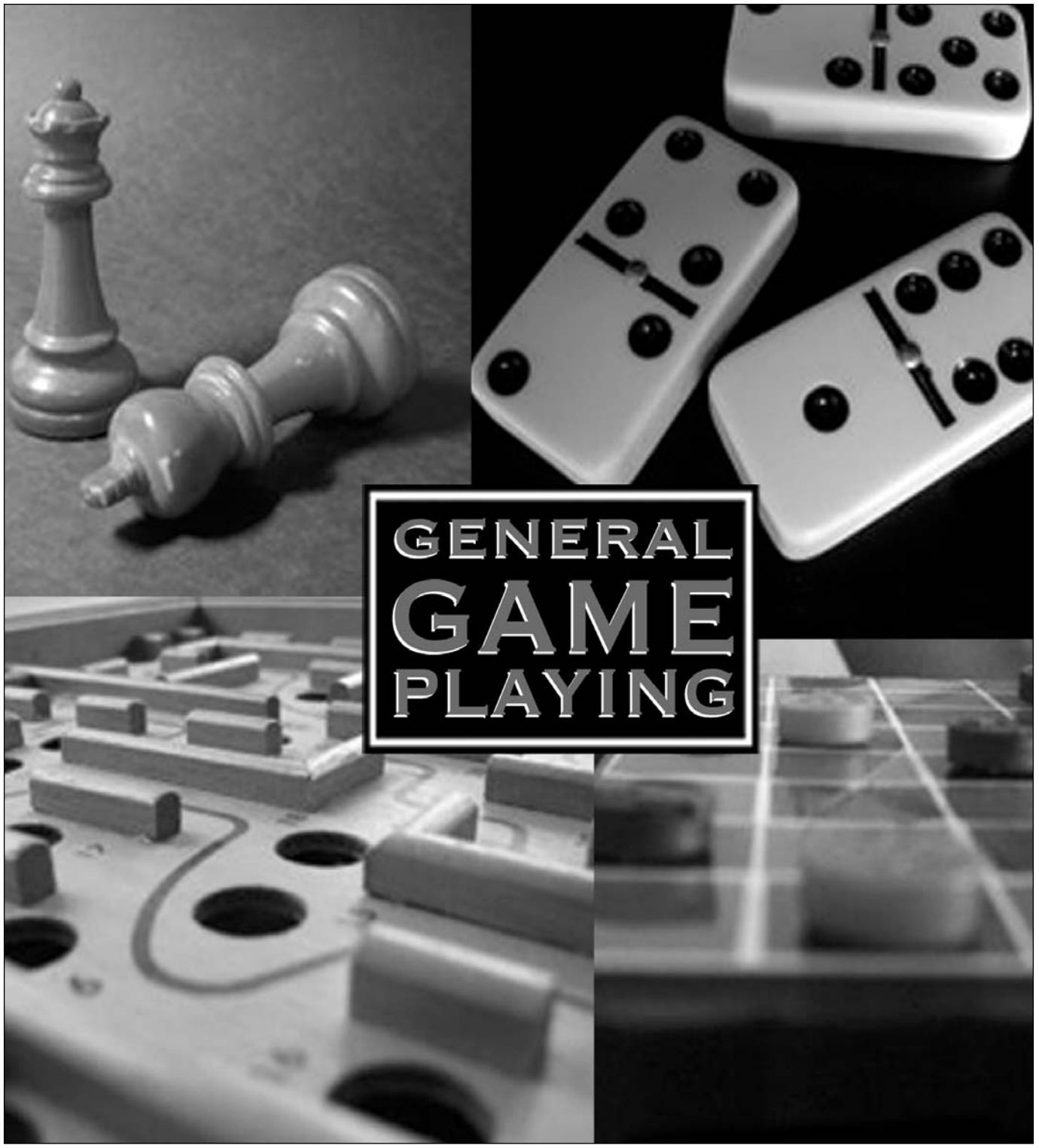


■ A general game playing system is one that can accept a formal description of a game and play the game effectively without human intervention. Unlike specialized game players, such as Deep Blue, general game players do not rely on algorithms designed in advance for specific games; and, unlike Deep Blue, they are able to play different kinds of games. In order to promote work in this area, the AAAI is sponsoring an open competition at this summer's Twentieth National Conference on Artificial Intelligence. This article is an overview of the technical issues and logistics associated with this summer's competition, as well as the relevance of general game playing to the long range-goals of artificial intelligence.

General Game Playing:

Overview of the AAAI Competition

Michael Genesereth, Nathaniel Love, and Barney Pell

Games of strategy, such as chess, couple intellectual activity with competition. We can exercise and improve our intellectual skills by playing such games. The competition adds excitement and allows us to compare our skills to those of others. The same motivation accounts for interest in computer game playing as a testbed for artificial intelligence: programs that think better should be able to win more games, and so we can use competitions as an evaluation technique for intelligent systems.

Unfortunately, building programs to play specific games has limited value in AI. To begin with, specialized game players are very narrow: they focus only on the intricacies of a particular game. IBM's Deep Blue may have beaten the world chess champion, but it has no clue how to play checkers; it cannot even balance a checkbook. A second problem with specialized game-playing systems is that they do only part of the work. Most of the interesting analysis and design is done in advance by their programmers, and the systems themselves might as well be teleoperated.

However, we believe that the idea of game playing can be used to good effect to inspire and evaluate good work in AI, but it requires moving more of the mental work to the computer itself. This can be done by focusing attention on general game playing (GGP).

