LG-2006-01 April 4, 2006

General Game Playing: Game Description Language Specification

Nathaniel Love, Timothy Hinrichs, Michael Genesereth
{nlove,thinrich,genesereth}@cs.stanford.edu

Stanford Logic Group
Computer Science Department
Stanford University
353 Serra Mall
Stanford, CA 94305

1 Introduction

The specification in this document covers the class of games used for General Game Playing on Stanford's General Game Playing (GGP) test bed, and in the AAAI GGP Competitions: *finite, discrete, deterministic multi-player games of complete information.* This document begins by describing the mathematical models underlying general game playing and gives the specification for the game description language. The document follows with a description of the GGP game manager and the exchange of messages that constitute game play, along with a specification for the game communication language.

2 General Game Playing

The General Game Playing test bed (*Gamemaster*)[1] consists of an online Game Manager supporting multiple remote General Game Playing systems connecting over the web, playing *general games*: games with rules *unknown* to the players in advance, described in a *known* logical Game Description Language.

The Game Manager (GM) has many games available for play, each described by rules in a logical Game Description Language. When a game is initiated by remote GGP systems, the Game Manager sends each player the game description; shortly afterwards, the game begins, as the GM notifies the players that a turn has begun. The players submit their moves back to the GM, which tracks the state of the game (checking the goal conditions and verifying legal moves), and notifies players of the next turn, sending them information about their opponents' most recent moves so that the players can themselves update their own knowledge of the game state. When the game terminates, as dictated by the rules of the game, the GM records the winning players of the game.

3 Modeling General Games

The natural underlying model of environment, percepts and actions in a finite, discrete, deterministic multi-player games is a finite state machine. Given a finite environment with deterministic dynamics, a unique state can be created in a state machine for each distinct state of the environment, with outgoing arcs labeled with the joint moves of all players interacting with the system, and with percepts for each player interacting with the system. Since the number of players and their sets of possible percepts are all finite, the environment of the game world is representable as a finite state machine.

While the state machine formalism can represent multi-player game worlds and their dynamics, it presents problems when used for General Game Playing. Specifically, the size of a state machine representation is impractical for direct use in reasoning, and in many cases may be too large to store in memory or practically communicate to an agent at all. Both the monolithic representation of states and of player actions contribute to these problems, and complex rules for goals and legality increase this cost further. For example, a state machine representation of Chess requires over 10^{28} distinct states.

A further drawback of the state machine representation is the lack of modularity that accompanies the monolithic states and actions. The entire dynamics of the game world must be encoded in the transition function of the state machine, which makes no distinction between changes resulting from each player's

[1] http://games.stanford.edu:4000

actions (or even particular combinations of player moves) and those made by the inherent physics of the game.

In order to efficiently perform reasoning tasks necessary for General Game Playing, a compact, modular representation for the game world and its dynamics is needed. GDL (Game Description Language) has been developed to satisfy this need.

4 GDL

GDL describes the state of a game world in terms of a set of true facts. The transition function between states– the rules of the game –are described using logical rules in GDL that define the set of facts true in the next state in terms of the current state and the moves of all of the players. GDL contains constructs for distinguishing the initial state of the game, as well as goal states and terminal states. In this way, a game description in GDL defines a state machine satisfying the modeling requirements detailed in the previous section. Additionally, certain player moves are deemed *legal* depending on the state. Logical rules are also used to describe the legal move definitions, the termination conditions, and the goal conditions.

Given a game description in GDL, a GGP player can compute the set of facts true in the initial state of the game. Given that set of facts and the moves made by all players in the game, the GDL game description completely defines the set of facts true in the next state of the game. The game description, along with the set of facts currently true, also completely defines the current set of legal moves for each player, and defines whether the game state is terminal, and if so, what the goal values are for the players.

The following two sections define the syntax and semantics of GDL. Section 5 describes GDL as a variant of the Datalog language. Section 6 details the distinguished relations in GDL that are used to describe the components of the state machine model: *role, init, true, does, next, legal, goal,* and *terminal*. The use of these terms within a GDL game description is restricted in order to ensure that each game description corresponds to a state machine; section 6 defines these restrictions.

5 GDL and Datalog

The Game Description Language is a variant of Datalog that allows function constants, negation, and recursion. In GDL, a state of the game is defined by the set of propositions true in that state. The transition function between states– the rules of the game –are described using Datalog rules. These rules define the resulting state given the current state and the moves of all the players. Together with the initial state, the transition function defines a state machine satisfying the modeling requirements detailed in the previous section.

Additionally, certain player moves are deemed *legal* depending on the state. In certain states, the game terminates, and in those states each player is assigned a score. Datalog rules are also used to describe the legal move definitions, the termination conditions, and the goal conditions.

To be precise, GDL is $datalog^{\neg}$ with function constants, along with a set of distinguished relational predicates. The syntax and semantics of $datalog^{\neg}$ are described below, followed by the additional features GDL includes.

5.1 Syntax of $datalog^{\neg}$

The vocabulary, or signature, of a language is the set of building blocks from which sentences are formed.

Definition 1 (Vocabulary)**.** *A vocabulary consists of*

- *A set of relation constants with associated arity, e.g. adjacent, on.*
- *A set of object constants, e.g. a, b, c*

A *variable* is any symbol that starts with a capital letter.

Definition 2 (Term)**.** *A term is*

- *A variable*
- *An object constant*

Atomic sentences and literals are defined as is usual.

Definition 3 (Atomic Sentence)**.** *An atomic sentence is a relation constant of arity n applied to n terms, e.g.* $p(1, 1, X)$.

Definition 4 (Literal)**.** *A literal is an atomic sentence or the negation of an atomic sentence, e.g.* $p(1, 1, X)$ *or* $\neg p(1, 1, X)$.

Definition 5 (Ground Expression)**.** *An expression is ground if and only if it contains no variables.*

Datalog rules are implications where the head of the rule is always an atomic sentence. The body of the rule contains zero or more literals, with some special constraints on the negative literals.

