ON USING A COMPUTER TO SOLVE GAME THEORY PROBLEMS: GAME 01 DOMAIN PRUNING BY REVISIONS

Bruce R. Barkstrom

15 Josie Lane
Asheville, NC 28804

1. INTRODUCTION

This text is devoted to implementing multiagent simulations of various sorts for computational economics and related disciplines. However, before we can apply this approach, we need a deeper introduction to some known algorithms in multiagent systems. Accordingly, we will follow many of the cases in Shoham and Leyton-Brown [2009], taking the unadorned algorithm description in their text and converting it into working computer code. In a sense, then, this text consists of a set of exercises based on that reference.

When we implement the solution to these exercises, we will use Ada95 or Ada2005. This language is a sensible choice because the agent-to-agent interactions are already supported by the Ada *rendezvous* mechanism. The agents in these exercises will be Ada *tasks*, which are roughly equivalent to threads of computation in modern computer architectures. Indeed, Ada tasks are usually implemented as threads. Thus, this kind of simulation fits well with recent developments in multi-core CPU architectures. Ada also provides an operating-system-independent method of writing code. This independence means that a developer can transfer source code from one computer to another without modifying it and have the code operate in the same way after it is compiled and linked, assuming that each system has an Ada compiler.

We also use a Unified Modeling Language (UML) design tool to conceptualize and then design each exercise's implementation. In these designs, each agent becomes an object. To initialize each agent or to record the history of the interactions between the agents, we also need to introduce a "Referee" and a "Scorekeeper." The whole game concept becomes very similar to designing a community of interacting people who can do things (some, of course, more interesting than others).

In the sections that follow, we also follow a fairly standard design process:

1. Conceptualize the nature of the game, discovering likely agents
2. Write out the use cases that describe the most important phases of the design
3. Derive the detailed data structures and algorithms for each agent, including translation of the algorithm sketches in the sources
4. Create the Ada code
5. Test the code for proper results

2. GAME 01: DOMAIN-PRUNING BY REVISION

Shoham and Leyton-Brown's first example [pp. 3-5] identifies a system with n "variables," $X = X_1, \ldots, X_n$. Figures 1.3 and 1.4 in their text show cases where $n = 3$. The term "variable" seems rather a bit of a misnomer. "Agent" would probably better describe the role of these entities in the algorithm. Oddly, the book's index does not include the term "agent," (although it does include "agent-based simulation"). The closest one might come to a definition of the term "agent" in the book's context is connected with a highly restricted meaning associated with "speech acts."

It might be helpful to think of this algorithm as involving automated radio transmitters and receivers (transceivers) to allocate three transmission frequencies amongst three pieces of

equipment bought by a town government. The town has a choice of three different models. The first is quite expensive. It comes with three different transmitting frequencies. A second model is somewhat less expensive, having only two frequencies. The third is cheapest, having only one frequency. The manufacturer has only three frequencies available. However, if a tranceiver has more than one oscillator, it can select which one it will use to transmit. If two tranceivers attempt to broadcast on the same frequency, they interfere with each other. In this case, no communication is possible. Each model also comes with a modest computer capability that can send a signal to other pieces of equipment and discover the frequencies at which the other can transmit. Thus, the point of the algorithm is to find a transmission frequency for each of the three tranceivers such that all three can transmit while receiving signals from the other two.

3. DISCOVERING LIKELY AGENTS AND SOME OF THEIR PROPERTIES

In accord with this discussion, we assume that an "agent" in our simulation games is programmable as an Ada *task*. The Ada Reference Manual defines tasks in the following way [Section 9]:

> "The execution of an Ada program consists of the execution of one or more tasks. Each task represents a separate thread of control that proceeds independently and concurrently between the points where it interacts with other tasks. The various forms of task interaction are described in this section, and include:
>
> - the activation and termination of a task;
> - a call on a protected subprogram of a protected object, providing exclusive read-write access, or concurrent read-only access to shared data;
> - a call on an entry, either of another task, allowing for synchronous communication with that task, or of a protected object, allowing for asynchronous communication with one or more other tasks using that same protected object;
> - a timed operation, including a simple delay statement, a timed entry call or accept, or a timed asynchronous select statement (see next item);
> - an asynchronous transfer of control as part of an asynchronous select statement, where a task stops what it is doing and begins execution at a different point in response to the completion of an entry call or the expiration of a delay;
> - an abort statement, allowing one task to cause the termination of another task.
>
> In addition, tasks can communicate indirectly by reading and updating (unprotected) shared variables, presuming the access is properly synchronized through some other kind of task interaction.
>
> The properties of a task are defined by a corresponding task declaration and task body, which together define a program unit called a task unit.
>
> Over time, tasks proceed through various states. A task is initially inactive; upon activation, and prior to its termination it is either blocked (as part of some task interaction) or ready to run. While ready, a task competes for the available execution resources that it requires to run."

