Technical Note November 20, 2006

Automatically Proving Playability through Abstraction

Timothy Hinrichs

Stanford Logic Group
Computer Science Department
Stanford University
353 Serra Mall
Stanford, CA 94305
thinrich@cs.stanford.edu

1 Introduction

A General Game Playing (GGP) system is one that can accept a formal description of an unknown game in a known language at runtime and, without further human interaction, can play the game effectively.

General Game Playing systems are characterized by their use of general cognitive information processing technologies (such as knowledge representation, reasoning, learning, and rational behavior). Unlike specialized game playing systems (such as Deep Blue), they do not rely exclusively on algorithms designed in advance for specific games. Deep Blue may have beaten the world chess champion, but it cannot play checkers or even balance a check book.

While general games have undeniable inherent interest, work in this area has value that goes well beyond game playing. The underlying technology can also be used in a variety of other, arguably more important application areas, such as enterprise management, electronic commerce, security, and military operations.

This document addresses a problem that authors of general games must face: checking whether the game they have written meets all the requirements of being a GGP game. The first requirement is that the game must be written in GDL (Game Description Language); that check is a simple syntactic one. The second requirement is that all GGP games must satisfy the following properties.

Termination A game terminates if and only if there is some maximum number of steps for the game. Most traditional games are of this form. For instance, there is a rule in chess that states the game is over if no pieces have been captured for 40 moves.

Winnability A game is winnable for a particular player if and only if there is some path in the game tree where that player wins. A game is winnable iff it is winnable for every player.

Playability A game state is playable if and only if it is either terminal or there are legal moves for each player. A game is playable if and only if every reachable state in the game is playable.

A game author usually has a good idea that a game satisfies all the above properties, but on several occasions we have found bugs in games we thought were safe. Our energies have been spent mostly on proving playability, as most of the bugs we have found in our games have failed to satisfy this property, and it is the most damaging property for a game not to satisfy.

If at some point in the game, some player has no legal move, it is unclear what should happen; consequently, the general game playing testbed will produce unpredicable results in this case. The entire system built with the assumption that every player will have some legal move in every state of every game.

Our approach to proving playability is to construct an abstract version of the game in question such that its playability implies the playability of the original game. Checking the playability of this abstract game has fewer states than the original game; thus, checking its playability is less computationally demanding.

2 Game Description Language

Because the central concern of this document is to convey methods for proving playability in the context of general game playing, here we review the game description language (GDL).

GDL is a language for compactly describing finite state machines whose transitions are triggered by actions of external agents. Recall that a state machine is defined formally as a 5-tuple $<S, \Sigma, \delta, s_0, F>$ where

- S is the finite set of states
- Σ is the finite set of possible inputs (or actions in the GGP setting)
- δ is the transition function that defines what state results when an input is received in a state. $\delta : S \times \Sigma \rightarrow S$
- s_0 is the initial state. $s_0 \in S$
- F is the set of terminal states. $F \subseteq S$

A finite state machine is a system that transitions from state to state based on external inputs. The system halts whenever the machine transitions into one of the terminal states.

In GDL a state has structure; it is represented as the set of ground atoms true in the state. The transition function is represented as a finite set of logical sentences. To compute the next state given the current state and players' actions, we add to the game description all the ground atoms $true(\beta)$ where β is a ground atom true in the state and $does(\rho, \alpha)$ for each player ρ who performed action α, and then compute all the x such that $next(x)$ is entailed. A game is over whenever *terminal* is true in the current state.

For example, in tic-tac-toe, we might write the following rule to define a part of the transition function.

$$next(cell(M, N, x)) \Leftarrow does(xplayer, mark(M, N)) \wedge true(cell(M, N, b))$$

Here we use the convention that variables start with a capital letter. Cells on the tic-tac-toe board are identified by their x and y coordinates in the cartesian plane. This rule says that if *xplayer* marks cell $\langle M, N\rangle$ when cell $\langle M, N\rangle$ was blank, then in the resulting state, cell $\langle M, N\rangle$ has an x placed in it.

Either we can interpret *next*, *true*, and *does* as modals so that *cell* and *mark* can be relations, or if we want to avoid modals, *next*, *true*, and *does* can be interpreted as distinguished relations, which makes *cell*, and *mark* function constants.

In GDL, there is another distinguished, binary relation *legal* that is true of a player and all the actions it is allowed to perform in the current state.

