> syntax matches any of the bracketed values *except* the value in <r1>. This extra check keeps Soar from matching the same card twice.

An ace and king of the same suit will match both rules. In that case, both rules fire and the ^game attribute ends up with two ^preflop-strength attributes, one with a value of 1, the other with a value of 2. The two ^preflop-strength attributes will in turn trigger two different operator proposal rules. This is not a problem. Soar will use the operator preference rules (§ 1.7.4.2.2) to select just one operator. Soar's rich execution cycle (§ 1.7.4.2.1) frees the knowledge engineer from having to worry about what might be conflicting rules in another rule-based system.

The elaboration rules add one or more ^preflop-strength attributes to the ^game. preflop_loose.soar has seven proposal rules that match ^preflop-strength, and other factors, to come up with betting action proposals. For example, this rule proposes that SoarBot check-raise:

```

sp {preflop*propose*check-raise
    (state <s> ^name preflop
        ^io <io>)
    (<io> ^input-link <il>
        ^output-link <ol>)
    (<il> ^clock <cl>
        ^game <g>)
    (<g> ^num-bets <= 1                # line 8
        ^bet-timing early             # line 9)
}

```

```

        ^preflop-strength 1)          # line 10
-->
    (<s> ^operator <o> +)
    (<o> ^name preflop*check-raise
        ^bet.action check-raise
        ^score 40)
}

```

The rule matches three key conditions in lines 8-10:

- No more than one bet so far. (There's always at least one bet in the preflop stage, because of the big blind.);
- SoarBot is in early position at the table, so that there's a good chance some other player acting after SoarBot will bet;
- SoarBot has a very strong hand.

2.2.6.1 The ^maybe Attribute

In order to be somewhat unpredictable to other players, the `preflop_loose` rule set varies its play when it has strong cards. The bet-raise proposals match against the `^maybe` attribute, which is randomized in the Tcl layer so that it's "yes" half the time and "no" half the time. If `^maybe` is "yes", the bet-raise rules match and SoarBot will bet or raise. If not, the bet-raise rules don't match, and SoarBot will simply call. In this example, the `^maybe` attribute test is on line 9:

```

sp {preflop*propose*bet-raise*excellent-cards
    (state <s> ^name preflop
        ^io <io>)
    (<io> ^input-link <il>
        ^output-link <ol>)
    (<il> ^clock <cl>
        ^game <g>)
    (<g> ^num-bets <= 1
        ^maybe yes          # line 9
        ^preflop-strength <= 2)
-->
    (<s> ^operator <o> +)
    (<o> ^name preflop*bet-raise*excellent-cards
        ^bet.action raise
        ^score 30)
}

```

The basic strategy behind `preflop_loose.soar` is my own, based on my experience in low-stakes hold'em games. Its name is perhaps misleading, because I don't think it's a particularly loose strategy. In poker parlance, a "loose" player stays in too many hands, playing cards that have poor winning chances. A

“tight” player stays in too few hands, discarding playable cards while he waits for sure winners. I think the strategy behind `preflop_loose.soar` is a good balance.

I called the rule set “`preflop_loose`” to distinguish it from another, noticeably tighter, `preflop` strategy that I experimented with: Edward Hutchison’s system. Hutchison’s system is a straightforward algorithm that assigns a point value to any two hold’em hole cards. I call this point value a Hutchinson number. (While writing this thesis, I noticed that I had spelled Edward Hutchison’s name incorrectly during the development of SoarBot. There’s no “n” in his last name. I have not fixed the mistake in SoarBot’s code.)

SoarBot computes the Hutchinson number in the Java layer and adds it to the `^game` as the `^hutchinson-num` attribute. This simplifies the `preflop_hutchinson.soar` rule set - it needs no elaborations to assess the strength of the hole cards, just proposals based on position and the `^hutchinson-num`.

In practice, I found Hutchison’s system much too tight. Using it, SoarBot played too few hands, waiting for a sure thing. Hutchison designed his system for novice players, and he explains that it’s tight for a good reason: “...the newcomer will pay a higher penalty for being too loose than being too tight.”