3.1 Agents

So, what agents do we need in order to create a simulation of the first Domain-pruning algorithm? Clearly, we're going to need Players – three to be exact. The text description doesn't identify who keeps score and who decides the game is over. It would be interesting to see if each Player could decide that on its own. However, it isn't easy to see how that would be done. Thus,

we'll create another agent, a "Referee" who can make sure the Players are ready to start and who will decide that the game is over. In addition, because we're doing an example, we want to see how the game has progressed. That means we need an agent to keep track of the game's history. We'll call this agent the Scorekeeper.

At this point it seems like we have five agents:

1. An array of three *Players* (who are called "variables" in the text)
2. A *Referee* who starts the game and ensures it finishes in accordance with the rules
3. A *Scorekeeper* who records the moves of each play as the game progresses and writes out these moves so we can see the history of play

For now, we'll assume these five agents are sufficient.

3.2 Agent Properties

Even at this early point in designing the game, we can see some of the attributes the agents will need in order to carry out the game.

As a precursor, we note that the text assigns names to the possible frequencies the Players can use for transmitting: Red, Green, and Blue. We'll call these 'Possible_Choices'. We'll represent these choices as an Ada enumerated type (meaning that the language assigns names to what is essentially an index).

Of course, each Player can start with a transmitter for one frequency, two frequencies, or three frequencies. As the Pruning algorithm proceeds, of course, the possible selections decrease. In the event that two Players have incompatible demands for a transmitting frequency, it's even possible that a Player will have no possible choices. This freedom within the constraint of having only three frequencies available suggests that we represent a Player's possible selection of frequencies as an array of Boolean values, where the index of the array is chosen from the possible frequencies. If the value for a chosen frequency is 'False', then that selection is not available to the Player. If it is 'True', then that selection is available.

Next, we note that for the game to work, each Player has to communicate with the other Players. If we think of the Players as coming in an array of three, then we need to provide each Player with a list of the other available Players with whom to communicate. As play proceeds, each Player simply goes down his own list of who to communicate with next, obtaining the selection of frequencies available to each other player, in turn.

Listing 1 shows the types that belong to each Player. We have assigned a maximum number of players and set it equal to three, since there are only three frequencies (or colors) assigned in the text.

```
type Possible_Choices is (Red, Green, Blue);
type Choice_Selection is array (Red .. Blue) of Boolean;
--
Max_No_Of_Players      : constant natural
  := 3;
type Player_Communications is array (1 .. Max_No_Of_Players - 1) of natural;
```

Listing 1. *Types for Players.*

At this point, we don't have a clear picture as to whether the Referee or the Scorekeeper have any particular attributes associated with them.

4. CREATING USE CASE SCENARIOS AND USE CASES

Use case scenarios are "stories" that describe the interaction between various entities in the simulation. *Use cases* are more formalized descriptions of these stories. What we're after is

capturing the kinds of interactions we expect in the game, and then moving the description toward a design that leads to computer code.

It sounds like there are three phases in the game:

- **Play**, in which each Player contacts the other two and may prune his own frequencies if those frequencies would interfere with the others
- **Writing Out the History**, in which the Scorekeeper writes out the record of play
- **Initializing Each Agent**, probably based on input from outside the simulation

Of course, this description doesn't place them in proper order. However, the necessary reordering is quite simple. For now, let us look at them in the order presented.

4.1 Play

The text doesn't give explicit instructions about how play proceeds. We just know that each Player needs to get the frequency selections available to each of the other Players, compare its selection, and revise its own selection if the other Player has an overlapping selection.

4.1.1 The pruning procedure

Listing 2 provides an appropriate procedure that accepts the selections from a given Player and the selections from another Player and prunes the given Player's selections for consistency. The procedure returns the pruned selections. It also returns a score indicating either 'No Change' (meaning no pruning was necessary), 'Pruned', or 'Empty Selection', meaning that there was no consistent possibility between the given Player and the other. Listing 2 shows an Ada instantiation of this procedure.

This listing starts with an Ada enumerated type that shows the possible result of the procedure.

The procedure interface specification shows four variables:

- `This_Player_Selection`, an input variable from the Player checking for consistency
- `Other_Player_Selection`, an input variable from the Player whose selections will govern the consistency check
- `Revised_Selection`, an output variable that contains the appropriate revisions
- `Result`, an output variable that summarizes the nature of the change

The procedure has two parts. The first part determines if `This_Player` has overlapping selections with the `Other_Player`. The second part prunes the selections for `This_Player` if that is prudent. Based on the description in the text, prudent pruning requires checking for the number of selections each Player in the comparison has.