For example, in tic-tac-toe, the first rule below says that it is legal for *xplayer* to mark a blank square if it is *xplayer*'s turn. The second rule says it is legal to noop if it is not *xplayer*'s turn.

$$legal(xplayer, mark(M, N)) \Leftarrow true(control(xplayer)) \wedge true(cell(M, N, b))$$
$$legal(xplayer, noop) \Leftarrow \neg true(control(xplayer))$$

The reserved symbols in GDL are *true*, *next*, *does*, *legal*, *terminal*, *goal*, *init*, and *role*. $goal(\rho, ?x)$ defines how many points the current state is worth to player ρ. *init* defines the initial state. *role* defines the names of the players.

It is important to note that GDL is a subset of stratified Datalog with negation; thus, it employs minimal model semantics – the only things true are those that must be true. GDL allows a restricted form of recursion and function symbols, so that entailment is decidable and the number of states in the game finite. More information can be found in the GDL specification [LHG06].

Games in GDL are fully-observable, finite, discrete, multi-agent, synchronous environments. Every agent can observe everything about the game except what moves its opponents are going to make. The number of distinct states is always finite; multiple-agents work cooperatively, competitively, or in some combination to achieve their goals; and the environment changes only in response to agent actions, i.e. it does not change while the agents are deliberating over what move to make.

3 Checking Playability

Because GDL restricts the number of states to be finite and the logical description of the game to be decidable, playability is trivially decidable by walking the game tree. Of course, for games like chess where the number of states is around 10^{30}, brute force is impractical.

So far, we have experimented with three types of algorithms for proving the playability of a game.

Brute Force : Check whether each state in the game tree is playable. It is sound, complete, and always terminates but is expensive.

Random Probing : Randomly explore the game tree, checking whether each visited state is playable. If a state is found to be unplayable, it is returned and the algorithm halts. This algorithm will give us a certain confidence interval on the playability of the game. Always terminates.

Abstraction : Compute an abstract characterization of the game tree, which sometimes includes states not in the game tree, and show that each such abstract state is playable. Sound (for certain classes of GDL games) but incomplete . Performance and termination are highly dependent on the game axiomatization.

Brute force and random probing are fairly straightforward. Treating the game tree as a graph instead of a tree can greatly improve the performance of brute force. Abstraction seems to either terminate immediately or not terminate at all; nevertheless, it is an attractive option because its cost is sometimes independent of the size of game. For example, the proof of playability by abstraction for the current tic-tac-toe axioms is the same regardless whether the board is 3x3 or 100x100.

3.1 Playability via Abstraction

To prove the playability of a game induced by a game G, we need to show that for every state s, either s is unreachable, or it is terminal, or every player has a legal move.

$$G \models \neg reachable(s) \vee terminal(s) \vee legals(s)$$

where $legals(s)$ means every player has a legal move in state s.[1] When we do this via abstraction, we show the equivalent statement for the abstract states, one by one:

$$G \models reachable(s) \wedge \neg terminal(s) \Rightarrow legals(s)$$
$$G \wedge reachable(s) \wedge \neg terminal(s) \models legals(s)$$

That is, we compute an abstract characterization of the reachable states, and for each one, add in the negation of terminal to the game description, and try to prove there are legal moves for each player in that state. If we succeed, the game is playable. If for some abstract state, we fail to prove there is a legal move for some player, either the game is not playable or the abstract state in question includes unreachable states

[1]While we have not formally defined what it means for a game G to entail anything, the intuition is hopefully clear, which is all we are trying to convey.

that do not satisfy $legals(s)$. If our test for playability succeeds, we know the game is playable; if the test fails however, we cannot conclude the game is unplayable.

Our implementation of this idea has three parameters that control how hard the machine tries to prove the game playable. It is available at the website http://games.stanford.edu:4000/analyzer/analyzer?.

1. Whether to include the negation of terminal
2. Whether to approximate the set of reachable states
3. Whether to compute mutexes

By turning all the options off, the machine attempts to prove

$$G \models \forall s.legals(s)$$

By turning on the option that includes the negation of terminal, the machine tries to prove

$$G \models \forall s.(\neg terminal(s) \Rightarrow legals(s))$$

By turning on the options for the negation of terminal and approximation of reachable states, the machine computes a set of abstract states and tries to prove the following for each abstract state s

$$G \wedge \neg terminal(s) \wedge true(s) \models legals(s)$$

where $true(s)$ corresponds to setting the current state to s. In this case, we move the universal quantifier outside the logic.