To make it simple to switch `preflop` strategies, I used the Tcl source command. `poker.soar` sources `preflop.soar`, which looks like this:

```
#source preflop_hutchinson.soar
source preflop_loose.soar
```

To switch `preflop` strategies, comment out one line and uncomment the other:

2.2.7 flop.soar

At the `flop` stage, the dealer turns over the first three board cards. `flop.soar` begins with 22 elaborations that use the `^analysis` attribute set by `analysis.soar` to set a `^flop-strength` attribute:

```
sp {flop*elaborate*flop-strength*flush
   (state <s> ^name flop
     ^io.input-link.game <g>)
   (<g> ^analysis <an>)
   (<an> ^flush-with-hole-card yes)
-->
   (<g> ^flop-strength 1)
}
```

2.2.7.1 Semibluffs

Some of the elaborations look for “semibluff” opportunities (Sklansky and Malmuth 33). A semibluff is

“...a bet with a hand which, if called, does not figure to be the best hand at the moment but has a reasonable chance of outdrawing those hands that initially called it...there are two ways that you might win the pot. First, no one may call and you win the pot immediately. Second, if you do get customers, you still may improve to the best hand.”

A semibluff hand has a couple of ways to turn into a strong hand. For example, this rule matches hands that are a card away from turning into either a flush or three-of-a-kind:

```
sp {flop*elaborate*flop-strength*semibluff*4-flush*pair
    (state <s> ^name flop
        ^io.input-link.game <g>)
    (<g> ^analysis <an>)
    (<an> ^4-flush-with-hole-card yes
        ^pair yes)
-->
    (<g> ^flop-strength 3.5)
}
```

Many of the flop rules that propose actions look at whether or not the other players have done any betting. This rule proposes a bet if SoarBot has good cards (line 9) and no one else has bet yet (line 8). SoarBot does not want to give the other players a free card:

```
sp {flop*propose*bet*first
    (state <s> ^name flop
        ^io <io>)
    (<io> ^input-link <il>
        ^output-link <ol>)
    (<il> ^clock <cl>
        ^game <g>)
    (<g> ^num-bets 0 # line 8
        ^flop-strength <= 4) # line 9
-->
    (<s> ^operator <o> +)
    (<o> ^name flop*bet*first
        ^bet.action raise
        ^score 30)
}
```

Depending on how many other players folded in the preflop and earlier in the flop, SoarBot may find itself playing heads-up against just one other player. The other player may bluff in an attempt to steal the pot, and SoarBot may have a hand that, while too weak to play in a multi-player game, is now playable. flop.soar has a special rule for this situation:

```

sp {flop*propose*call*1-player
  (state <s> ^name flop
    ^io <io>)
  (<io> ^input-link <il>
    ^output-link <ol>)
  (<il> ^clock <cl>
    ^game <g>)
  (<g> ^num-active-players 2 # line 8
    ^flop-strength <= 7) # line 9
-->
  (<s> ^operator <o> +)
  (<o> ^name flop*call*1-player
    ^bet.action call
    ^score 20)
}
```

If SoarBot is one of just two players in the hand (line 8), it will stay in with cards that have any potential at all (line 9).

Of course, such special-case fine tuning is not without its own risks. On more than one occasion, I added a rule to correct a weakness in SoarBot's play, only to create a new, more glaring problem. Unintended consequences are an ongoing challenge for the knowledge engineer. A rule-based system invites tweaking. It's easy to add a rule here, change a pattern there. But the knowledge engineer may find himself poking a balloon - fix a problem in one place, and out pops trouble somewhere else.

2.2.8 turn.soar

At the turn stage, the dealer turns over the fourth board card, and the bet amount doubles. There are usually few players left in the hand, and they have solid cards. Bad decisions from here on are expensive.

turn.soar is similar in structure to flop.soar. It begins with 27 elaborations that use the ^analysis attribute set by analysis.soar to set a ^turn-strength attribute. As with the flop elaborations, the elaborations include semibluffs.

However, unlike the flop elaborations, the turn elaborations look closely at the board cards. With four cards exposed, dangers emerge, and the elaborations test SoarBot's hand against these dangers.

For example, top pair is a pair, using a hole card, at least as high as the highest board card. Its value varies significantly depending on what the board looks like:

```

sp {turn*elaborate*turn-strength*top-pair*no-danger
  (state <s> ^name turn
    ^io.input-link.game <g>)
  (<g> ^analysis <an>)
  (<an> ^top-pair yes
    # Any obvious dangers?
    -^pair-on-board yes
    -^2-pair-on-board yes
    -^trips-on-board yes
    -^4-straight-on-board yes
    -^straight-on-board yes
    -^4-flush-on-board yes
    -^flush-on-board yes
    -^full-house-on-board yes
    -^4-of-a-kind-on-board yes)
-->
  (<g> ^turn-strength 2)
}

```

This elaboration gives top pair a high ^turn-strength *if* no dangerous hand is on board. Its companion elaboration sets a much lower ^turn-strength for top pair:

```

sp {turn*elaborate*turn-strength*top-pair*danger
  (state <s> ^name turn
    ^io.input-link.game <g>)
  (<g> ^analysis <an>
    ^pot-odds >= 5) # line 5
  (<an> ^top-pair yes)
-->
  (<g> ^turn-strength 6)
}

```

In addition, the above elaboration matches only if SoarBot is getting pot odds of 5-to-1 or better (line 5).