General game players are systems able to accept declarative descriptions of arbitrary games at run time and able to use such descriptions to play those games effectively (without human intervention). Unlike specialized game players such as Deep Blue, general game players cannot rely on algorithms designed in advance for specific games. General game-playing expertise must depend on intelligence on the part of the game player itself and not just intelligence of the programmer of the game player. In order to perform well, general game players must incorporate various AI technologies, such as knowledge representation, reasoning, learning, and rational decision making; these capabilities must work together in an integrated fashion.

Morever, unlike specialized game players, a general game player must be able to play different kinds of games. It should be able to play simple games (like tic-tac-toe) and complex games (like chess), games in static or dynamic worlds, games with complete and partial information, games with varying numbers of players, with simultaneous or alternating play, with or without communication among the players.

While general game playing is a topic with inherent interest, work in this area has practical value as well. The underlying technology can be used in a variety of other application areas, such as business process management, electronic commerce, and military operations.

In order to promote work in this research area, the AAAI is running an open competition on general game playing at this summer's Twentieth National Conference on Artificial Intelligence. The competition is open to all computer systems, and a $10,000 prize will be awarded to the winning entrant.

This article summarizes the technical issues and logistics for this summer's competition. The next section defines the underlying game model. The "Game Descriptions" section presents the language used for describing games

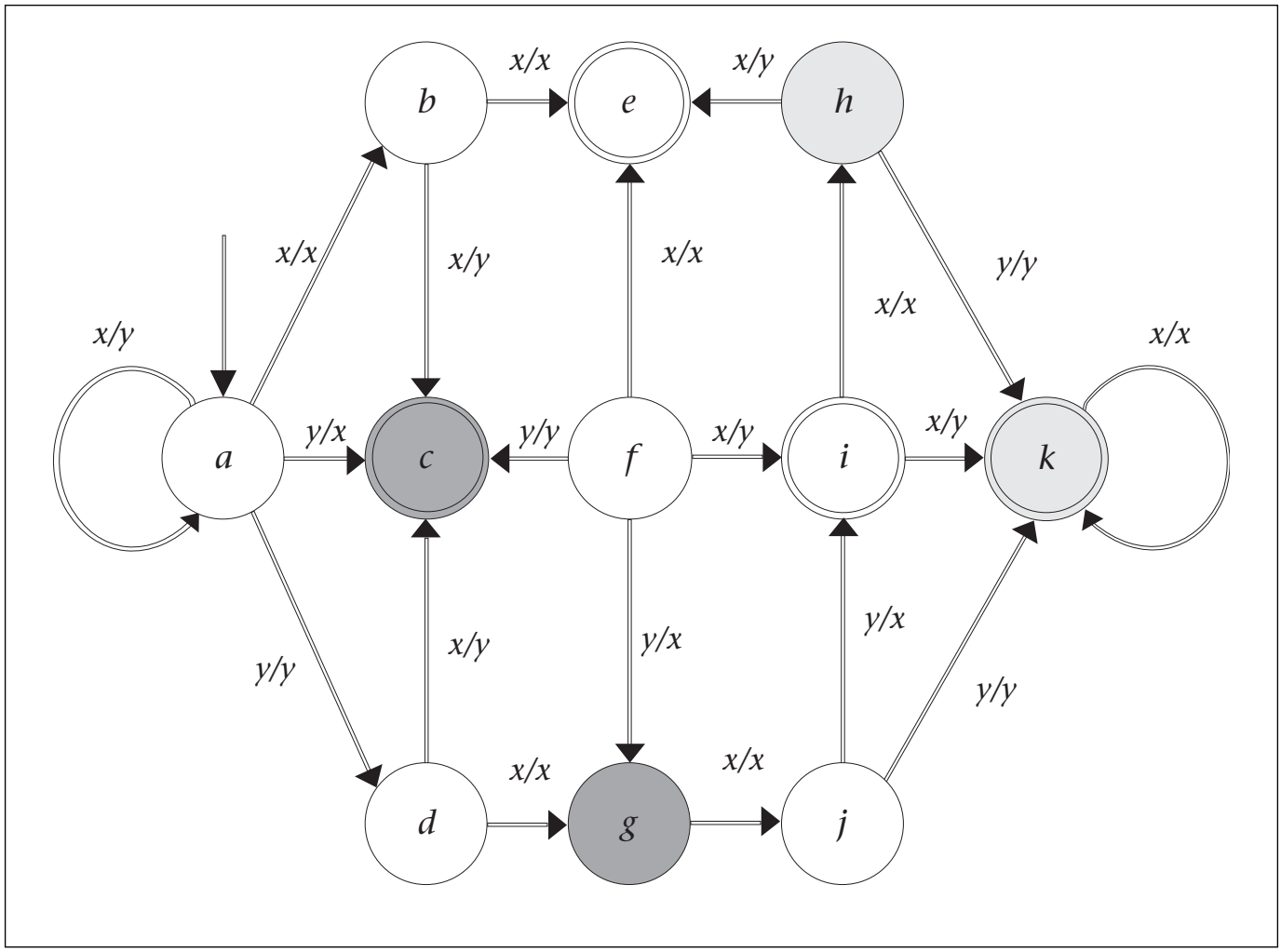


Figure 1. Simultaneous Actions.

according to this model. The "General Game Players" section outlines issues and strategies for building general game players capable of using such descriptions; it is followed by the "Game Management Infrastructure" section, which discusses the associated general game management infrastructure. Our conclusion offers some perspective on the relationship between general game playing and the long-range goals of AI.