If the mutex option is turned on, the machine tries to find properties of states that are never true at the same time. For example, in tic-tac-toe, there are properties for keeping track of whose turn it is: $control(white)$ and $control(black)$. It turns out that in the algorithms we currently use to compute abstract states, many of the states include both $control(white)$ and $control(black)$. Computing mutexes allows us to prune those unreachable abstract states. This final option causes the machine to try to eliminate from the set of reachable states any state that includes two mutex properties, thus reducing the number of states we check for playability even further.

$$G \wedge \neg terminal(s) \wedge true(s) \wedge \neg mutexes(s) \models legals(s)$$

The next three sections detail how this algorithm computes the negation of terminal, the set of abstract states, and the mutex properties. They are combined as described above, according to the options set by the user. In contrast to this section where our goal is to convey the algorithm that manipulates a set of GDL axioms at the metalevel, in this and subsequent sections, our goal is to demonstrate the algorithms used to reason with GDL directly.

3.2 The Negation of *terminal*

Recall from the earlier discussion that in GDL, computing whether a state s is terminal means adding the ground atoms true in s to the game description and computing whether *terminal* is entailed. In this section, we demonstrate how one computes the negation of terminal, i.e. an expression for all the states in which *terminal* is false.

Figure 1: Computing $\neg terminal$

```
INPUT: $\Delta$, a GDL description
OUTPUT: $\phi$, an expression whose only relation constant is $true$
1. $e :=$ proofTreeCompletion[$terminal, \Delta$]
2. return CNF[$\neg e$]
```

To compute an expression that represents all the nonterminal states, we first perform proof-tree completion, a form of abduction, [McI98] to rewrite *terminal* in terms of *true*. This process simply flattens the definition of *terminal*, a standard technique that can be applied when the definition of *terminal* is not recursive. Then after existentially quantifying all the variables, we negate the result, and convert to clausal form. (This last step is noteworthy only because it can be expensive.)

For example, suppose we have the following definition for *terminal*.

$$\begin{aligned} terminal &\Leftarrow row \\ row &\Leftarrow true(p) \\ row &\Leftarrow true(q) \end{aligned}$$

After rewriting *terminal* in terms of *true*, we get

$$true(p) \vee true(q)$$

whose negation is

$$\neg true(p) \wedge \neg true(q)$$

This expression represents the conditions under which the state is not terminal. That is, if *terminal* is false, we know that both $true(p)$ and $true(q)$ are both false.

For this computation to work, it assumes parallel predicate completion applied to the GDL game description produces a first-order theory equivalent to the GDL theory. Unfortunately, recursive rules often break this assumption because GDL has stratified datalog semantics (a form of minimal model semantics) while first-order logic does not.

Consider the following example.

$$\begin{aligned} &p(x,y) \Leftarrow p(x,z) \wedge p(z,y) \\ &p(a,b) \\ &p(b,c) \end{aligned}$$

The only ground literals entailed in first-order logic are $p(a,b)$, $p(b,c)$, and $p(a,c)$, but in GDL $\neg p(a,a)$, $\neg p(b,a)$, $\neg p(b,b)$, $\neg p(c,a)$, $\neg p(c,b)$, $\neg p(c,c)$ are entailed as well. First-order logic admits an axiomatization of this theory because the set of objects is finite, but parallel predicate completion does not result in such an axiomatization. Predicate completion produces the following sentence.

$$\begin{aligned} p(x,y) \Leftrightarrow \quad & (x=a \wedge y=b) \vee \\ & (x=b \wedge y=c) \vee \\ & \exists z.(p(x,z) \wedge p(z,y)) \end{aligned}$$

This sentence is satisfied by at least two different models in first-order logic: the GDL consequences from the last paragraph, and the one where everything is true of p. Thus, the original GDL axioms define the theory

$$p(a, b), p(b, c), p(a, c), \neg p(a, a), \neg p(b, a), \neg p(b, b), \neg p(c, a), \neg p(c, b), \neg p(c, c)$$

but parallel predicate completion produces only a subset of that theory:

$$p(a, b), p(b, c), p(a, c)$$

Whenever parallel predicate completion can be applied to the GDL definition for *terminal* to produce a first-order theory that is logically equivalent to the GDL definition of *terminal*, the procedure outlined in this section can be applied to compute an expression that characterizes all the nonterminal states as soundly and completely as proof-tree completion rewrites *terminal* in terms of *true* in that first-order axiomatization.

4 Approximating the Set of Reachable States

The set of reachable states for a game is defined recursively. The initial state is reachable; and every state that is the result of each player choosing a legal action in a reachable state is reachable. The set of reachable states may or may not have a compact description. This section discusses one algorithm for over-approximating the set of reachable states through abstraction.