2.2.8.1 Pot Odds

Pot odds are

the odds the pot is giving you for calling a bet. If there is \$50 in the pot and the final bet was \$10, you are getting 5-to-1 odds for your call...if you figure

you chances of winning are better than 5-to-1, then it is correct to call. If you think your chances are worse than 5-to-1, you should fold. (T of P 35).

Each time SoarBot has to act, the Java layer recalculates the pot odds value and passes it down as an attribute of the ^game. The value is always expressed in terms of a ratio to 1, i.e. ^pot-odds of 5 means 5-to-1.

The 5 or better pot odds requirement in the above elaboration rule is not mathematically rigorous. Rather, it's an attempt to make sure that, if SoarBot chases rainbows, the pots of gold have some heft.

If SoarBot has nothing more than top pair, and there's a 4-flush on board, and the pot odds are only 3-to-1, neither of the above turn elaborations will match and fire. There will be no ^turn-strength attribute on the ^game, so none of the turn stage proposals will fire. SoarBot will fold because of the propose*fold*default rule (§ 2.2.5). This is what we want.

On the other hand, if there's no danger on board and the pot odds are at least 5-to-1, *both* turn elaborations will fire, and the ^game will end up with two ^turn-strength attributes, with different values. This is not a problem either. See section 2.2.6 for a discussion of multiple strength-type attributes and operator proposals.

The rule set has two proposals that match an ^analysis value of ^blank-on-board. A blank is a board card that doesn't improve the board, i.e. it isn't part of a pair or a straight or flush. A blank is significant on the turn because it may mean that the turn card did not help anyone. For example, suppose the flop board cards are three of clubs, nine of diamonds, and queen of spades. If the turn board card is the four of hearts, it's a blank - it probably didn't help anyone.

This rule proposes a bet if the turn card is a blank:

```
sp {turn*propose*bet-first*blank-on-board
  (state <s> ^name turn
    ^io <io>)
  (<io> ^input-link <il>
    ^output-link <ol>)
  (<il> ^clock <cl>
    ^game <g>)
  (<g> ^num-bets 0
    ^unacted 1 # We're last
    ^analysis.blank-on-board yes
    ^turn-strength <= 7)
-->
  (<s> ^operator <o> +)
  (<o> ^name turn*bet-first*blank-on-board
    ^bet.action raise
```

```

    ^score 30)
}

```

The rule also makes sure that SoarBot is in last position with at least mediocre cards, and that no one else has bet. SoarBot may be able to drive out the other players, and the blank card makes it unlikely that any other player has a strong hand and is waiting to check-raise. The situation is an opportunity for SoarBot to feign more strength than it may really have.

Most of the proposal rules in `turn.soar` are similar to those in `flop.soar`, using `^turn-strength` and other `^game` attributes to decide when to propose bets, raises, check-raises, or calls. The turn rule set also includes two special fold proposals that match and fire when there are dangerous board cards. These fold proposals will override any of the other proposals. For example:

```

sp {turn*propose*fold*4-flush-on-board
  (state <s> ^name turn
    ^io <io>)
  (<io> ^input-link <il>
    ^output-link <ol>)
  (<il> ^clock <cl>
    ^game <g>)
  (<g> ^bets-to-call >= 1.0
    ^analysis <an>)
  (<an> ^4-flush-on-board yes
    # Any reason to argue?
    -^2-pair yes
    -^top-2-pair yes
    -^trips-with-hole-card yes
    -^flush-with-hole-card yes
    -^full-house-with-hole-card yes
    -^4-of-a-kind-with-hole-card yes)
  -->
  (<s> ^operator <o> +)
  (<o> ^name turn*fold*4-flush-on-board
    ^bet.action fold
    ^score 100)
}

```

If the four board cards are all of the same suit, SoarBot will fold unless it has a chance to end up with a flush or better. The high `^score` of 100 ensures that SoarBot will prefer this fold operator to any other operators proposed.