Game Model

In general game playing, we consider finite, synchronous games. These games take place in an environment with finitely many states, with one distinguished initial state and one or more terminal states. In addition, each game has a fixed, finite number of players; each player has finitely many possible actions in any game state, and each terminal state has an associated goal value for each player. The dynamic model for general games is synchronous update: all players move on all steps (although some moves could be "no-ops"), and the environment updates only in response to the moves taken by the players.

In its most abstract form, we can think of a finite, synchronous game as a state machine with the following components:

S, a set of game states

$r_1, \ldots, r_n$, the n roles in an n-player game.

$I_1, \ldots, I_n$, n sets of actions, one set for each role.

$l_1, \ldots, l_n$, where each $l_i \subseteq I_i \times S$. These are the legal actions in a state.

n, an update function mapping
$I_i \times \ldots \times I_n \times S \rightarrow S$.

s_1, the initial game state, an element of S.

$g_1, \ldots, g_n$, where each $g_i \subseteq S \times [0 \ldots 100]$.

t, a subset of S corresponding to the terminal states of the game.

A game starts out in state s_1. On each step, each

player r_i makes a move $m_i \in I_i$ such that $l_i(m_i, s_1)$ is true. The game then advances to state $n(m_1, \ldots, m_n, s_1)$. This continues until the game enters a state $s \in S$ such that $t(s)$ is true, at which point the game ends. The value of a game outcome to a player r_i is given by $g_i(s, value)$. Note that general games are not necessarily zero-sum, and a game may have multiple winners. Figure 1 shows a state machine representation for a general game with $S = \{a, b, c, d, e, f, g, h, i, j, k\}$, $s_1 = a$, and $t = \{c, i, k\}$. The shading of states c, g, h, and k indicates that these are highly valued states for two different players of the game (determined by the g_i for those players).

Figure 1 exhibits the transition function n with double arrows labeled with the set of moves made by the players on a step of the game. This is a two player game, and each player can perform actions x or y. Note that it is not the case that every state has an arc corresponding to every action pair: only the legal actions in I_i can be made in a particular state. For example, from state d, one player can legally play both x and y, while the other player's only legal move is x.

This definition of games is similar to the traditional extensive normal form definition in game theory, with a few exceptions. In extensive normal form, a game is modeled as a tree with actions of one player at each node. In state machine form, a game is modeled as a graph, and all players' moves are synchronous. State machine form has a natural ability to express simultaneous moves; with extensions, extensive normal form could also do this, albeit with some added cost of complexity. Additionally, state machine form makes it possible to describe games more compactly, and it makes it easier for players to play games efficiently.

Game Descriptions

Since all of the games that we are considering are finite, it is possible, in principle, to describe such games in the form of lists (of states and actions) and tables (to express legality, goals, termination, and update). Unfortunately, such explicit representations are not practical in all cases. Even though the numbers of states and actions are finite, they can be extremely large; and the tables relating them can be larger still. For example, in chess, there are thousands of possible moves and approximately 10^{30} states.

All is not lost, however: in the vast majority of games, states and actions have composite structure that allows us to define a large number of states and actions in terms of a smaller number of more fundamental entities. In chess, for example, states are not monolithic; they can be conceptualized in terms of pieces, squares, rows and columns and diagonals, and so forth. By exploiting this structure, it is possible to encode games in a form that is more compact than direct representation.

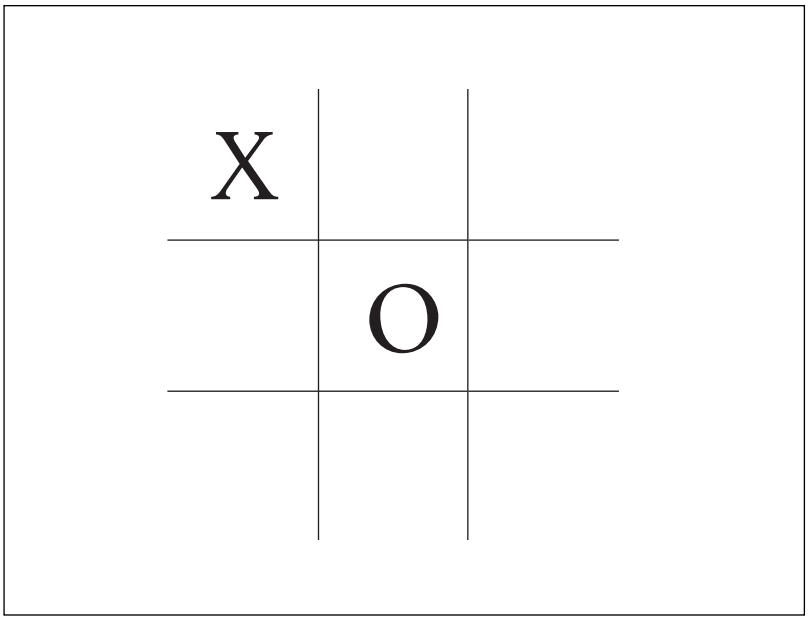


Figure 2. A Tic-Tac-Toe Game State.

Game definition language (GDL) is a formal language for defining discrete games with complete information. GDL supports compact representation by relying on a conceptualization of game states as databases and by relying on logic to define the notions of legality, state update, and so on.

In what follows, we look at a model for games in which states take the form of databases. Each game has an associated database schema, and each state of the game is an instance of this schema. Different states correspond to different instances, and changes in the world correspond to movement among these database instances.

A database schema is a set of objects, a set of tables, and a function that assigns a natural number to each table (its arity—the number of objects involved in any instance of the relationship). The Herbrand base corresponding to a database schema is defined as the set of expressions of the form $r(a_1, \ldots, a_n)$, where r is a relationship of arity n and $a_1, \ldots, a_n$ are objects. An instance of a database schema, then, is a subset of the corresponding Herbrand base.