An abstraction of some set of reachable states is useful for the purposes of playability whenever that abstraction (1) represents a significant number of reachable states and (2) represents no unreachable state that is not playable, i.e. every state represented admits at least one legal move for each player. Our approach represents abstract states as existentially quantified sets of literals, .

For example, the state of tic-tac-toe where there is some mark in the center square can be represented as the following sentence.

$$\exists x.true(cell(2, 2, x))$$

The state where there is some mark somewhere can be represented by

$$\exists xyz.true(cell(x, y, z))$$

Equivalently, we can use skolem constants to take the place of existential quantifiers.

$$true(cell(2, 2, k))$$

$$true(cell(k_1, k_2, k_3))$$

Because the set of all reachable states is defined recursively, it is natural to prove something about all the reachable states inductively. In our approach, the base case is the initial state; it must first be shown that every player has a legal move in the initial state. Then, the inductive step assumes that states n transitions from the initial state are playable, i.e. that there is a legal move for each player in those states and attempts to show the same is true for all states n+1 transitions from the initial state.

For the inductive step, the inductive hypothesis is operationalized as follows. First, the machine computes an abstract characterization of states for which there is a legal move for each player. Then for each of those abstract states, the machine computes the state resulting from the players executing the (abstract) action that is guaranteed to exist. The result is a set of abstract states for which we must show every player has a legal move.

To compute an abstract characterization of the states with legal moves, we use proof-tree completion again starting with $\exists x.legal(r, x)$ for each role r. That is, we rewrite $\exists x.legal(r, x)$ in terms of *true* and *does* just like we rewrote *terminal* in terms of *true* in the last section. These facts are then converted into disjunctive normal form (after skolemizing). We treat each disjunct as the set of statements that must be true in any state for which r has a legal move. To get an abstract state in which *every* role has a legal move, we compute the cross product of these abstract statements. In the end we get a set of abstract states (with skolem constants) with abstract actions for every player.

For example, in tic-tac-toe, we compute the following abstract states and abstract actions.

1. $true(cell(k_1, k_2, b)) \wedge true(control(xplayer)) \wedge does(xplayer, mark(k_1, k_2)) \wedge$ $true(cell(k_3, k_4, b)) \wedge true(control(oplayer) \wedge does(oplayer, mark(k_3, k_4))$
2. $true(cell(k_1, k_2, b)) \wedge true(control(xplayer)) \wedge does(xplayer, mark(k_1, k_2)) \wedge does(oplayer, noop)$
3. $true(cell(k_1, k_2, b)) \wedge true(control(oplayer)) \wedge does(oplayer, mark(k_1, k_2)) \wedge does(xplayer, noop)$
4. $true(control(oplayer)) \wedge true(control(xplayer)) \wedge does(oplayer, noop) \wedge does(xplayer, noop)$

The first item in the list above is one where both *xplayer* and *oplayer* have control, there is at least one blank square, and both players mark a square. If tic-tac-toe were written so that both players could mark at the same time, this state would cover the case where that happens. But since in tic-tac-toe that can never happen, this abstract state represents a set states all of which are unreachable. The next section discusses how to compute that the properties $control(xplayer)$ and $control(oplayer)$ are mutually exclusive; with that information, the first state above can be recognized as unreachable and discarded.

The second item above is the typical one where *xplayer* marks a square and *oplayer* does nothing. In the third item, the opposite happens: *oplayer* marks and *xplayer* does nothing. In the fourth item neither player does anything, which again requires both *xplayer* and *oplayer* to be in control. It is also discarded once mutexes are computed.

Each of the items above represents an abstract state and the abstract actions taken in that state. These can be used to compute the next (abstract) state. But notice that unlike a non-abstract state, these abstract states are only partially defined. $true(cell(k_1, k_2, b))$ only says that there is at least one blank cell. It does not say there is only one; it does not say there are no cells marked with an x. Thus these abstract states are actually partial state descriptions. Consequently, we cannot in general assume that the set of true facts in an abstract state are the only true facts. And because of this, instead of using algorithms for processing GDL, which assume the given set of true facts are the only true facts, we must use algorithms that do not make that assumption. Our approach uses standard first-order deduction techniques.

The difficulty in using first-order techniques is that not all GDL axiom sets can be represented in first-order logic, as discussed in the last section. Because GDL which employs a form of single-model semantics, whereas first-order logic employs standard model-theoretic semantics. For example, in Datalog, we can compute $ancestor(x, y)$, the transitive closure of $parent(x, y)$, whereas in first-order logic, transitive closure and therefore *ancestor* is not definable. If, however, the GDL axioms are non-recursive, we can always compute a first-order representation of those axioms by employing predicate completion.