Obviously, at the end of the first part, the procedure knows how many selections each Player has, as well as how many overlaps there are. It also knows which of the possible choices do, in fact, overlap.

In the second part, there is clearly no need to prune `This_Player`'s selection if there isn't any overlap. However, if there is some overlap, the procedure needs to see if it is prudent to prune that selection.

There are three checks for prudence:

1. If `This_Player` has more choices than the other player, then there's no problem in removing the overlaps. `This_Player` will still have at least one selection. The `Result` will clarly be a `Pruned` selection for `This_Player`.
2. If `This_Player` and the `Other_Player_Selection` both have one selection – and if they overlap, then the selection choices cannot be consistent. The `Result` has to be an `Empty_Selection`.
3. If neither of these choices are true, then it's imprudent to do anything. The `Result` is `No_Change`.

```
type Consistency_Result_Type is (No_Change, Pruned, Empty_Selection);
--
procedure Prune_For_Consistency(This_Player_Selection  : in     Choice_Selection;
                                Other_Player_Selection : in     Choice_Selection;
                                Revised_Selection      :    out Choice_Selection;
                                Result                 :    out Consistency_Result_Type)
is
   Overlap                          : array (Red .. Blue) of Boolean;
   Number_Of_Overlaps               : natural;
   Number_Of_This_Player_Selections  : natural;
   Number_Of_Other_Player_Selections : natural;
begin
   --
   -- Determine Original State of Agreement Between This Player and the Other One
   --
   Number_Of_Overlaps := 0;
   Number_Of_This_Player_Selections := 0;
   Number_Of_Other_Player_Selections := 0;
   Overlap := (Red => False, Green => False, Blue => False);
   for Possible_Choices in Red .. Blue loop
      if This_Player_Selection(Possible_Choices) then
         Number_Of_This_Player_Selections := Number_Of_This_Player_Selections + 1;
      end if;
      if Other_Player_Selection(Possible_Choices) then
         Number_Of_Other_Player_Selections := Number_Of_Other_Player_Selections + 1;
      end if;
      if This_Player_Selection(Possible_Choices)
        and Other_Player_Selection(Possible_Choices) then
         Overlap(Possible_Choices) := True;
         Number_Of_Overlaps := Number_Of_Overlaps + 1;
      end if;
   end loop;
   --
   -- Prune Selections if Prudent
   --
   if Number_Of_Overlaps = 0 then
      --
      --   No Need to Remove Selections For This Player
      --
      Revised_Selection := This_Player_Selection;
      Result := No_Change;
   else
      if Number_Of_This_Player_Selections > Number_Of_Other_Player_Selections then
         --
         --    Prudent Removal of Overlapped Selections
         --      The Other Player has fewer selections than This Player does.  Thus,
         --      This Player will still have some selections after removing overlaps.
         --      This section does not drop the number of selections in the Revised
         --      Selection below 1.
         --
         -- Remove overlapped slections from This_Players selections
         --
         Revised_Selection := This_Player_Selection;
         for Possible_Choices in Red .. Blue loop
            if Overlap(Possible_Choices) then
               Revised_Selection(Possible_Choices) := False;
               Result := Pruned;
            end if;
         end loop;
```

Listing 2. *Prune for Consistency Procedure.*

```
      elsif Number_Of_This_Player_Selections = 1 and Number_Of_Other_Player_Selections = 1 then
        if Number_Of_Overlaps = 1 then
           --
           --   Make This Player Selections be EMPTY
           --
           Revised_Selection := (Red .. Blue => False);
           Result := Empty_Selection;
        end if;
      else
        --
        --   No Need to Remove Selections For This Player
        --
        Revised_Selection := This_Player_Selection;
        Result := No_Change;
      end if;
    end if;
  end Prune_For_Consistency;
```

Listing 2 (cont'd.). *Prune for Consistency Procedure.*

4.1.2 Procedural rules for play

The main text provides few (or no) procedural rules for how play should proceed.

We do know that we would like the Scorekeepeer to keep track of each pruning attempt and its results. We could write that record out to a file as each result comes in, or we could let the Scorekeeper accumulate the results for each attempt and then write the whole record of the game out at the end. It seems reasonable to only open up the results file at the end, particularly since the game is likely to be very short.

This approach still means that at the end of each pruning attempt, the Player being pruned will need to send the Scorekeeper a record that contains such items as the original selection of frequencies for the Player, the current selection of frequencies for the Other Player, the pruned selection, and the result (which is like a score for the play). Listing 3 shows such a record type.

```
type Transaction_Type is record
   Round                  : natural;
   Player_ID              : natural;
   Other_Player_ID        : natural;
   Original_Selection     : Choice_Selection;
   Other_Player_Selection : Choice_Selection;
   Revised_Selection      : Choice_Selection;
   Result                 : Consistency_Result_Type;
end record;
```

Listing 3. *Transaction Type passed from Player to Scorekeeper after each call to* `Prune_for_Consistency`.