Definition 6 (Datalog Rule)**.** *A datalog rule is an implication of the form*

$$h \Leftarrow b_1 \wedge \cdots \wedge b_n$$

- *The head, h, is an atomic sentence.*
- *Each literal in the body,* b_i*, is a literal.*
- *Safety: if a variable appears in the head or in a negative literal, it must appear in a positive literal in the body.*

The next section discusses why the last constraint exists; partly it exists to deal with negation. Datalog treats negation using negation as failure, which can cause problems; to avoid those problems, Datalog rules with negation must be *stratified*. A set of Datalog Rules is stratified if when starting from some relation constant p one can never reach $\neg p$ by following the rules backwards. For example, neither of the following two sets of rules are stratified.

- $p(X) \Leftarrow q(X) \wedge \neg p(X)$
- $p(X) \Leftarrow q(X)$
  $q(X) \Leftarrow r(X) \wedge \neg p(X))$

Stratification is defined in terms of a dependency graph.

Definition 7 (Dependency Graph)**.** *: Let Δ be a set of Datalog rules. The nodes of the dependency graph for Δ are the relation constants in the vocabulary. There is an edge from r_2 to r_1 whenever there is a rule with r_1 in the head and r_2 in the body. That edge is labeled with $\neg$ whenever r_2 is in a negative literal.*

Definition 8 (Stratified Datalog Rules)**.** *A set of Datalog rules is stratified if and only if there are no cycles that include an edge labeled with $\neg$ in the dependency graph for that rule set.*

In the second example above, there would be an edge from q to p without a label and an edge from r to q without a label and an edge from p to q with a $\neg$ label. Those two sentences are not stratified since the cycle p to q to p includes an edge labeled $\neg$.

5.2 Semantics of *datalog*$^{\neg}$

The meaning, or semantics, of a particular set of *datalog*$^{\neg}$ rules is based on the models that satisfy those rules.

Definition 9 (Model)**.** *A model for language L is a set of ground atomic sentences in L.*

Variables in Datalog range over all the ground terms in the language. Thus assuming the object constants in the vocabulary are a, b, and c, $p(X)$ means all the following are true. $p(a)$, $p(b)$, $p(c)$. Consequently, for the purpose of defining satisfaction, we can assume explicit universal quantifiers have been added so that every variable in a Datalog rule is captured, e.g. $p(X, Y)$ is treated as $\forall XY.p(X, Y)$.

Definition 10 (Satisfaction)**.** *Let M be a model, and let the sentence in question be an explicitly universally quantified Datalog rule.*

- $\models_M t_1 = t_2$ *if and only if* t_1 *and* t_2 *are the same term, syntactically.*
- $\models_M p(t_1, \ldots, t_n)$ *if and only if* $p(t_1, \ldots, t_n) \in M$
- $\models_M \neg\phi$ *if and only if* $\not\models_M \phi$
- $\models_M \phi_1 \wedge \cdots \wedge \phi_n$ *if and only if* $\models_M \phi_i$ *for every* i.
- $\models_M h \Leftarrow b_1 \wedge \cdots \wedge b_n$ *if and only if either* $\not\models_M b_1 \wedge \cdots \wedge b_n$ *or* $\models_M h$ *or both.*
- $\models_M \forall X.\phi(X)$ *if and only if* $\models_M \phi(t)$ *for every ground term* t.

It is well known that when a set of Datalog rules contains no negation, i.e. the set is Horn, there is a unique minimal model that satisfies that set. Model M is smaller than model N exactly when $M \subset N$. In Datalog without negation, that smallest model is the meaning of the set of rules.

Definition 11 (Entailment)**.** *Let Δ be a set of Datalog rules without negation. Let M be the smallest model that satisfies Δ.*

$$\Delta \models \phi \text{ if and only if } \models_M \phi$$

Adding negation complicates the definition of the semantics, but when a set of Datalog rules with negation is stratified, it turns out there is always a single model that can be uniquely identified–that model is the meaning of the set of stratified Datalog rules. It can be identified by constructively extending the minimal model of the negation-free portion of the rules.

Recall that the test for stratification involved constructing a dependency graph for the set of rules. This graph helps us construct the semantics of that set.

Definition 12 (Stratum)**.** *Let Δ be a set of Stratified Datalog rules, and consider the dependency graph for Δ. We say that the relation constant r is in stratum i if and only if the maximum number of $\neg$ arcs on any path starting at r is i. A rule whose head includes a relation constant in stratum i is said to be a rule in stratum i itself.*

It turns out that every rule in a set of Stratified Datalog rules belongs to a finite stratum, and there is at least one rule in stratum 0. All rules in stratum 0 have no negation; thus, they are Horn and have a unique minimal model M_0. The Stratified Datalog semantics starts with that model and extends it by adding all the consequences of rules in stratum 1 by evaluating the negative literals of each such rule in M_0. Then that model, M_1, is extended to M_2 by repeating the process with the rules in stratum 2, and so on.

Definition 13 (Stratified Datalog Semantics)**.** *Let Δ be a set of stratified Datalog rules. Let M_0 be the minimal model that satisfies all the rules in stratum 0. To compute M_i, where $i > 0$, initialize M_i to M_{i-1}. Perform the following operations, adding the result to M_i until no changes occur.*

- *Let $h \Leftarrow b_1 \wedge \cdots \wedge b_n$ be a rule in stratum i.*
- *For each substitution σ such that $\models_{M_i} (b_1 \wedge \cdots \wedge b_n)\sigma$, output $h\sigma$.*

Let k be the largest stratum for any rule in Δ. The Stratified Datalog semantics of Δ is M_k.

Definition 14 (Stratified Entailment)**.** *Let Δ be a set of Stratified Datalog rules. Let M be the Stratified Datalog semantics for Δ.*

$$\Delta \models \phi \text{ if and only if } \models_M \phi$$

The definition of *datalog*$^\neg$ rules includes a statement that says for an implication to be a *datalog*$^\neg$ rule, it must obey the following constraint: if a variable appears in the head of the rule or a variable appears in a negative literal, that variable must also appear in a positive literal in the body. Now that the semantics have been defined, the reason for this constraint can be explained.