This special-case rule doesn't consider `^turn-strength` at all. It sits outside the framework of the `^turn-strength`-driven elaborations and proposals, as a necessary refinement. For example, the `turn*elaborate*turn-strength*top-pair*danger` rule I explained above will set `^turn-strength` to 6 when SoarBot has

top pair, even in the face of dangers that include a 4-flush on board. The `turn*propose*call*playable-cards` rule in `turn.soar` will propose calling a single bet with a `^turn-strength` of 6. This rule overrides them when the specific danger is a 4-flush on board, without interfering in other cases.

Soar's preference system makes it possible for the knowledge engineer to add special-case rules as needed. This simplifies maintenance: it's much easier to add one special-case rule to a rule set than to modify a dozen elaborations to make sure they check for a 4-flush. This approach also, I think, fits the way that many human experts talk about their domains, e.g. "Follow these rules most of the time, but there's a special case that takes precedence whenever it come up."

2.2.9 river.soar

The river card is the last board card. Once the dealer turns it over, each player is certain of his own hand, and he can calculate his hand's relative strength compared to the universe of possible hands based on the five board cards. SoarBot does this calculation in the Java layer, and passes the result to the `^game` as the `^best-hand-probability` attribute.

The calculations behind `^best-hand-probability` come from the CPRG framework `poker.HandEvaluator` class (Davidson, *Computer Poker*). The `HandEvaluator` class is itself a wrapper around the open source poker evaluation library available from PokerSource.org. A call to the evaluation library, through the CPRG wrapper, looks like this:

```
d = m_evaluator.handRank(m_holeCard1, m_holeCard2,
    boardCards, numActivePlayers);
```

The number that comes back is the probability that a hand formed from the input board cards and hole cards is the best hand in the game. In practice, this number is artificially high, because the evaluation library considers the universe of all possible hands. In a hold'em game, players with poor hole cards will usually fold in the preflop stage. Players whose potential strong hands break down in the flop and turn stages will also tend to fold. So, by the river stage, the players still in the game usually have much better hole cards, and hands, than a random distribution would suggest.

Because the river rule set uses the `^best-hand-probability` attribute, it does not need any elaboration rules for hand-strength analysis. It has just five rules, all of them proposals. The proposals adjust for the evaluation library's bias towards high probabilities.

For example, this rule makes sure SoarBot folds any hand with a `^best-hand-probability` under .40:

```

sp {river*propose*fold*based-on-probability
  (state <s> ^name river
    ^io <io>)
  (<io> ^input-link <il>
    ^output-link <ol>)
  (<il> ^clock <cl>
    ^game <g>)
  (<g> ^bets-to-call >= 1.0
    ^best-hand-probability < .40)
-->
  (<s> ^operator <o> +)
  (<o> ^name river*fold*based-on-probability
    ^bet.action fold
    ^score 100)
}

```

When ^best-hand-probability is .90 or higher and the game situation is right, SoarBot will check-raise:

```

sp {river*propose*check-raise
  (state <s> ^name river
    ^io <io>)
  (<io> ^input-link <il>
    ^output-link <ol>)
  (<il> ^clock <cl>
    ^game <g>)
  (<g> ^check-raise-used no # Just once per game
    ^num-bets 0
    ^unacted >= 4 # line 10
    ^best-hand-probability >= .90)
-->
  (<s> ^operator <o> +)
  (<o> ^name river*check-raise
    ^bet.action check-raise
    ^score 40)
}

```

The key match here is ^unacted in line 10. A value of four means that three other players will have a chance to bet after SoarBot checks, so there's a good chance SoarBot will get a chance to finish the check-raise.

If ^best-hand-probability is .75 or higher, SoarBot has a strong hand and will stay in the hand to the end. Whether it calls, bets, or raises depends on what the other players do. If ^best-hand-probability is between .40 and .75, SoarBot makes its decision based on this rule:

```

sp {river*propose*call*based-on-adjusted-odds
  (state <s> ^name river
      ^io <io>)
  (<io> ^input-link <il>
      ^output-link <ol>)
  (<il> ^clock <cl>
      ^game <g>)
  (<g> ^adjusted-odds { <ao> > 1.00 }) # line 8
-->
  (write (crlf) |adjusted-odds == | <ao>)
  (<s> ^operator <o> +)
  (<o> ^name river*call*based-on-adjusted-odds
      ^bet.action call
      ^score 20)
}

```

The key is the match on the ^adjusted-odds value (line 8). The ^adjusted-odds attribute is computed by a rule in pokerElab.soar, one of the files in poker.soar. For the river stage of the hand, ^adjusted-odds is simply ^best-hand-probability * ^pot-odds.