The **procedure** Revise on p. 4 of the main text does not specify a particular order in which players should check for pruning. It also doesn't contain an agent like the Scorekeeper who is responsible for recording the outcome. It doesn't include the agent we've called the Referee, who is presumably responsible for starting play and terminating it.

Thus, it is up to the author of these exercises to invent the communication design for the plays. If we think of games with humans, we might think of dividing up the game into *Plays*, during which each Player prunes his frequency selection after interacting with a single Other Player. This would mean that the Scorekeeper would get three Plays in a single group. If we don't allow anymore checking until each Player has provided a score, then we can keep the game fairly well-organized.

If there is a second group of three Plays, in which each Player prunes his frequency selections with the Player he didn't check with the first time, then we can call the pair of three plays a *Round*. At the end of a Round, the Scorekeeper can check in with the Referee and give him

(or her) the score for the Round. If the score is `Pruned`, meaning that at least one player pruned his selection, then the Referee can signal to the Scorekeeper to start another Round. On the other hand, if at least one of the Players has had a pruning with the Result `Empty_Selection` or `No_Change`, then play should terminate.

Figure 1 provides a UML Sychronization Diagram for the Plays within a Round. Play starts with the Referee, who sends a message to the Scorekeeper to Start the Round. The Scorekeeper synchronizes the interaction between the Players and collects the score at the end of each Player's call to the Pruning procedure. Note that this figure only shows the interaction in which Player 1 requests and receives the frequency selection of Player 2. During the actual computation, Player 2 does the same with respect to Player 1 and Player 3 also requests the same information from Player 1. The Scorekeeper does not specify the timing of these calls. As a result, the actual message sequence seen by any particular Player may be different than it might seem from this figure. This implementation factor means that the frequency selection of any particular Player may be pruned at seemingly random times – and may have a different sequence of changes in different runs of the program.

Figure 1 also does not show the second group of three prunings that completes the Round. However, the key message at the end of the Round is the one that goes from the Scorekeeper to the Referee passing the score for the Round to him. Thereafter, the Referee decides whether the game is over (because the score has no prunings or has discovered an inconsistent selection between two of the Players) or not. If the game is not over, the Referee sends another `Start_New_Round` message to the Scorekeeper to repeat the same pattern of activity in a new Round.

From the standpoint of actually coding a simulation of this algorithm, the Synchronization Diagram in fig. 1 is a very helpful design artifact because it identifies the **Entry** points for each task. For example, the first message that goes from the Referee to the Scorekeeper needs an Entry in the Scorekeeper task that will **accept** the call and provide the Scorekeeper with a new Round value (meaning an incremented integer). The second (as well as the third and fourth) message is an event for Player 1 (and then Player 2 and Player 3) that initiates the calls to the other Player tasks, which each Player task follows with a call to the Prune procedure. After Player 1 has completed the Prune call, it sends the total Transaction message to the Scorekeeper, who duly records it in an array. Players 2 and 3 perform the equivalent actions.

After each Player has sent the score to the Scorekeeper, that Player goes into a waiting state until it receives a new Start_New_Play message from the Scorekeeper. The Player awakens from that waiting state only after receiving that message.

Figure 2 shows a statechart description of the various states and transitions a Player can have. For those not familiar with these diagrams, the intial state of this agent is in the unadorned black circle. That's where you start tracing how the Player's state evolves.

The Player moves from its initial state by accepting input (from the Referee) based on the input text file. That interaction provides the initial frequency selection and the list of other Players to contact. After that information is digested, the Player is initialized and moves into a state in which it can participate in playing the game.

The main state of each Player consists of two substates: Waiting and Playing. After Initializing, the Player enters the `Waiting` substate until the Scorekeeper sends it a `Start_Play` signal. As the diagram shows, after the Player enters the `Playing` substate, it sends a signal to another Player to obtain that other Player's state. After pruning its own selection, the Player sends the score to the `Provide_Score` entry of the Scorekeeper and makes a transition back to the `Waiting` substate.

We note that the `Provide_Selection` state and the `Terminating` states are separate from the `Initializing` and the `Participate_In_Play` state. We'll deal with initialization in a bit. There really isn't much for a Player to do when it's terminating.