As an example of this conceptualization of games, let us look at the game of tic-tac-toe. The game environment consists of a 3 by 3 grid of cells where each cell is either blank or con-

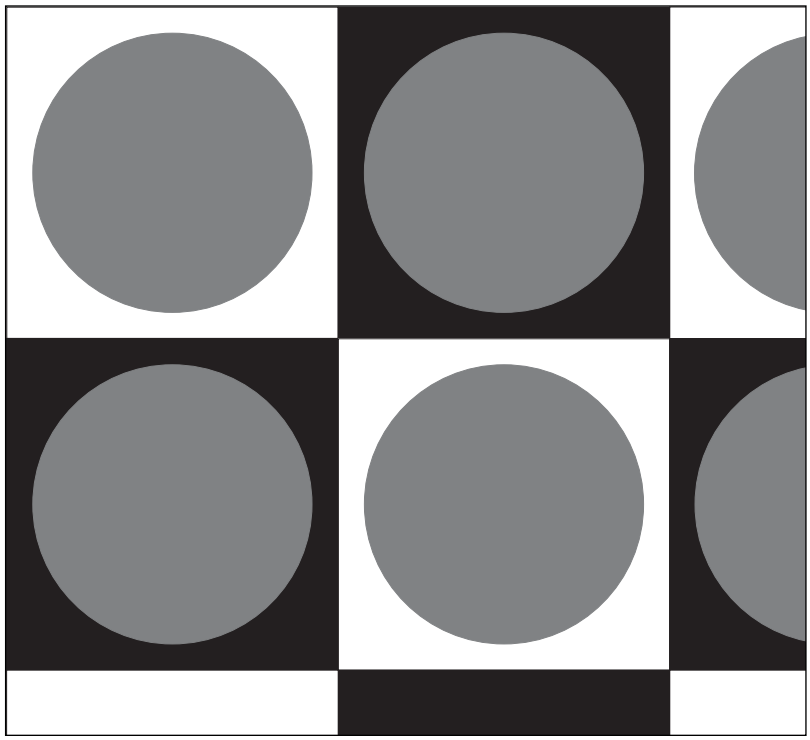
tains an X or an O. Figure 2 portrays one possible state of this game.

To describe game states, we assume a ternary relation *cell* that relates a row number {1, 2, 3}, a column number {1, 2, 3}, and a content designator {*X*, *O*, *B*}. Although it is not strictly necessary, we include an auxiliary relation control that designates the player with the authority to make a mark. The following data encode the game state shown above.

cell(1, 1, *X*)
cell(1, 2, *b*)
cell(1, 3, *b*)
cell(2, 1, *b*)
cell(2, 2, *O*)
cell(2, 3, *b*)
cell(3, 1, *b*)
cell(3, 2, *b*)
cell(3, 3, *b*)
control(*white*)

By itself, the switch from monolithic states to databases does not help. We must still encode tables that are as large as in the state model. However, with the database model, it is possible to describe these tables in a more compact form by encoding the notions of legality, goalhood, termination, and update as sentences in logic rather than as explicit tables.

In our GGP framework, we use a variant of first-order logic enlarged with distinguished names for the key components of our conceptualization of games.



Object variables: *X*, *Y*, *Z*
Object constants: *a*, *b*, *c*
Function constants: *f*, *g*, *h*
Relation constants: *p*, *q*, *r*
Logical operators: ~, |, &, =>, <=, <=>
Quantifiers: *A*, *E*
Terms: *X*, *Y*, *Z*, *a*, *b*, *c*, *f*(*a*), *g*(*b*, *c*)
Relational sentences: *p*(*a*, *b*)
Logical sentences: *r*(*a*, *c*) <= *r*(*a*, *b*) & *r*(*b*, *c*)
Quantified sentences: *p*(*a*, *S*) <= *Ex.q*(*x*, *S*)

GDL uses an indexical approach to defining games. A GDL game description takes the form of a set of logical sentences that must be true in every state of the game. The distinguished vocabulary words that support this are described below:

role(<*r*>) means that <*r*> is a role (player) in the game.

init(<*p*>) means that the datum <*p*> is true in the initial state.

true(<*p*>) means that the datum <*p*> is true in the current state.

does(<*r*>, <*a*>) means that player <*r*> performs action <*a*> in the current state.

next(<*p*>) means that the datum <*p*> is true in the next state.

legal(<*r*>, <*a*>) means it is legal for <*r*> to play <*a*> in the current state.

goal(<*r*>, <*v*>) means that player <*r*> would receive the goal value <*v*> in the current state, should the game terminate in this state.

terminal means that the current state is a terminal state.

distinct(<*p*>, <*q*>) means that the datums <*p*> and <*q*> are syntactically unequal.

GDL is an open language in that this vocabulary can be extended; however, the significance of these basic vocabulary items is fixed for all games.