The ^adjusted-odds number is my attempt to adjust the ^pot-odds to compensate for the high bias in ^best-hand-probability. The lower the ^best-hand-probability, the more likely the bias, and the greater the effect of the adjustment. If ^adjusted-odds is greater than 1.00, then the potential reward is greater than the risk, even with the bias, and SoarBot calls the bet. (I arrived at the 1.00 value heuristically - my first try was 1.50, but it made SoarBot's river play *too* cautious, so I dialed it down.)

2.2.10 Log Files

SoarBot maintains several log files as it plays. Output from the CPRG poker server goes into a daily log file named alberta-yyyy-mm-dd.log. This is the file that shows the poker game action: who bet, who folded, who won. Multiple poker sessions in a day are appended to the same day's file.

Debugging output from the Soar level goes into SoarBot-yyyy-mm-dd.log.

A comma-delimited file of game results goes into results.txt. This file is designed so that you can easily import it into a database or spreadsheet for further analysis. It's not a daily file - it grows bigger over time.

You can use the unique game numbers in these three files to cross-reference between them.

SoarBot also maintains a simple player history for all its opponents in PlayerHistory.xml, but it doesn't use this information in its play.

In a standard SoarBot installation on Linux, all log files end up in the logs subdirectory under /your/SoarBot/directory. See the readme.txt file that accompanies the SoarBot source code (Appendix A) for more information.

2.2.11 Bluffing and Deception

Before I leave SoarBot's technical details behind, I want to discuss its approach to bluffing and deception. In short, SoarBot does not ever bluff, where bluffing is

```
...betting when you are quite sure that you do not
have the best hand (and have little chance of making
the best hand) and hoping that your opponent folds.
(Sklansky and Malmuth, 57)
```

SoarBot bluffed quite a bit in its early versions, but the bluffs almost never worked. A bluff in a low-stakes poker game is difficult to pull off - anyone with playable cards will probably risk a bet or two to "keep you honest." And, in a game like the CPRG's, where the money is virtual and unlimited, bluffing is that much harder. Because there's no real financial penalty for bad play, the situation is almost paradoxical: more experienced human players are more bluffable than beginners, because the experienced players want to do well, while the beginners don't care how much they "lose."

But SoarBot still has some deception in its play:

- In the preflop stage, SoarBot uses the ^maybe attribute to randomize how aggressively it plays strong cards;
- In the flop and turn stages, SoarBot looks for semibluff opportunities, so that opponents cannot always take its aggressive play for granted;
- In the preflop, turn, and river stages, SoarBot looks for check-raise opportunities.

These strategies give SoarBot some ability to occasionally surprise its human opponents.

3. Results

3.1 Mixed Human/Bot Games

On thirty-five days spread out over a little more than three months, versions of SoarBot competed on the CPRG poker server against both humans and other bots:

Date	# of Hands *	\$ Won/Lost Per Hand	10-Day Weighted Moving Average
10/25/02	515	-0.48	
10/28/02	615	-3.56	
11/05/02	540	-4.16	
11/11/02	1,124	-1.24	
11/13/02	1,402	0.65	
11/14/02	1,787	-0.23	
11/15/02	851	-0.47	
11/16/02	1,029	-1.10	
11/23/02	1,593	-1.39	
12/02/02	877	3.38	-0.62
12/03/02	1,750	-0.86	-0.66
12/05/02	1,780	-1.08	-0.58
12/06/02	1,036	0.62	-0.34
12/07/02	2,159	-1.06	-0.38
12/09/02	515	-4.22	-0.63
12/10/02	1,068	0.26	-0.61
12/12/02	663	-0.43	-0.61
12/13/02	1,182	0.22	-0.49
12/14/02	544	4.33	-0.14
12/15/02	1,121	0.98	-0.30
12/30/02	1,181	-1.78	-0.37
01/01/03	541	-1.89	-0.32
01/03/03	1,339	-1.53	-0.57
01/10/03	714	-1.76	-0.55
01/11/03	1,099	0.05	-0.28
01/13/03	1,606	-1.23	-0.49
01/14/03	594	-0.63	-0.51
01/15/03	828	0.18	-0.54
01/17/03	1,118	1.89	-0.53
01/20/03	1,873	0.32	-0.54
01/22/03	810	-0.01	-0.36
01/23/03	2,545	-0.81	-0.38
01/26/03	1,858	-0.33	-0.26
01/27/03	592	-3.03	-0.30
01/28/03	1,003	-0.99	-0.39

* This chart omits days on which SoarBot played fewer than 500 hands.