Suppose we were to write a rule with a variable in the head that does not occur in the body or a rule with a variable in a negative literal that does not occur in a positive literal.

$$p(X, Y) \Leftarrow q(X)$$
$$s(a) \Leftarrow \neg r(Y)$$

In the first case, suppose $q(a)$ is true. Then $p(a, X)$ is also true. In *datalog*$^\neg$, the vocabulary is often assumed to include infinitely many object constants. With infinitely many object constants, there are infinitely many consequences of $q(a)$:

$$p(a, a), p(a, b), p(a, c), p(a, d), \ldots$$

That is, the model defined by the Datalog semantics is infinitely large. The constraint above removes this possibility by forcing every variable in the head of the rule to occur in a positive literal in the body. This condition is sufficient to ensure that a finite set of rules always has a finite model as its semantics.

In the second rule above, it is unclear how to interpret what the rule is supposed to mean. Does it mean that if there is some Y for which r is false that $s(a)$ is true? Or does it mean that if $r(t)$ for every ground term t is false that $s(a)$ is true? The constraint on Datalog rules removes this ambiguity by requiring every variable in a negative literal to also occur in a positive literal in the body; by first evaluating the positive literal, every negative literal can be ground before evaluating it with negation as failure.

It is noteworthy that as a consequence of these semantics, Datalog includes the Unique Names Assumption (UNA) and a Domain Closure Assumption (DCA). The UNA says every pair of terms that look different are unequal, and the DCA says that the only objects in the universe are those named by ground terms.

5.3 Differences from *datalog*$^\neg$

GDL builds directly on top of Datalog by making a small change having to do with equality, a syntactic feature that makes game descriptions slightly more compact, the inclusion of function constants, and adding a restriction on recursion that ensures finiteness of the semantics, which is required because of the presence of function constants.

Recall that in *datalog*$^\neg$, two ground terms are equal exactly when they are the same exact term, e.g. $a = a$ but $a \neq b$. Whereas *datalog*$^\neg$ includes equality directly allowing for both $=$ testing and $\neq$ testing, GDL only builds in a test for $\neq$ using the relation *distinct*. The rationale is that any test such as $X = Y$ can be built into a rule by replacing all occurrences of X with Y or vice versa. Alternatively, by including a rule such as $equal(Z, Z)$, the equality check $X = Y$ can be written as $equal(X, Y)$. Building `distinct` into the language is useful because encoding a relation for all unequal terms can be cumbersome.

Additionally, GDL allows a rule to include disjunctions of literals, mainly for convenience of writing game descriptions. This use of disjunctions can also speed up top-down processing of game descriptions. For example, consider two simple rules.

$$p(X) \Leftarrow q(X) \wedge r(X)$$
$$p(X) \Leftarrow q(X) \wedge s(X)$$

Each gives a different way to find an X such that $p(X)$ holds, but they differ in the second literal in the body. Suppose that computing $q(X)$ is expensive, e.g. in chess $q(X)$ might be true of all the pieces that have a player's king in check. Using typical Prolog-style top-down processing of these rules to find all the X in p, q must be computed in its entirety twice. But, if those two rules were to be written as the following single rule, we could avoid that expensive recomputation of q.

$$p(X) \Leftarrow q(X) \wedge (r(X) \vee s(X))$$

It should be noted, however, that this same expense could be avoided by introducing a new relation rs, with rules:

$$p(X) \Leftarrow q(X) \wedge rs(X)$$
$$rs(X) \Leftarrow r(X)$$
$$rs(X) \Leftarrow s(X)$$

In any case, the inclusion of disjunction allows for writing more compact game descriptions and removes the need for introducing additional relations.

As noted earlier, GDL is a language for describing finite state machines: it encodes an initial state and a transition function. Computing the transition function in a finite state machine is always decidable, and the output is always one of finitely many states; GDL should have similar properties.

As we shall see shortly, in GDL computing the transition function amounts to asking for all answers to a particular query about a game description. If GDL is to restrict games to a finite set of states and decidable transition functions, the result of this query must be finite and computing that result must always be decidable.

Unfortunately, in general, answering a Datalog query with recursion and function constants is formally undecidable. It is also possible that the number of answers to such a Datalog query is infinite. All is not lost. Several options exist for addressing these issues. Remove recursion and both problems disappear; remove function constants and both problems disappear. However, both these options make writing some games more cumbersome than necessary.

For that reason, we impose a syntactic restriction on the combination of function symbols and recursion in rules that is sufficient to ensure finiteness and decidability in all cases.

Definition 15 (Recursion Restriction)**.** *Let Δ be a set of rules, and let G be the dependency graph of Δ. Suppose that Δ contains the rule*

$$p(t_1, \ldots t_n) \Leftarrow b_1 \wedge \cdots \wedge q(v_1, \ldots, v_k) \wedge \cdots \wedge b_m$$

where q appears in a cycle with p in G. Then $\forall j \in \{1, \ldots, k\}$, either v_j is ground, $v_j \in \{t_1, \ldots, t_n\}$, or $\exists i_{\in\{1,\ldots,m\}}.b_i = r(\ldots, v_j, \ldots)$, where r does not appear in a cycle with p.

Essentially, this restriction ensures that functional terms cannot "grow" to unbounded size by use of recursive rules, since terms can only be passed along a recursive cycle without adding function symbols, and new terms introduced into recursive cycles must be grounded by a finite set of base relation sentences.

Below we summarize how the core of GDL differs from *datalog*$^\neg$ by altering the definitions to produce GDL, with changes denoted in bold (mathematical symbols surrounded by bold text are also new). The only remaining differences are the distinguished relation constants used to define a FSM with the variant of Datalog just described.