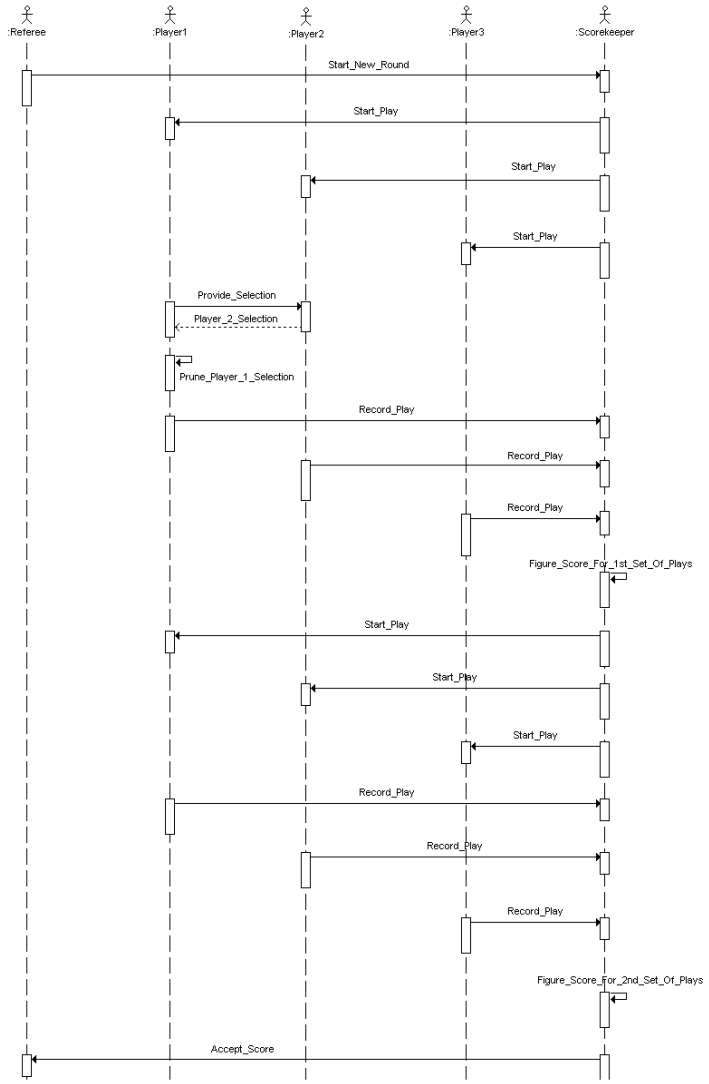

Figure 1. Synchronization Diagram for Proposed Play Procedure in a Single Round *Referee starts a round of play by contacting the Scorekeeper, who then signals each Player to start a single call to the* `Prune_for_Consistency` *procedure. This figure shows only one such call; in the program, all three Players have their own calls. The Scorekeeper receives the results from each of the three Players before sending a new message to each Player to contact the Player not previously checked for consistency. After each Player has sent the additional score to the Scorekeeper, the latter agent provides a score for the Round to the Referee. The Referee considers whether the game is over or not.*