3.1.1 Analysis

Because of fluctuations in quality of cards and quality of opponents, it's difficult to compare one single day to another in a meaningful way. Fluctuations are a normal part of poker. No one wins every hand, or even every day:

Each individual game is part of one big poker game. You cannot win every game or session you play, anymore than a golfer or bowler can win every match he or she plays. If you are a serious poker player, you must think in terms of your win at the end of the year or the end of the month - or, as sometimes happens, of your loss at the end of the year or the end of the month, which, of course, you want to keep as small as possible. (Sklansky 6)

The 10-Day Weighted Moving Average attempts to smooth out short-term fluctuations by taking a longer perspective. It shows an improvement in SoarBot's play over time, from losses of roughly \$0.60 per hand in early versions to consistent losses closer to \$0.35 a hand in the later versions.

The tweaks I made to the rule set improved SoarBot's play, but not dramatically. I was able to get it to lose money more slowly, but it was never a consistent winner.

To put SoarBot's numbers in perspective, this chart from a log file shows the best and worst human players in the CPRG game as of February 5, 2003:

```

+-----+
|          TOP TEN SHARKS AND FISH (sb/h)          |
| min 1000 hands required (513 eligible)          |
+-----+
|          DA SHARKS          |          DA FISH          |
+-----+
| swill          0.5029      | svmbot          -2.260    |
| peteux         0.4471      | livermore       -0.546    |
| frodo          0.35        | skarbie         -0.522    |
| sirwang        0.3234      | mona            -0.424    |
| rocko          0.2943      | fembot          -0.413    |
| illarionb      0.2925      | docmike         -0.409    |
| ducks          0.2382      | toonman         -0.37     |
| will           0.2308      | izedss          -0.366    |
| wmbot          0.2032      | mrgekko         -0.363    |
| willbot        0.2011      | amateur         -0.361    |
+-----+

```

The "sb/h" measure is small bets per hand, where a small bet is \$10. So swill has won about \$5.03 a hand, on average. In the other direction, livermore brings up the rear, losing about \$5.46 a hand. (I'm excluding svmbot because it's really a

bot and it plays pathologically poor poker, an order of magnitude worse than any other player in the game. Incidentally, the word “bot” in a name doesn’t mean anything. Some humans in the game use bot-like names: wmbot, willbot, and fembot are all humans.)

The CPRG’s Pokibot agent plays much better than SoarBot does in mixed human/bot games. As of late April 2003, Pokibot had played over half a million hands in mixed games, winning about \$0.95 per hand on average.

3.2 Bot-Only Games

Starting in February 2003, the CPRG set up a bot-only game. They asked bot developers to keep their bots in the bot-only game until/unless their bots were able to break even against the CPRG bots. The CPRG did this to open up more room for human players in the mixed games (Online Poker Protocol, “Bot Rules & Etiquette” par.).

Their casino, their rules, so I moved SoarBot to the bot-only games for four days, where it played against Poki and some other CPRG bots:

Date	# of Hands	\$ Won/Lost Per Hand
02/02/03	5,435	-0.70
02/03/03	2,152	-0.99
02/04/03	2,469	-1.45
02/05/03	1,491	-1.54

3.2.1 Analysis

SoarBot did significantly worse when it was playing against just the CPRG bots, losing about \$1.02 a hand on average. I wasn’t surprised; the CPRG bots play better than most of the human players in the mixed games. There’s an old saying in poker: “If you look around the table and you can’t spot the sucker, it’s you.” Playing in a game without any weaker players, SoarBot was the sucker.

I think the strong play of Poki and the other CPRG bots comes from their opponent modeling capabilities. As they gather data over a sufficiently large number of games, they grow very good at predicting each player’s hidden hole cards. In poker, if you can make your information a little less imperfect, you have a marked advantage.

4. Conclusions

SoarBot is a mediocre poker player. It has some skill - it plays better than weak human players, and it could probably make money against them. Against strong human and bot players, it loses money.

SoarBot does not play nearly as well as the CPRG bots. SoarBot uses a static rule set; over time, its bluffing and deception strategies are not enough to keep the CPRG bots from modeling it accurately. SoarBot does no opponent modeling of its own, so it's at a considerable competitive disadvantage against bots that do it well. Playing poker without opponent modeling is like playing chess without looking at the other player's pieces.

Another part of the explanation may be that, throughout this project, I wore three hats:

- Systems programmer, integrating Soar and the CPRG poker framework;
- Knowledge engineer, writing SoarBot's rule set;
- Expert, telling the knowledge engineer how to play poker well.

Arguably, the expert hat is not a good fit, because I'm far from an expert poker player. In fact, SoarBot plays about as well as I do: I can beat weak players in a home game, but my shortcomings are apparent in a casino game with strong players. As I developed SoarBot's rule set, I used the poker literature to supplement my knowledge, but my limited abilities may well be a key factor in SoarBot's mediocre play.