Definition 16 (Vocabulary)**.** *A vocabulary consists of*

- *A set of relation constants with associated arity, e.g. adjacent, on.*
- ***A set of function constants with associated arity, e.g.*** *f, g, h*
- *A set of object constants, e.g. a, b, c*

Definition 17 (Term)**.** *A term is*

- *A variable*
- *An object constant*
- ***A function constant of arity n applied to n terms, e.g.*** *$f(a, X, g(h(c, Y), e))$*

Definition 18 (Datalog Rule)**.** *A datalog rule is an implication of the form*

$$h \Leftarrow b_1 \wedge \cdots \wedge b_n$$

- *The head, h, is an atomic sentence.*
- *Each literal in the body, b_i, is a literal* ***or a disjunction of literals****.*
- ***After converting a rule with disjunction into a set of rules without disjunction, the following must hold of all resulting rules.*** *If a variable appears in the head or in a negative literal, it must appear in a positive literal in the body. distinct* ***is treated as a negative literal.***
- ***The Recursion Restriction given above must be satisfied.***

Definition 19 (Satisfaction)**.** *Let M be a model, and let the sentence in question be an explicitly universally quantified Datalog rule.*

- $\models_M distinct(t_1, t_2)$ ***if and only if*** t_1 ***and*** t_2 ***are not the same term, syntactically****.*
- $\models_M p(t_1, \ldots, t_n)$ *if and only if* $p(t_1, \ldots, t_n) \in M$
- $\models_M \neg\phi$ *if and only if* $\not\models_M \phi$
- $\models_M \phi_1 \wedge \cdots \wedge \phi_n$ *if and only if* $\models_M \phi_i$ *for every* i*.*
- $\models_M \phi_1 \vee \cdots \vee \phi_n$ ***if and only if*** $\models_M \phi_i$ ***for some*** i***.***
- $\models_M h \Leftarrow b_1 \wedge \cdots \wedge b_n$ *if and only if either* $\not\models_M b_1 \wedge \cdots \wedge b_n$ *or* $\models_M h$ *or both.*
- $\models_M \forall X.\phi(X)$ *if and only if* $\models_M \phi(t)$ *for every ground term* t*.*

The rest of this section gives a brief overview of Datalog written in KIF, the official language of GDL.

In KIF, variables begin with a ?. All operators are written in prefix notation. Every term and every sentence is a Lisp S-expression. The arity and type (relation, function, object) of constants are determined by context. For example,

- X is written `?x`
- $f(a, X, g(h(c, Y), e))$ is written `(f a ?x (g (h c ?y) e))`
- $true(cell(1, 1, X))$ is written `(true (cell 1 1 ?x))`
- $\neg true(cell(1, 1, X))$ is written `(not (true (cell 1 1 ?x)))`
- $p(X) \Leftarrow q(X) \wedge \neg r(X)$ is written `(<= (p ?x) (q ?x) (not (r ?x)))`
- $p(X) \Leftarrow q(X) \wedge (r(X) \vee s(X))$ is written
  `(<= (p ?x) (q ?x) (or (r ?x) (s ?x)))`

6 Defined Relations in GDL

The Game Description Language distinguishes the following set of relations: *role, init, true, does, next, legal, goal,* and *terminal*, in addition to *distinct*, the semantics of which are detailed in section 5.3. These relations provide the correspondence between a GDL game description and the mechanics of the game world.

Running Example: Tic-Tac-Toe

To illustrate the constructs of the game description language, we will use a simple familiar example, the game of Tic-Tac-Toe. The states of the Tic-Tac-Toe environment consist of a 3 x 3 grid where each cell in the grid is either blank or marked with an `x` or `o`. In the initial state, the cells are blank, and on each turn, the player with control makes their move by indicating which grid cell to mark.

6.1 Players: the *role* relation

The game description defines the players of the game through the *role* relation. In Tic Tac Toe, the complete *role* relation is:

```
(role xplayer)
(role oplayer)
```

telling us that the game has two players, referred to in the axioms as `xplayer` and `oplayer`.

6.2 Game State: the *true* relation

The current state of a game is represented by facts that take the form of ground functional terms in relational logic. The game description states which facts are true in the initial step of the game, and the axioms indicate how these facts change between game steps. In Tic-Tac-Toe, the ground term

```
(control xplayer)
```

is built from the function constant *control*, which is used in this turn-taking game to indicate whether it is a particular player's turn in the current state, and the object constant `xplayer`, which is a name of a player in the game. The axioms refer to certain such facts holding in the current state of the game through the *true* relation: *true(fact)*. In Tic-Tac-Toe, the atomic sentences

```
(true (control xplayer))
(true (cell 2 2 b))
```

indicate that in the current state, the center cell of the grid (at location 2,2) is blank (`b`), and the `xplayer` player has control.

6.3 Initial State: the *init* relation

The *init* predicate is an analogue to *true* for the initial state of the game, and is contained in the game description to provide a starting point for the game. In the initial state of a Tic-Tac-Toe game, all cells are blank, and the `x` player has control:

```
(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 xplayer))
```

6.4 Game State Update: the *next* relation

The *next* relation is an analogue to *true*, referring to facts that will hold in the next (as opposed to current) state of the game. For example, in Tic-Tac-Toe, control alternates between the two roles:

```
(<= (next (control xplayer))
    (true (control oplayer)))

(<= (next (control oplayer))
    (true (control xplayer)))
```

6.5 Legal Moves: the *legal* relation

The rules of a game restrict the moves played in the game by designating as legal certain moves in certain states. The *legal* relation: *legal(player,move)* holds for player-move combinations allowed by the game rules. In Tic-Tac-Toe, an alternating turn game, players can only mark cells when it is their turn in the current state (and only when those cells are blank), and must otherwise wait (perform an action called "noop").

```
(<= (legal ?w (mark ?x ?y))
    (true (cell ?x ?y b))
    (true (control ?w)))

(<= (legal xplayer noop)
    (true (control oplayer)))

(<= (legal oplayer noop)
    (true (control xplayer)))
```

6.6 Moves: the *does* relation

The *does* relation: *does(player, move)* indicates the moves actually made by players in a particular step of the game. The axioms reference the *does* relation in rules that govern state update. In Tic-Tac-Toe, the rule:

```
(<= (next (cell ?m ?n x))
    (does xplayer (mark ?m ?n))
    (true (cell ?m ?n b)))
```

states that when the `xplayer` player performs the *mark* move on a previously blank grid location, the grid location will be marked with an `x` in the next step. In addition, frame axioms define what remains unchanged given a particular player action. In Tic-Tac-Toe, we have:

```
(<= (next (cell ?m ?n b))
    (does ?w (mark ?j ?k))
    (true (cell ?m ?n b))
    (or (distinct ?m ?j) (distinct ?n ?k)))
```

which indicates that a blank cell remains blank if a player marks a different cell.