As an example of GDL, the following demonstrates how the game tic-tac-toe is formalized as a general game. First, we define the roles for the game:

role(*white*)
role(*black*)

Next, we characterize the initial state. In this case, all cells are blank, and white holds control, a term that gains meaning through the descriptions of legal moves that follow.

init(*cell*(1, 1, *b*))
init(*cell*(1, 2, *b*))
init(*cell*(1, 3, *b*))
init(*cell*(2, 1, *b*))
init(*cell*(2, 2, *b*))
init(*cell*(2, 3, *b*))
init(*cell*(3, 1, *b*))
init(*cell*(3, 2, *b*))
init(*cell*(3, 3, *b*))
init(*control*(*white*))

Next, we define legality. A player may mark a cell if that cell is blank and the player has control. Otherwise, so long as there is a blank cell, the only legal action is noop.

```
legal(Y, mark(M, N)) <=
    true(cell(M, N, b)) &
    true(control(Y))

legal(white, noop) <=
    true(cell(M, N, b)) &
    true(control(black))

legal(black, noop) <=
    true(cell(X, Y, b)) &
    true(control(white))
```

Next, we look at the update rules for the game. A cell is marked with an *X* or an *O* if the appropriate player marks that cell. If a cell contains a mark, it retains that mark on the subsequent state. If a cell is blank and is not marked, then it remains blank. Finally, control alternates on each play.

```
next(cell(M, N, x)) <=
    does(white, mark(M, N)) &
    true(cell(M, N, b))

next(cell(M, N, o)) <=
    does(black, mark(M, N)) &
    true(cell(M, N, b))

next(cell(M, N, W)) <=
    true(cell(M, N, W)) &
    distinct(W, b)

next(cell(M, N, b)) <=
    does(W, mark(J, K)) &
    true(cell(M, N, W)) &
    (distinct(M, J) | distinct(N, K))

next(control(white)) <= true(control(black))
next(control(black)) <= true(control(white))
```

A game terminates whenever either player has a line of marks of the appropriate type. The line and open relations are defined as follows.

```
terminal <= line(x)
terminal <= line(o)
terminal <= ~open
```

The following rules define the players' goals. The white player achieves the maximal goal if there is a line of *x*s; the black player does so if there is a line of os. The final termination condition (when the board is full) may be true in a state in which neither player has a line, and this necessitates the goal rules with a value of 50:

```
goal(white, 100) <= line(x)
goal(white, 50) <=
    ~line(x) &
    ~line(o) &
    ~open
goal(white, 0) <= line(o)
goal(black, 100) <= line(o)
goal(black, 50) <=
    ~line(x) &
    ~line(o) &
    ~open
goal(black, 0) <= line(x)
```

A specific game description may require some supporting concepts. In tic-tac-toe, we define a line as a row of marks of the same type or a column or a diagonal. A row of marks means that there are three marks with the same first coordinate. The column and diagonal relations are defined analogously. We also need to define *open*, a condition that holds whenever the board is not yet full of marks.

```
line(W) <= row(M, W)
    line(W) <= column(M, W)
    line(W) <= diagonal(W)

row(M, W) <=
    true(cell(M, 1, W)) &
    true(cell(M, 2, W)) &
    true(cell(M, 3, W))

column(M, W) <=
    true(cell(1, N, W)) &
    true(cell(2, N, W)) &
    true(cell(3, N, W))

diagonal(W) <=
    true(cell(1, 1, W)) &
    true(cell(2, 2, W)) &
    true(cell(3, 3, W))

diagonal(W) <=
    true(cell(1, 3, W)) &
    true(cell(2, 2, W)) &
    true(cell(3, 1, W))

open <= true(cell(M, N, b)
```

Note that, under the full information assumption, any of these relations can be assumed to be false if it is not provably true. Thus, we have complete definitions for the relations *legal*, *next*, *goal*, *terminal* in terms of *true* and *does*. The *true* relation starts out identical to *init* and on each step is changed to correspond to the extension of the *next* relation on that step. The upshot of this is that in every state of the game, each player can determine legality, termination, and goal values and—given the joint move of all players—can update the state.

Although GDL is designed for use in defining complete information games, it may also be extended to partial information games relatively easily. Unfortunately, the resulting descriptions are more verbose and more expensive to process. This extension to GDL is the subject of a future document.

Given the state machine model for games, we can define notions of playability and

winnability. While any description built in the language described above defines a game, for the purposes of analyzing intelligent behavior, we are interested in "good" games: games where players are able to move and have some chance of achieving their goals.

A game is *playable* if and only if every player has at least one legal move in every nonterminal state. In the GGP setting, we require that every game be playable.

A game is *strongly winnable* if and only if, for some player, there is a sequence of individual moves of that player that leads to a terminal state of the game where that player's goal value is maximal. A game is *weakly winnable* if and only if, for every player, there is a sequence of joint moves of all players that leads to a terminal state where that player's goal is maximal. In the GGP setting, every game must be weakly winnable, and all single-player games are strongly winnable. This means that in any general game, every player at least has a chance of winning.

General Game Players

Having a formal description of a game is one thing; being able to use that description to play the game effectively is something else entirely. In this section, we examine some of the problems of building general game players and discuss strategies for dealing with these difficulties.

Let us start with automated reasoning. Since game descriptions are written in logic, it is obviously necessary for a game player to do some degree of automated reasoning.

There are various choices here. (1) A game player can use the game description interpretively throughout a game. (2) It can map the description to a different representation and use that interpretively. (3) It can use the description to devise a specialized program to play the game. This is effectively automatic programming. There may be other options as well.

The good news is that there are powerful reasoners for first-order logic freely available. The bad news is that such reasoners do not, in and of themselves, solve the real problems of general game playing, which are the same whatever representation for the game rules is used, namely dealing with indeterminacy, size, and multigame commonalities.

The simplest sort of game is one in which there is just one player and the number of states and actions is relatively small. For such cases, traditional AI planning techniques are ideal. Depending on the shape of the search space, the player can search either forward or backward to find a sequence of actions/plays that convert the initial state into an acceptable goal state. Unfortunately, not all games are so simple.