On the plus side, SoarBot is a straightforward rule-based system, developed by one person in less than a year. The CPRG bots reflect several years of work by several developers and researchers, integrating several artificial intelligence techniques into one program (§§ 1.4.1 to 1.4.5). From this perspective, the gap between SoarBot and the CPRG bots is less daunting; it's possible that, with further development, SoarBot might bridge that gap.

5. Opportunities for Further Development

5.1 Move Hand Analysis Out Of the Rule Set

As I discussed in section 2.2.4, poker hand analysis is not something that a rule-based system does well. Moving it into either the Java or Tcl layers would make SoarBot simpler, and the hand analysis would be more precise - a win-win modification.

It would also be worth extending the analysis to include more advanced poker concepts like implied pot odds, "...the possibility of winning money in later betting rounds over and above what is in the pot already" (Sklansky 55), and outs, the number of "...cards, that, if drawn, will give you the winning hand" (Bellin 254).

5.2 Opponent Modeling

SoarBot's most glaring weakness is its inability to model its opponents. In poker, opponent modeling is the heart of the matter:

```
The art of poker is filling the gaps in the incomplete
information provided by your opponent's betting and
the exposed cards in open-handed games, and at the
same time preventing your opponents from discovering
any more than what you want them to know about your
hand. (Sklansky 17)
```

Such opponent knowledge might include things like:

- Strength of starting cards an opponent plays, broken down by position;
- How often the opponent bluffs;
- How often the opponent check-raises;
- Whether the opponent is a winner or a loser.

As a first attempt, it would be useful simply to collect this information and make it available to the rule set. I started this work with the PlayerHistory.xml file (§ 2.2.10), but it's a long way from useful.

5.3 Learning

One of SoarBot's problems is the consistency of its play. This follows from its static rule set. What's needed, I think, is a set of metarules that tell SoarBot not how to play poker, but how to learn about playing poker.

With such metarules in place, the more SoarBot played, the better it would get. The potential for SoarBot to expand its own poker rule set is exciting. It is also enormously challenging.

Appendix A: Downloading the Source Code

Source code for SoarBot and SoarSession is available at <http://codeblitz.com/poker.html>. Each download includes a readme.txt file with detailed installation instructions

Bibliography

- Addis, T. R. Designing Knowledge-Based Systems. Englewood Cliffs, N.J.: Prentice-Hall, 1986.
- Alvarez, A. The Biggest Game In Town. San Francisco: Chronicle Books, 2002.
- Bellin, Andy. Poker Nation. New York: HarperCollins, 2002.
- Bigus, Joseph P. and Jennifer Bigus. Constructing Intelligent Agents Using Java. 2nd ed. New York: John Wiley, 2001.
- Billings, Darse, et al. "Poker as a Testbed for Machine Intelligence Research." 1998. 3 Sep. 2002
<<http://www.cs.ualberta.ca/~jonathan/Papers/Papers/ai98.poker.html>>.
- Billings, Darse, et al. "The Challenge of Poker." 2001. 9 Apr. 2003
<<http://www.cs.ualberta.ca/~darse/Papers/AIJ-01.ps>>.
- CLIPS: A Tool for Building Expert Systems. 5 Oct. 2002. 9 Apr. 2003
<<http://www.ghg.net/clips/CLIPS.html>>.
- Congdon, Clare Bates and Karl B. Schwamb. "The Soar Advanced Applications Manual, Version 7." 7 Nov. 1995. 9 Apr. 2003
<<http://ai.eecs.umich.edu/soar/docs/manuals/advanced-manual.ps.Z>>.
- Davidson, Aaron. Computer Poker (Texas Hold'em) Java Source Code. No date. 9 Apr. 2003 <<http://spaz.ca/aaron/poker/src/index.html>>.
- Davidson, Aaron. Generated Documentation: Computer Poker (Texas Hold'em) Java Source Code. No date. 12 Apr. 2003
<<http://spaz.ca/aaron/poker/src/docs/>>.
- Davidson, Aaron. "Re: Poker Server." E-mail to the author. 30 Sep. 2002.
- Eaters Download. No date. 10 Apr. 2003
<<http://ai.eecs.umich.edu/soar/getstart/Eaters30.tgz>>.
- Feather 0.1 Home Page. Oct. 1999. 9 Apr. 2003
<<http://www.itl.nist.gov/div897/ctg/java/feather/>>.
- Hutchison, Edward. Texas Hold'Em. No date. 14 Apr. 2003
<<http://www.homestead.com/erh2/HEM.html>>>

Jackson, Peter. Introduction to Expert Systems. Wokingham, England: Addison-Wesley, 1986.