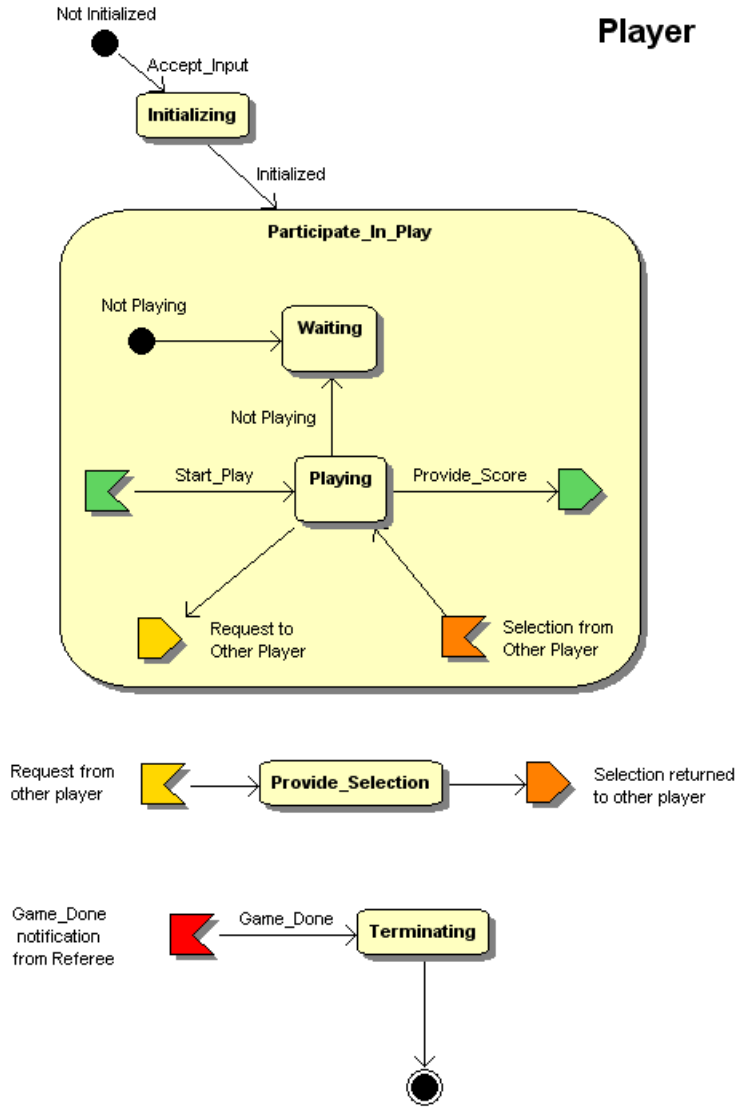

Figure 2. Player State Chart *This state chart shows the various internal states of a Player. When play starts, the Player is not Initialized. After Initialization occurs, there is a transition to a Waiting state. When the Player receives the* `Start_Play` *signal from the Scorekeeper, the Player makes a transition to a Playing state. In this state, the Player asks another Player for its selection of frequencies. The other Player makes a transition to the state in which it provides that selection. When the selection returns, the Player prunes its selection and then returns the score to the Scorekeeper. When the game is ready to end, the Player receives a* `Game_Done` *signal from the Referee and transitions to a* `Prepare_To_Terminate` *state, after which it terminates.*

4.2 Writing Out the History

When we think of a game history, one part is contained in the array of transitions that the Scorekeeper maintains. Another part of the history is a record of the events that cause each agent to change state. In the program we deal with these separately.

4.2.1 Play history

In a sequential program, we could count on lines of text being output in the order in which they're called from the single sequence of control. In a concurrent program, we can't count on that sequencing. The safer approach is to open the file, write all of the output, and then close the file.

As a result, we have the Scorekeeper accumulate all of the scores in an in-memory array while the Players are still playing. Once the Referee decides the game has ended, then it's safe to have the Referee call the Scorekeeper to write out the history of the game as part of the Scorekeeper's preparation to terminate. When the Scorekeeper finishes with the writing, this agent can send a message back to the Referee that the Scorekeeper is ready to terminate.

In a bit more detail, if the Scorekeeper maintains the number of score transactions that it has received previously, it can simply increment that number by one when a new one arrives and tuck the new arrival into the transaction array at the new number. When the time comes to write the history out to a file, the Scorekeeper can loop through the array from 1 to the last number of transactions, writing each transaction out to the file.

Output from Scorekeeper
Number of Transactions : 12

Round	Transaction	Player Index	Original Selection	R	G	B	Other Player Index	Other Player Selection	R	G	B	Revised Player Selection	R	G	B	Results
1	1	1		T	F	F	2		T	F	F		T	F	F	N
1	2	2		T	T	F	1		F	T	F		F	T	F	P
1	3	3		T	T	T	1		F	T	T		F	T	T	P
1	4	1		T	F	F	3		T	F	F		T	F	F	N
1	5	2		F	T	F	3		F	T	F		F	T	F	N
1	6	3		F	T	T	2		F	F	T		F	F	T	P
2	7	1		T	F	F	2		T	F	F		T	F	F	N
2	8	2		F	T	F	1		F	T	F		F	T	F	N
2	9	3		F	F	T	1		F	F	T		F	F	T	N
2	10	1		T	F	F	3		T	F	F		T	F	F	N
2	11	2		F	T	F	3		F	T	F		F	T	F	N
2	12	3		F	F	T	2		F	F	T		F	F	T	N

Figure 3. Results Tabulation for Game With Initialization Corresponding to a Setup Similar to that Fig. 1.4 (a) of Shoham and Layton-Brown *This table shows the history of a game whose initial conditions has Player 1 with red, Player 2 with red and green, and Player 3 with red, green, and blue.*

Figure 3 shows the output from this writing, using a spreadsheet to help format the text file from a single run. At the top of the output, there is a brief note on the source of the output, followed by a line with the total number of transactions in the game. The remainder of the output is a tabulation of the transactions. The leftmost column contains the round number. In this case, there are only two rounds. We also recall that each round for this game will contain six individual plays, one for each Player. The second column from the left is the transaction, which is also the array index within the Scorekeeper.

The central part of the table has the frequency selections for the Player and for the other Player whose selections create the pruning conditions. We can see the original selections of the pruning Player, followed by the selections of the other Player. The four columns on the right contain the pruned selections of the original player and the result value for the pruning process. To visualize the sets of three plays, we have inserted a blank row in the spreadsheet output.

The game's initial condition is very similar to the one in fig. 1.4 (a) of the main text – except that we've switched green and blue in the initial conditions. From the first results row, we can

see that the pruning process does not change the first Player when it checks with the frequency selection of Player 2. On the other hand, we can see that the pruning process removes the red frequency from the second and third player's selections when they check on the selections of Player 1. Note that at the end of the first round of play (with all six possible interactions between Players) there is a unique selection for each Player.