6.7 Prior Moves

In General Game Playing with complete information games, before making a move, each player in a game needs to be notified of the moves made by the other players on the previous step of the game. With this information, each player can compute the current state of the game (the complete *true* relation). This information takes the form of ground atomic *does* statements. In Tic-Tac-Toe, if the `xplayer` player marks the center of the grid, while the `oplayer` player performs a noop, the prior move information

```
(does xplayer (mark 2 2))
(does oplayer noop)
```

when added to the previous state of the game will enable computation of the new game state. In the GGP setting, the Game Manager transmits this information to the players after each step of the game.

6.8 Goal States: the *goal* relation

The game description includes rules that describe goals for the game's players in terms of the game state. The goals of two players may be inverses (as in zero-sum games), asymmetric, or even complementary (as in cooperative games). The rules defining goals are used by the GM to determine the value of the final state of the game to the various players. Goals may be binary valued– either a goal is satisfied in a step (score = 100) or it is not (score = 0) –or the goal relation may define different scores for players given different conditions on propositions, allowing integer goal values between 0 and 100. In Tic-Tac-Toe, we find the following goal rule (among others), stating that the role `xplayer` achieves a maximal goal value for making a line of `x`s:

```
(<= (goal xplayer 100)
    (line x))
```

where the *line* relation is defined elsewhere in the game's axioms to hold for a player who has a marked row, column, or diagonal of cells on the board.

6.9 Terminal States: the *terminal* relation

The game description includes a definition of the set of terminal states in terms of facts true in those states. The nullary *terminal* relation: *terminal* is used in the axioms to define these termination conditions for a game state. In Tic-Tac-Toe, the game terminates when a player has made a line of marks, or after the board is full:

```
(<= terminal
    (line x))

(<= terminal
    (line o))

(<= terminal
    (not open))
```

6.10 Restrictions on GDL Relations

In order to ensure that GDL game descriptions correspond to game worlds (and state machines) in such a way that the reserved relations match their intended semantics (given above), we need to place restrictions on how these relations appear in Datalog rules in a game description. For example, if a game description contained the rule

```
(<= (role p) (true q))
```

this would seem to say that "if `q` is true in the current state, then `p` is a player in the game." This is a rule we would not want to include in a game description– the names and number of players in a game is fixed, and is not a condition of the current state. We add the following additional syntactic restrictions on Datalog rules in GDL.

Definition 20 (GDL Restrictions)**.** *Let Δ be a GDL game description, and let G be the dependancy graph for Δ. Then each of the following conditions must hold of Δ:*

- *The* `role` *relation only appears in ground atomic sentences (as the head of a Datalog rule with no body).*
- *The* `init` *relation only appears in the heads of Datalog rules (not in the bodies), and in G, the* init *node is not in the same connected component as* true, does, next, legal, goal, *or* terminal.
- *The* `true` *relation only appears in the bodies of Datalog rules (not in the heads).*
- *The* `next` *relation only appears in the heads of Datalog rules.*
- *The* `does` *relation only appears in the bodies of Datalog rules, and in G, there are no paths between the* does *node and any of* legal, goal, *or* terminal.

6.11 Producing Well-Formed Games

The preceding definitions constrain GDL to produce game descriptions for which players can compute their legal moves and compute the next state given the moves of all players. However, there are additional constraints required to limit the scope of GDL to well-formed games.

Definition 21 (Termination)**.** *A game description in GDL* terminates *if all infinite sequences of legal moves from the initial state of the game reach a terminal state after a finite number of steps.*

Definition 22 (Playability)**.** *A game description in GDL is* playable *if and only if every role has at least one legal move in every non-terminal state reachable from the initial state.*

Definition 23 (Winnability)**.** *A game description in GDL is* strongly winnable *if and only if, for some role, there is a sequence of individual moves of that role that leads to a terminal state of the game where that role's goal value is maximal. A game description in GDL is* weakly winnable *if and only if, for every role, there is a sequence of joint moves of all roles that leads to a terminal state where that role's goal value is maximal.*

Definition 24 (Well-formed Games)**.** *A game description in GDL is* well-formed *if it terminates and is both playable and weakly winnable.*

This definition requires that all well-formed single player games be strongly winnable. Clearly, it is possible to generate game descriptions in GDL which are not well-formed. Checking game descriptions to see if they are well-formed can certainly be done in general by using brute-force methods (expanding the entire game graph), and for some games, faster algorithms may exists. Game descriptions used in GGP competitions should always be well-formed.

7 Game Manager

General game player systems compete in games by communicating with a game manager (GM). The GM distributes the description of the game to the players, then solicits moves from the players, updating the state of the game at each step by performing the sequence of state machine operations outlined above. The entire sequence of game steps can be completely computed by the GM, given the players' moves, and upon reaching a terminal state, the GM stops the game and records the goal values achieved by the players. This process (a complete run of a game, with GGP players submitting moves) is called a match.

7.1 Roles

The binding of game roles to particular GGP players in a match is performed by the GM by passing an additional argument to along with the game description. Each GGP player is assigned a single role in a match.

7.2 Illegal Moves

The *legal* relation allows the GM to determine whether the moves submitted by players are legal with respect to the rules of the game. In a multi-player game, if a player submits an illegal move (or fails to submit a

move at all), the GM will substitute a random legal move for that player on that turn. In subsequent turns, the GM will again request moves from the player, and the player should be able to resume submitting their own legal moves.