JDOM. No date. 9 Apr. 2003 <<http://www.jdom.org/>>.

Jess, the Expert System Shell for the Java Platform. 8 Apr. 2003. 9 Apr. 2003 <<http://herzberg.ca.sandia.gov/jess/>>.

JNI - Java Native Interface. 2002. 10 Apr. 2003 <<http://java.sun.com/j2se/1.4.1/docs/guide/jni/>>.

Koller, Daphne and Avi Pfeffer. "Representations and Solutions for Game-Theoretic Problems." 16 Apr. 1997. 9 Apr. 2003 <<http://robotics.stanford.edu/~koller/papers/galapaper.ps>>.

Laird, John E. "The Soar 8 Tutorial." 23 Feb. 2001. 9 Apr. 2003 <http://ai.eecs.umich.edu/soar/tutorial/Soar_Part1.pdf>, <http://ai.eecs.umich.edu/soar/tutorial/Soar_Part2.pdf>, <http://ai.eecs.umich.edu/soar/tutorial/Soar_Part3.pdf>, <http://ai.eecs.umich.edu/soar/tutorial/Soar_Part4.pdf>.

Laird, John E., Clare Bates Congdon, and Karen J. Coulter. "The Soar 8 User's Manual, Version 8.2". 23 June 1999. 9 Apr 2003 <<http://ai.eecs.umich.edu/soar/docs/manuals/soar8manual.pdf>>.

Lehman, Jill Fain, John Laird, and Paul Rosenbloom. "A Gentle Introduction to Soar, an Architecture for Human Cognition." No date. 9 Apr. 2003 <<http://ai.eecs.umich.edu/soar/docs/Gentle.pdf>>.

LISA Homepage. 19 Dec. 2002. 9 Apr. 2003 <<http://lisa.sourceforge.net/>>.

Martin, James and Steven Oxman. Building Expert Systems: A Tutorial. Englewood Cliffs, N.J.: Prentice-Hall, 1988.

Online Poker Protocol. 28 Jan. 2003. 9 Apr. 2003 <<http://games.cs.ualberta.ca/webgames/poker/bots.html>>.

PokerSource.org. No date. 14 Apr. 2003 <<http://pokersource.sourceforge.net>>.

Russell, Stuart J. and Peter Norvig. Artificial Intelligence: A Modern Approach. New Jersey: Prentice Hall, 1995.

Schaeffer, Jonathan, et al. "Learning to Play Strong Poker." 1999. 3 Sep. 2002 <<http://www.cs.ualberta.ca/~darse/Papers/ML99.html>>.

Schaeffer, Jonathan. "Kasparov versus Deep Blue: The Re-match." 1997. 8 April 2003 <<http://www.cs.vu.nl/~aske/db.html>>.

Silberstang, Edwin. Winning Poker For The Serious Player. New York: Cardoza Publishing, 1992.

Sklansky, David. The Theory of Poker. 4th ed. Henderson, Nevada: Two Plus Two Publishing, 2002.

Sklansky, David and Mason Malmuth. Hold'em Poker For Advanced Players. Henderson, Nevada: Two Plus Two Publishing, 1988.

Soar Home Page. 9 Apr. 2003 <<http://ai.eecs.umich.edu/soar/>>.

Soar Technology. "Soar: A Comparison with Rule-based Systems." 2002. 9 Apr. 2003 <<http://ai.eecs.umich.edu/soar/docs/SoarRBSComparison.pdf>>.

Soar Technology. "Soar: A Functional Approach to General Intelligence." 2002. 9 Apr. 2003 <<http://ai.eecs.umich.edu/soar/docs/SoarFunctionalOverview.pdf>>.

Tcl Developer Site. 21 Nov. 2002. 10 Apr. 2003 <<http://www.tcl.tk/>>.

Tcl Library Manual. 1997. 10 Apr. 2003 <<http://www.tcl.tk/man/tcl8.0/TclLib/contents.htm>>.

University of Alberta Computer Poker Research Group. Dec. 2001. 9 Apr. 2003 <<http://www.cs.ualberta.ca/~games/poker/>>.

University of Alberta GAMES Group. 5 Apr. 2003. 9 Apr. 2003 <<http://www.cs.ualberta.ca/~games/>>.

Welch, Brent B. Practical Programming in Tcl and Tk. 3rd ed. New Jersey: Prentice Hall, 2000.