To begin with, there is the indeterminacy that arises in games with multiple players. Recall that the succeeding state at each point in a game depends on the actions of all players, and remember that no player knows the actions of the other players in advance. Of course, in some cases, it is possible for a player to find sequences of actions guaranteed to achieve a goal state. However, this is quite rare.

More often, it is necessary to create conditional plans in which a player's future actions are determined by the player's earlier actions and those of the other players. For such cases, more complex planning techniques are necessary.

Unfortunately, even this is not always sufficient. In some cases, there may be no guaranteed plan at all, not even a conditional plan. Tic-tac-toe is a game of this sort. Although it can be won, there is no guaranteed way to win in general. It is not really clear what to do in such situations. The key to winning in such situations is to move and hope that the moves of the other players put the game into a state from which a guaranteed win is possible. However, this strategy leaves open the question of which moves to make prior to arrival at such a state. One can fall back on probabilistic reasoning. However, this is not wholly satisfactory since there is no justifiable way of selecting a probability distribution for the actions of the other players. Another approach, of primary use in directly competitive games, is to make moves that create more search for the other players so that there is a chance that time limitations will cause those players to err.

Another complexity, independent of indeterminacy, is sheer size. In tic-tac-toe, there are approximately 5,000 distinct states. This size is large but manageable. In chess there are approximately 10^{30} states. A state space of this size, being finite, is fully searchable in principle but not in practice. Moreover, the time limit on moves in most games means that players must select actions without knowing with certainty whether they are the best or even good moves to make.

In such cases, the usual approach is to conduct a partial search of some sort, examining the game tree to a certain depth, evaluating the possible outcomes at that point, and choosing actions accordingly. Of course, this approach relies on the availability of an evaluation function for nonterminal states that is roughly mo-

notonic in the actual probability of achieving a goal. While, for specific games, such as chess, programmers are able to build in evaluation functions in advance, this is not possible for general game playing, since the structure of the game is not known in advance. Rather, the game player must analyze the game itself in order to find a useful evaluation function.

Another approach to dealing with size is abstraction. In some cases, it is possible to reformulate a state graph into a more abstract state graph with the property that any solution to the abstract problem has a solution when refined to the full state graph. In such cases, it may be possible to find a guaranteed solution or a good evaluation function for the full graph. Various researchers have proposed techniques along these lines (Sacerdoti 1974; Knoblock 1991), but more work is needed.

The third issue is not so much a problem as an opportunity: multigame commonalities. After playing multiple instances of a single game or after playing multiple games against a given player, it may be possible to identify common lessons that can be transferred from one game instance to another. A player that is capable of learning such lessons and transferring them to other game instances is likely to do better than one without this capability.

One difficulty with this approach is that, in our current framework, players are not told the names of games, only the axioms. In order to transfer such lessons, a player must be able to recognize that it is the same game as before. If it is a slightly different game, the player must realize which lessons still apply and which are different.

Another difficulty, specific to this year's competition, is that players are not told the identity of the other players. So, lessons specific to players cannot be transferred, unless a player is able to recognize players by their style of play. (In future years, the restriction on supplying identity information about players may be removed, making such learning more useful.)

Game Management Infrastructure

In order to engage in competitive play, general game players need a central mediator to distribute game axioms, maintain an official game state, update it with player moves, verify the legality of those moves, and determine winners. Gamemaster is a generally available web service designed to assist the general game-playing community in developing and testing general game players by performing these functions. It is also intended to be used in managing tournaments such as this summer's GGP competition.

Gamemaster has three major components—the Arcade, the Game Editor, and the Game Manager. The Arcade is a database of information about games, players, and matches. The Game Editor assists individuals in creating and analyzing games. The Game Manager is responsible for running games or, precisely, matches, that is, instances of games. Of these components, the Game Manager is the most relevant to the theme of this paper. In the interest of brevity, we skip over the details of the Arcade and the Game Editor and devote the remainder of this section to the Game Manager. Figure 3 illustrates Game Manager operation.

To run a match with Gamemaster, an individual (hereafter called the game director) first creates a match by specifying (1) a game already known to the system, (2) the requisite number of players, (3) a *startclock* value (in seconds) and (4) a *playclock* value (in seconds).

Once this is done, the director can cause the match at any time by pressing the start button associated with the match he has created. The Game Manager then assumes all responsibiity for running the game (unless the director presses the stop button to abort the match). It communicates with players via messages using HTTP.

The process of running a game goes as follows: upon receiving a request to run a match, the Game Manager first sends a *Start* message to each player to initiate the match. Once game play begins, it sends *Play* messages to each player to get the plays and simulates the results. This part of the process repeats until the game is over. The Game Manager then sends *Stop* messages to each player. Figure 4 illustrates these exchanges for a game of tic-tac-toe, showing just the messages between the Game Manager and one of the players

The *Start* message lists the name of the match, the role the player is to assume (for example, white or black in chess), a formal description of the associated game (in GDL), and the startclock and playclock associated with the match. The startclock determines how much time remains before play begins. The playclock determines how much time each player has to make each move once play begins.

Upon receiving a *Start* message, each player sets up its data structures and does whatever analysis it deems desirable in the time available. It then replies to the Game Manager that it is ready for play. Having sent the *Start* message, the Game Manager waits for replies from the players. Once it has received these replies or once the startclock is exhausted, the Game Manager commences play.