7.3 Clocks

To ensure that game play progresses in a fair and expeditious manner, the Game Manager maintains a play clock to track the move responses of each of the players. For any given game, an integer is defined as the *playclock* for all players, for each move of the game. That is, if the playclock is set to 20 for a particular game, game players have 20 seconds after receiving a request to reply to the game manager with their move for that step. A failure to reply with a legal move before the clock expires will carry the same results as making an illegal move, described above. Players must maintain their own internal clocks, since they are only informed once of the move clock setting for the game, and are not warned or given any clock information during game play.

In addition, when the game commences, and the game description is first sent to the players, an initial *startclock* (separate from the playclock) gives players time to deliberate before the first turn begins. When *startclock* seconds have passed since sending the game descriptions, the GM will request the first move from all players. Both of these clock settings are determined by the game administrator when initiating a match.

7.4 Game Manager and Gamemaster

The Game Manager plays the central role in the entire General Game Playing web framework (Gamemaster). Gamemaster holds registration information for remotely connecting game players, stores game descriptions, maintains game and player histories, and facilitates the TCP/IP communication between the GM and the game players.

Figures 1 through 7 illustrate Gamemaster architecture, including the communication model for game play and the sequence of messages exchanged between the GM and the game players during game execution.

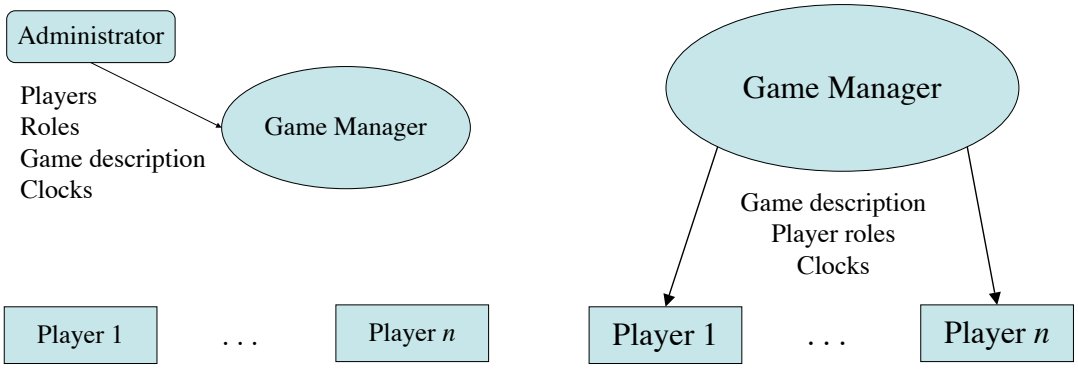

Figure 1: Match Administration and Initialization

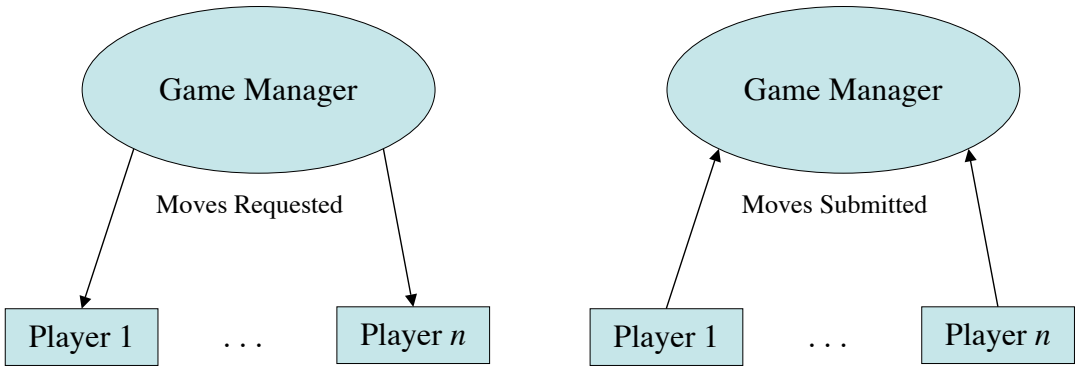

Figure 2: Requesting and Submitting Moves

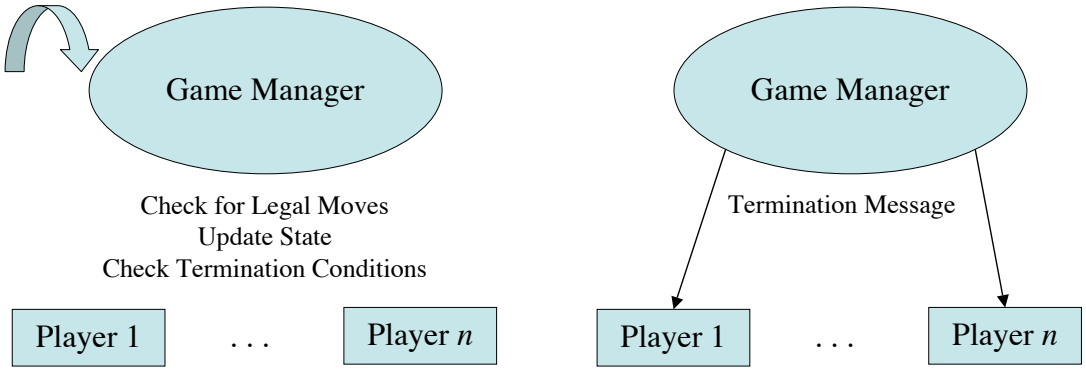

Figure 3: Update and Termination

8 Communication Language

Communication between game players and the Game Manager takes place through HTTP connections. The communication model assumes a player running on a host listening on a particular port. HTTP messages sent to players have the standard HTTP header, with content type text/acl. In the case of games played through the GGP website, the sender specified in the header will always be "Gamemaster," and the receiver will be the remote player name as registered on the Gamemaster system.

The body of the HTTP message consists one of several commands with sets of KIF sentences as arguments.