In the second round, we find that there are no prunings. The frequency selection of each Player is stable and unchanging when compared with the frequency selection for each of the other players. As a result, the referee terminates the game.

4.2.2 Event history

As part of checking on how the program is proceeding through its activities, the programmer can embed "PRINT" statements throughout the souce code. In a sequential program, this is one method of debugging. The output allows the user to check on numerical values as well as the proper sequencing of statements. This is not so straightforward for a concurrent (or multi-threaded) program.

Listing 4 shows a text file created by statements that print notes that identify the state of the agents at various points in the computation. The first line identifies the executable, together with the input file name and the output file name. The second and third lines show a verification of these names output from the main program, which is named Game01. The fourth line of this listing is likely to be surprising: it suggests that the main program has ended before any of the other agents has even started to notify output of anything they're doing. Of course, that just means that the main program has reached the end of the items that it directly controls. It actually waits until the other agents have terminated before terminating itself.

After the game had produced the text listing, we augmented it slighly by adding four lines that help distinguish segments of activities. Segment 1 deals with initialization of each agent. The Scorekeeper doesn't have much to record. However, each player has to receive its list of other agents and its initial selection of frequencies. At the end of this segment of play, the listing shows that the Referee had received a notice that each of the other agents were ready to play.

Segment 2 of play contains the activities and events in round 1. When the Referee sends out the message to the Scorekeeper that play should begin for round 1, the Scorekeeper provides a text line to standard output noting that the message has been received. The bulk of this segment shows the interaction between the Scorekeeper and the three Players, with the Player messages noting the index of the other Player with whom they are interacting. At the end of the segment, we see that score that the Scorekeeper has passed to the Referee.

Segment 3 is essentially the same as segment 2, except that none of the Players has to change their selection.

Segment 4 has a more interesting output. We recall that the Scorekeeper is charged with creating an output file before it can terminate. The fourth line of segment 4's listing shows that standard output has started to output a message from Player 1's termination – except that the Scorekeeper finishes the output file and then interrupts the Referee's output with a message that the `Scorekeeper_is_Prepared_to_Terminate`. Interruptions are one of the difficulties (or perhaps the major difficulty) in understanding how a concurrent program is proceeding. Listing 4 does not contain a mistake – but the output may be misleading. We comment in the concluding section of this chapter on ways of overcoming this difficulty.

4.3 Initializing the Game

The Referee has the responsibility for reading in the values each agent needs to have to operate during the game. Listing 5 shows the text file we have used for this purpose.

While a number of formats could have been used, it seemed sensible to keep the input readable by humans, while still avoiding the programming complexity of XML. From the author's point of view, this is intended to encapsulate a certain amount of common sense about how easy

```
Game01.exe Game01_Input.txt Game01_Output.txt
Input_File_Name   : Game01_Input.txt
Output_File_Name  : Game01_Output.txt
Main procedure is done with Game01
-- [Segment 1: Agent initialization] ------------------------------------------
Scorekeeper received Output_File_Name : Game01_Output.txt
From Player   1
   Permission to Contact Player (  1)   2 (  2)   3
   Initial Color Choices Red T Green F Blue F
From Player   2
   Permission to Contact Player (  1)   1 (  2)   3
   Initial Color Choices Red T Green T Blue F
From Player   3
   Permission to Contact Player (  1)   1 (  2)   2
   Initial Color Choices Red T Green T Blue T
Referee received note that Scorekeeper with ID   0 is ready to play
Referee received note that Player      with ID   1 is ready to play
Referee received note that Player      with ID   2 is ready to play
Referee received note that Player      with ID   3 is ready to play
-- [Segment 2: Round 1 of play] -----------------------------------------------
Referee has received notification that all players are ready
Start Round 1
Scorekeeper Ready to Start Round   1
Player   1 starting play with Allowed_Players_For_Comm(Other_Player_Index)   2
Player   2 starting play with Allowed_Players_For_Comm(Other_Player_Index)   1
Player   3 starting play with Allowed_Players_For_Comm(Other_Player_Index)   1
Scorekeeper Play Recorded for Received_Player_Index   1
Scorekeeper Play Recorded for Received_Player_Index   2
Scorekeeper Play Recorded for Received_Player_Index   3
Player   1 starting play with Allowed_Players_For_Comm(Other_Player_Index)   3
Player   2 starting play with Allowed_Players_For_Comm(Other_Player_Index)   3
Player   3 starting play with Allowed_Players_For_Comm(Other_Player_Index)   2
Scorekeeper Play Recorded for Received_Player_Index   1
Scorekeeper Play Recorded for Received_Player_Index   2
Scorekeeper Play Recorded for Received_Player_Index   3
At end of Round   1
Referee received SCORE : Pruned
-- [Segment 3: Round 2 of play] -----------------------------------------------
Scorekeeper Ready to Start Round   2
Player   1 starting play with Allowed_Players_For_Comm(Other_Player_Index)   2
Player   2 starting play with Allowed_Players_For_Comm(Other_Player_Index)   1
Player   3 starting play with Allowed_Players_For_Comm(Other_Player_Index)   1
Scorekeeper Play Recorded for Received_Player_Index   1
Scorekeeper Play Recorded for Received_Player_Index   2
Scorekeeper Play Recorded for Received_Player_Index   3
Player   1 starting play with Allowed_Players_For_Comm(Other_Player_Index)   3
Player   2 starting play with Allowed_Players_For_Comm(Other_Player_Index)   3
Player   3 starting play with Allowed_Players_For_Comm(Other_Player_Index)   2
Scorekeeper Play Recorded for Received_Player_Index   1
Scorekeeper Play Recorded for Received_Player_Index   2
Scorekeeper Play Recorded for Received_Player_Index   3
At end of Round   2
Referee received SCORE : No Change
-- [Segment 4: Cleanup and Termination] ---------------------------------------
Player(  1) is Prepared to Terminate
Player(  2) is Prepared to Terminate
Player(  3) is Prepared to Terminate
Referee received note that PlayerScorekeeper is Prepared to Terminate
  1 is terminating
Referee received note that Player  2 is terminating
Referee received note that Player  3 is terminating
Referee received note that Scorekeeper is terminating
```