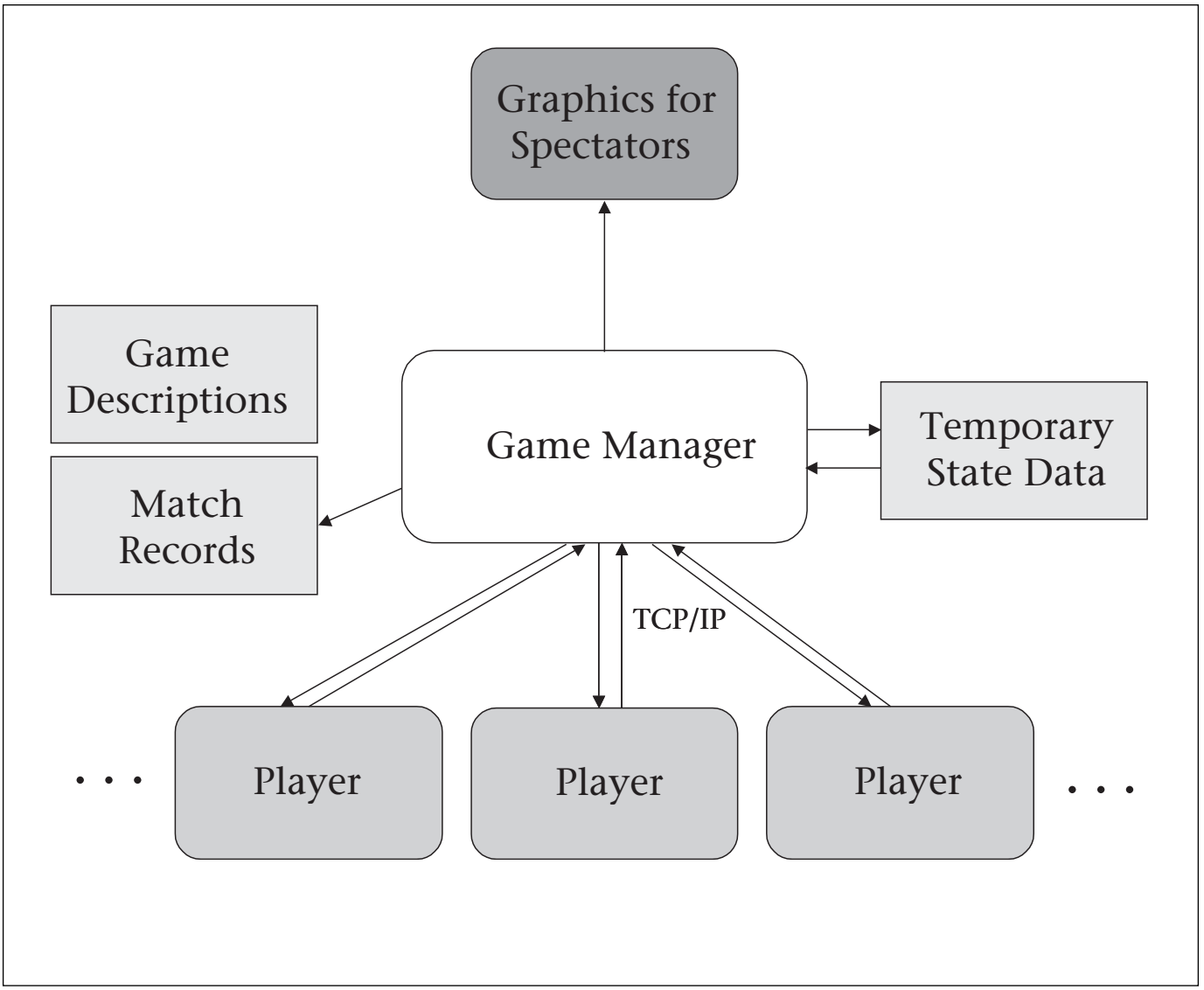


Figure 3. Game Manager.

On each step, the Game Manager sends a *Play* message to each player. The message includes information about the actions of all players on the preceding step. (On the first step, the argument is "nil"). On receiving a *Play* message, players spend their time trying to decide their moves. They must reply within the amount of time specified by the match's playclock.

The Game Manager waits for replies from the players. If a player does not respond before the playclock is exhausted, the Game Manager selects an arbitrary legal move. In any case, once all players reply or the playclock is exhausted, the Game Manager takes the specified moves or the legal moves it has determined for the players and determines the next game state. It then evaluates the termination condition to see if the game is over. If the game is not over, the Game Manager sends the moves of the players to all players and the process repeats.

Once a game is determined to be over, the Game Manager sends a *Stop* message to each player with information about the last moves made by all players. The *Stop* message allows players to clean up any data structures for the match. The information about previous plays is supplied so that players with learning components can profit from their experience. Having stopped all payers, the Game Manager then computes the rewards for each player, stores this information together with the play history in the Arcade database, and ceases operation.

Game Manager Message	Game Player Response
(START MATCH.435 WHITE *description* 90 30)	READY
(PLAY MATCH.435 (NIL NIL))	(MARK 2 2)
(PLAY MATCH.435 ((MARK 2 2) NOOP))	NOOP
(PLAY MATCH.435 (NOOP (MARK 1 3))	(MARK 1 2)
(PLAY MATCH.435 ((MARK 1 2) NOOP))	NOOP
. . .	. . .
(STOP MATCH.435 ((MARK 3 3) NOOP)	DONE

Figure 4. Game Communication.

Competition Details

The AAAI competition is designed to test the abilities of general game-playing systems by comparing their performance on a variety of games. The competition will consist of two phases: a qualification round and a runoff competition.