8.1 START Command

The `START` command is used to initialize game play. The command takes three arguments, providing the player with their role in the game, the game description, and the value of the move clock:

```
(START <MATCHID> <ROLE> <DESCRIPTION> <STARTCLOCK> <PLAYCLOCK>)
```

`<MATCHID>` is a unique identifier of the match being played, since a particular player (communicating with the GM from the same host and port) could play more than one match at a time. All subsequent messages exchanged for this match will contain the same ID.
`<ROLE>` is a single element of the *role* relation of the game description, and this is the role assigned to this player for this match.
`<DESCRIPTION>` is a set of KIF sentences, enclosed by an outer set of parentheses.
`<STARTCLOCK>` and `<STARTCLOCK>` are integers. A message with this command is sent once by the GM to each player to initialize the game.

8.2 PLAY Command

The `PLAY` command is used to initiate each step of the game. The command takes two arguments: the first the match ID, the second is is a list of the actions taken by all players on the previous step:

```
(PLAY <MATCHID> (<A1> <A2> ... <An>))
```

Each `<An>` is an action taken by one of the players of the game in the previous step. The ordering of actions corresponds to the ordering of roles given in the game description axioms. For example, if the game description contained `role(white)` `role(black)` in that order, then `(EAST SOUTH)` would inform the players that the white player moved east on the previous step, while the black player moved south (this game, unlike Tic-Tac-Toe, has simultaneous moves).

8.3 STOP Command

The `STOP` command is used to notify the end of the game. The command is similar to the `PLAY` command in structure:

```
(STOP <MATCHID> (<A1> <A2> . . . <An>))
```

This command, like the `PLAY` command, contains the actions from the previous step, so that players can completely compute the next step of the game, in which the `terminal` condition has been met. A message with this command is sent once by the GM to each player to terminate the game.

8.4 Player Replies

After receiving a `START` message, each player should send an HTTP reply with the body

```
READY
```

before *startclock* seconds have elapsed. However, when the startclock has expired, the Game Manager will send `PLAY` messages to all players regardless of whether they have replied. If all players have responded with

```
READY
```

before this time is up, the Game Manager will begin the game immediately, sending the first `PLAY` messages.

The `PLAY` command expects an HTTP reply with a body containing the player's selected move:

```
<ACTION>
```

For example, in a game of Tic-Tac-Toe, a player might reply with

```
(MARK 2 2)
```

In response to a `STOP` message, a game player should reply[2] with

```
DONE
```

Appendix B contains some additional detail on the format of the HTTP messages and an extended example.

A Full Tic-Tac-Toe Game Description in prefix KIF

Note that lines beginning with `;` (semicolon) are comments.

```
;;;;;;;;;;;;;;;;;;;;;;;;
;; Tictactoe
;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;
;; Roles
;;;;;;;;;;;;;;;;;;;;;;;;
  (role xplayer)
  (role oplayer)

;;;;;;;;;;;;;;;;;;;;;;;;
;; Initial State
;;;;;;;;;;;;;;;;;;;;;;;;
  (init (cell 1 1 b))
  (init (cell 1 2 b))
  (init (cell 1 3 b))
  (init (cell 2 1 b))
```

[2] Just to be polite.

```
  (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 xplayer))

;;;;;;;;;;;;;;;;;;;;;;;;
;; Dynamic Components
;;;;;;;;;;;;;;;;;;;;;;;;
;; Cell

  (<= (next (cell ?m ?n x))
      (does xplayer (mark ?m ?n))
      (true (cell ?m ?n b)))

  (<= (next (cell ?m ?n o))
      (does oplayer (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 b))
      (or (distinct ?m ?j) (distinct ?n ?k)))

;; Control

  (<= (next (control xplayer))
      (true (control oplayer)))

  (<= (next (control oplayer))
      (true (control xplayer)))

;;;;;;;;;;;;;;;;;;;;;;;;
;; Views
;;;;;;;;;;;;;;;;;;;;;;;;

  (<= (row ?m ?x)
```

```
      (true (cell ?m 1 ?x))
      (true (cell ?m 2 ?x))
      (true (cell ?m 3 ?x)))

  (<= (column ?n ?x)
      (true (cell 1 ?n ?x))
      (true (cell 2 ?n ?x))
      (true (cell 3 ?n ?x)))

  (<= (diagonal ?x)
      (true (cell 1 1 ?x))
      (true (cell 2 2 ?x))
      (true (cell 3 3 ?x)))

  (<= (diagonal ?x)
      (true (cell 1 3 ?x))
      (true (cell 2 2 ?x))
      (true (cell 3 1 ?x)))


  (<= (line ?x) (row ?m ?x))
  (<= (line ?x) (column ?m ?x))
  (<= (line ?x) (diagonal ?x))


  (<= open
      (true (cell ?m ?n b)))

;;;;;;;;;;;;;;;;;;;;;;;;
;; Legal Moves
;;;;;;;;;;;;;;;;;;;;;;;;

  (<= (legal ?w (mark ?x ?y))
      (true (cell ?x ?y b))
      (true (control ?w)))

  (<= (legal xplayer noop)
      (true (control oplayer)))

  (<= (legal oplayer noop)
      (true (control xplayer)))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;
;; Goals
;;;;;;;;;;;;;;;;;;;;;;;;

  (<= (goal xplayer 100)
      (line x))

  (<= (goal xplayer 50)
      (not (line x))
      (not (line o))
      (not open))

  (<= (goal xplayer 0)
      (line o))

  (<= (goal oplayer 100)
      (line o))

  (<= (goal oplayer 50)
      (not (line x))
      (not (line o))
      (not open))

  (<= (goal oplayer 0)
      (line x))

;;;;;;;;;;;;;;;;;;;;;;;;
;; Terminal
;;;;;;;;;;;;;;;;;;;;;;;;

  (<= terminal
      (line x))

  (<= terminal
      (line o))

  (<= terminal
      (not open))
```