Listing 4. *A listing of most of the events produced by running the program.*

it is to use the program and how easy it should be to modify the input. While an XML parser is available for Ada, the code library that goes with this parser is (much) bigger than the souce code of the program to solve the problem. Using XML wouldn't add anything to the solution of the problem by a concurrent program, which, after all, is the intent of this exercise. Readers wanting an introduction to XML should look in any bookstore that has a decent computer book section.

```
Number of Agents
3
Agent 1
Communicates with Agents 2 3
Number of Colors 1
Colors are R
Agent 2
Communicates with Agents 1 3
Number of Colors 2
Colors are R G
Agent 3
Communicates with Agents 1 2
Number of Colors 3
Colors are R G B
```

Listing 5. *Text file used for input to set up the game.*

The input text file is primarily dealing with initializing the attributes of the Player agents. Because concurrency may cause indeterminancy in the order in which the agents act, this input allows a user to change that order for each agent. We encourage such experimentation in the exercises that follow.

We also allow each agent to start with a different selection of frequencies. Again, the exercises encourage experimentation on these selections.

4.4 Terminating the Game

At this point, game termination has been almost entirely covered by the material leading to Listing 3.

5. REVIEWING THE GAME

Having worked our way through a complete example of a multiagent program, we can see some of the changes induced by this non-sequential paradigm.

First, visualizing the agents as people who interact by having a purposeful "conversation" seems very helpful in creating a design. There are then three design elements that need to be filled in:

1. Identifying the attributes that carry the decision-making capability for each agent
2. Formalizing the "speech pattern" that the agents carry on as the game or simulation progresses
3. Formalizing the states and transitions that each agent has

Second, translating the agents into code becomes relatively straightforward. Each agent becomes an Ada task. In the second design element, we associate the message passing interactions of the "conversation" between agents as Ada task entry points that use a rendezvous to pass message contents between agents. These conversations may start design informally, using use case scenarios or use cases. For clarification and documentation, UML synchronization diagrams are quite helpful.

Third, UML state charts can provide an additional design clarification for labelling the task entry point logic. Ada accept interfaces can use the state charts to design guards that control when a state is allowed to accept a rendezvous.

Fourth, having a debugging tool that can follow messages from task to task (or agent to agent) is likely to be essential in obtaining a properly working program that invokes concurrent, multi-threaded programming. Printing out "debugging" messages is not adequate for understanding what each agent may be experiencing in the course of running the program.

6. EXERCISES

EXERCISE 1

Create a UML use case scenario for a round of play and place it in the UML documentation Web site for this game, based on the partially filled in one that's provided with this package. This means filling in the Player, Referee, and Scorekeeper attributes needed for the use case, as well as filling in the procedures and functions.

EXERCISE 2

Run the program with all of the input cases provided. Verify that the Ada program properly records the intermediate states and derives the proper end state for the algorithm in each of these cases.

EXERCISE 3

Modify the input text file for the standard game such that the order in which each Player queries the other Players is different than the original test case. Verify that the solution is the same.

EXERCISE 4

Modify the input text file to cover all four cases shown in figure 1.4 of the main text. Verify that the solutions you obtain are sensible.

7. REFERENCES

Shoham, Y. and Leyton-Brown, K., 2009: *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*, Cambridge, Cambridge, UK, 483 pp.