In the qualification round, entrants will play several different types of games, including single player games (such as maze search), competitive games (such as tic-tac-toe or some variant of chess), games with both competitors and cooperators. In some cases, the game will be exhaustively searchable (as in tic-tac-toe); in other cases, this will not be possible (as in chess). Players will have to handle these possibilities. For this year's competition, in all cases, complete information of the game will be available (as in chess or tic-tac-toe); in future competitions, only partial information will be available (as in battleship). Entrants will be evaluated on the basis of consistent legal play, ability to attain winning positions, and overall time; and the best will advance to the second round.

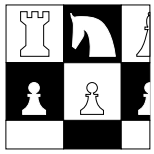
In the runoff round, the best of the qualifiers will be pitted against each other in a series of games of increasing complexity. The entrant to win the most games in this round will be the winner of the overall competition.

A $10, 000 prize will be awarded to the winning entrant. The competition is open to all computer systems, except those generated by affiliates of Stanford University. Clearly, no human players are allowed. The competition website[1] contains further details, including the description of the underlying framework, the game description language, and the programmatic interfaces necessary to play the games.

Conclusion

General game playing is a setting within which AI is the essential technology. It certainly concentrates attention on the notion of specification-based systems (declarative systems, self-aware systems, and, by extension, reconfigurable systems, self-organizing systems, and so forth). Building systems of this sort dates from the early years of AI.

In 1958, John McCarthy invented the concept of the "advice taker." The idea was simple: he wanted a machine that he could program by description. He would describe the intended environment and the desired goal, and the machine would use that information in determin-



ing its behavior. There would be no programming in the traditional sense. McCarthy presented his concept in a paper that has become a classic in the field of AI:

> The main advantage we expect the advice taker to have is that its behavior will be improvable merely by making statements to it, telling it about its environment and what is wanted from it. To make these statements will require little, if any, knowledge of the program or the previous knowledge of the advice taker (McCarthy 1959).

An ambitious goal! But that was a time of high hopes and grand ambitions. The idea caught the imaginations of numerous subsequent researchers—notably Bob Kowalski, the high priest of logic programming, and Ed Feigenbaum, the inventor of knowledge engineering. In a paper written in 1974, Feigenbaum gave his most forceful statement of McCarthy's ideal:

> The potential use of computers by people to accomplish tasks can be one-dimensionalized into a spectrum representing the nature of the instruction that must be given the computer to do its job. Call it the what-to-how spectrum. At one extreme of the spectrum, the user supplies his intelligence to instruct the machine with precision exactly how to do his job step-by-step.... At the other end of the spectrum is the user with his real problem.... He aspires to communicate what he wants done ... without having to lay out in detail all necessary subgoals for adequate performance (Feigenbaum 1974).

Some have argued that the way to achieve intelligent behavior is through specialization. That may work so long as the assumptions one makes in building such systems are true. General intelligence, however, requires general intellectual capabilities, and generally intelligent systems should be capable of performing well in a wide variety of tasks. In the words of Robert Heinlein (1973):

> A human being should be able to change a diaper, plan an invasion, butcher a hog, conn a ship, design a building, write a sonnet, balance accounts, build a wall, set a bone, comfort the dying, take orders, give orders, cooperate, act alone, solve equations, analyze a new problem, pitch manure, program a computer, cook a tasty meal, fight efficiently, die gallantly. Specialization is for insects.

We may wish to require the same of intelligent computer systems.

Note

1. http://games.stanford.edu.

References

Feigenbaum, E. A. 1974. Artificial Intelligence Research: What Is It? What Has It Achieved? Where Is It Going? Paper presented at the Symposium on Artificial Intelligence, Canberra, Australia.

Heinlein, R. 1973. *Time Enough for Love*. New York: Berkely Books, 1973.

Knoblock, C. A. 1991. Search Reduction in Hierarchical Problem Solving. In *Proceedings of the Ninth National Conference on Artificial Intelligence*. Menlo Park, CA: AAAI Press.

McCarthy, J. 1959. Programs with Common Sense. In *Proceedings of the Teddington Conference on the Mechanization of Thought Processes*, 75–91. London: Her Majesty's Stationary Office.

Sacerdoti, E. D. 1974. Planning in a Hierarchy of Abstraction Spaces. *Artificial Intelligence Journal* 5(2): 115–135.

Michael Genesereth is an associate professor in the Computer Science Department at Stanford University. He received his Sc.B. in physics from the Massachusetts Institute of Technology and his Ph.D. in applied mathematics from Harvard University. Genesereth is most known for his work on computational logic and applications of that work in enterprise computing and electronic commerce.

Nathaniel Love is a Ph.D. candidate in computer science at Stanford University. His research interests include computational logic, behavioral constraints, and their applications to both legal and game-playing domains. He earned a B.A. in mathematics and in computer science from Wesleyan University. He can be reached at natlove@stanford.edu.

Barney Pell, Ph.D. holds a B.S. degree from Stanford University, where he graduated Phi Beta Kappa, and a Ph.D. in computer science from Cambridge University, where he was a Marshall Scholar. Pell's doctoral research introduced the idea of a general game-playing competition, developed the first program to generate interesting chesslike games, and created METAGAMER, the first program to play existing and new chesslike games without any human assistance. Pell spent 7 of the previous 11 years at NASA, during which he was the architect of the Remote Agent, the first AI system to fly onboard and control a spacecraft, and later the manager of the 80-person "Collaborative Assistant Systems" research area. From 1998 to 2002, Pell served as vice president of Strategy and Business Development for two start-up companies, StockMaster.com and Whizbang! Labs. Pell is currently an entrepreneur in residence at Mayfield, a leading venture capital firm in the Silicon Valley.