B Communication Protocol

Sample Start Message

```
POST / HTTP/1.0
Accept: text/delim
Sender: GAMEMASTER
Receiver: GAMEPLAYER
Content-type: text/acl
Content-length: 1194

(START MATCH.3316980891 ROBOT
((ROLE ROBOT) (INIT (CELL A)) (INIT (GOLD C)) (INIT (STEP 1))
(<= (NEXT (CELL ?Y)) (DOES ROBOT MOVE) (TRUE (CELL ?X))
(ADJACENT ?X ?Y)) (<= (NEXT (CELL ?X)) (DOES ROBOT GRAB)
(TRUE (CELL ?X))) (<= (NEXT (CELL ?X)) (DOES ROBOT DROP)
(TRUE (CELL ?X))) (<= (NEXT (GOLD ?X)) (DOES ROBOT MOVE)
(TRUE (GOLD ?X))) (<= (NEXT (GOLD I)) (DOES ROBOT GRAB)
(TRUE (CELL ?X)) (TRUE (GOLD ?X))) (<= (NEXT (GOLD I))
(DOES ROBOT GRAB) (TRUE (GOLD I)))
(<= (NEXT (GOLD ?Y)) (DOES ROBOT GRAB) (TRUE (CELL ?X))
(TRUE (GOLD ?Y)) (DISTINCT ?X ?Y)) (<= (NEXT (GOLD ?X))
(DOES ROBOT DROP) (TRUE (CELL ?X)) (TRUE (GOLD I)))
(<= (NEXT (GOLD ?X)) (DOES ROBOT DROP) (TRUE (GOLD ?X))
(DISTINCT ?X I)) (<= (NEXT (STEP ?Y)) (TRUE (STEP ?X))
(SUCC ?X ?Y)) (ADJACENT A B) (ADJACENT B C) (ADJACENT C D)
(ADJACENT D A) (SUCC 1 2) (SUCC 2 3) (SUCC 3 4) (SUCC 4 5)
(SUCC 5 6) (SUCC 6 7) (SUCC 7 8) (SUCC 8 9) (SUCC 9 10)
(<= (LEGAL ROBOT MOVE)) (<= (LEGAL ROBOT GRAB)
(TRUE (CELL ?X)) (TRUE (GOLD ?X))) (<= (LEGAL ROBOT DROP)
(TRUE (GOLD I))) (<= (GOAL ROBOT 100) (TRUE (GOLD A)))
(<= (GOAL ROBOT 0) (TRUE (GOLD ?X)) (DISTINCT ?X A))
(<= TERMINAL (TRUE (STEP 10))) (<= TERMINAL (TRUE (GOLD A))))
 30 30)
```

Reply

```
HTTP/1.0 200 OK
Content-type: text/acl
Content-length: 5

READY
```

Play Message

```
POST / HTTP/1.0
Accept: text/delim
Sender: GAMEMASTER
Receiver: GAMEPLAYER
Content-type: text/acl
Content-length: 27

(PLAY MATCH.3316980891 NIL)
```

Reply

```
HTTP/1.0 200 OK
Content-type: text/acl
Content-length: 4

MOVE
```

Since this is the Maze game, "MOVE" is the name of one of the legal actions (the others being GRAB and DROP), and this has no other arguments. In a game of Tic-Tac-Toe, this message might look like:

```
HTTP/1.0 200 OK
Content-type: text/acl
Content-length: 10

(MARK 2 2)
```

Next, after receiving this move, the GM sends the message

```
POST / HTTP/1.0
Accept: text/delim
Sender: GAMEMASTER
Receiver: GAMEPLAYER
Content-type: text/acl
Content-length: 30

(PLAY MATCH.3316980891 (MOVE))
```

where (MOVE) appears since it is the set of actions taken by all players on the previous step. In this one-player game, this information only reflects what the single player has done, which may seem extraneous. However, in the case that the player had submitted an illegal move, and the GM made an aribtrary legal move on its behalf, the player would need to know what arbitrary move had been made. In a game like Tic-Tac-Toe, we would see a message like the following, where the list ((MARK 2 2) NOOP) indicates that on the previous time step, the first player marked the center square and the second player performed a noop.

```
POST / HTTP/1.0
Accept: text/delim
Sender: GAMEMASTER
Receiver: GAMEPLAYER
Content-type: text/acl
Content-length: 41

(PLAY MATCH.3316980892 ((MARK 2 2) NOOP))
```

The sequence of messages continues:

```
HTTP/1.0 200 OK
Content-type: text/acl
Content-length: 4

MOVE

POST / HTTP/1.0
Accept: text/delim
Sender: GAMEMASTER
Receiver: GAMEPLAYER
Content-type: text/acl
Content-length: 30

(PLAY MATCH.3316980891 (MOVE))
HTTP/1.0 200 OK
Content-type: text/acl
Content-length: 4

GRAB

POST / HTTP/1.0
Accept: text/delim
Sender: GAMEMASTER
Receiver: GAMEPLAYER
Content-type: text/acl
Content-length: 30

(PLAY MATCH.3316980891 (GRAB))

HTTP/1.0 200 OK
Content-type: text/acl
Content-length: 4
```

```
MOVE

POST / HTTP/1.0
Accept: text/delim
Sender: GAMEMASTER
Receiver: GAMEPLAYER
Content-type: text/acl
Content-length: 30

(PLAY MATCH.3316980891 (MOVE))

HTTP/1.0 200 OK
Content-type: text/acl
Content-length: 4

MOVE

POST / HTTP/1.0
Accept: text/delim
Sender: GAMEMASTER
Receiver: GAMEPLAYER
Content-type: text/acl
Content-length: 30

(PLAY MATCH.3316980891 (MOVE))

HTTP/1.0 200 OK
Content-type: text/acl
Content-length: 4

DROP
```

At this point, the player has won the game, so the GM sends the message

```
POST / HTTP/1.0
Accept: text/delim
Sender: GAMEMASTER
Receiver: GAMEPLAYER
Content-type: text/acl
Content-length: 30

(STOP MATCH.3316980891 (DROP))
```

To which the Gameplayer replies

```
HTTP/1.0 200 OK
Content-type: text/acl
Content-length: 4

DONE
```