Figure 2: Approximating Reachable States

```
ALG: inductiveStep
INPUT: $\Delta$, a set of GDL axioms with roles $R$
OUPUT: Whether the inductive step is completed successfully or not.
$S[r]$ :=rewrite$[\exists x.legal(r,x), \Delta, \{true, does\}]$
$S := S[r_1] \times \cdots \times S[r_n]$
$C$ :=predicateCompletion$[\Delta]$
for each $s \in S$
    $n := \{next(x) | C \cup s \models_{FO} next(x)\}$
    for each $r \in R$
        if $n \cup C \not\models \exists x.legal(r_1,x) \wedge \cdots \wedge \exists x.legal(r_n,x)$
            return FAIL
return SUCCESS
```

The first-order query we use to compute the state resulting from executing abstract actions A in abstract state S in the GDL game Δ is find all the x such that

$$A \wedge S \wedge predicateCompletion[\Delta] \models next(x)$$

Thus, for each A and S, we can compute an abstract state the represents the result of executing action A in state S. We then use first-order techniques (again because the resulting abstract state is a partial state) to ask whether the rules of the game ensure there is a legal move for each player in the resulting state. If every player has a legal move in that state, we continue on to the next A,S. If some role does not have a legal move, we stop and report that we have failed to prove the game is playable, and point to the A,S pair. If on the other hand, every A,S pair produces a state in which every player has a legal action, then we conclude the inductive step for the playability proof has been completed.

There is a bit of difficulty worth discussing at this point. Consider a variant of tic-tac-toe where the players move simultaneously: at every step of the game, both players choose where they want to mark. If they choose different squares, their marks are placed in the cells they chose. If they choose the same square, no mark gets placed. Interestingly, if the game does not end before the board is full, there will be only a single blank square. Since both players will only have one choice for where to put their mark, they will choose the same square, which results in the square not being marked, and the game continues forever.

To correctly prove that this game is playable, we must consider two qualitatively different cases of the abstract state/action pair for item one in the list above: (1) where the cells the two players mark are different, i.e. $\langle k_1, k_2 \rangle \neq \langle k_3, k_4 \rangle$ and (2) where the cells the two players mark are the same, i.e. $k_1 = k_3$ and $k_2 = k_4$. This corresponds to equality reasoning where some subset of the skolem constants k_1, k_2, k_3, k_4 are all equal. To address this, our approach has been to preprocess all the possible equivalences, producing from each of these abstract state/action pairs a set of abstract state/action pairs where two skolems that look unequal are unequal. Thus, we would produce the following (incomplete) list of abstract state/actions from the first abstract state/action pair above.

- $true(cell(k_1, k_2, b)) \wedge true(control(xplayer)) \wedge does(xplayer, mark(k_1, k_2)) \wedge$

Figure 3: Abstract Playability Algorithm

```
ALG: abstractPlayability
INPUT: $\Delta$, a set of GDL axioms
OUPUT: Success if $\Delta$ has been proven playable.
if the initial state in $\Delta$ is not terminal and there is some player without a legal move, return FAIL.
return inductiveStep[$\Delta$]
```

$true(cell(k_3, k_4, b)) \wedge true(control(oplayer) \wedge does(oplayer, mark(k_3, k_4))$

- $true(cell(k_1, k_2, b)) \wedge true(control(xplayer)) \wedge does(xplayer, mark(k_1, k_2)) \wedge$ $true(control(oplayer) \wedge does(oplayer, mark(k_1, k_2))$
- $true(cell(k_1, k_2, b)) \wedge true(control(xplayer)) \wedge does(xplayer, mark(k_1, k_2)) \wedge$ $true(cell(k_1, k_4, b)) \wedge true(control(oplayer) \wedge does(oplayer, mark(k_1, k_4))$
- $true(cell(k_1, k_2, b)) \wedge true(control(xplayer)) \wedge does(xplayer, mark(k_1, k_2)) \wedge$ $true(cell(k_3, k_2, b)) \wedge true(control(oplayer) \wedge does(oplayer, mark(k_3, k_2))$

The first one has the players marking in different cells. The second has them marking in the same cell. The third has them marking in the same column, and the fourth has them marking in the same row. In some games, rows and columns might be important, so unless we prove that the only important cases are when the cells are the same or when the cells are not the same, we need to check all possible skolem equivalences.

Assuming the inductive step succeeds without finding an abstract state/action pair that produces an unplayable state, and the initial state was playable, the machine can conclude that the game is playable. The inductive algorithm is given in Fig. 3.

5 Computing Mutually Exclusive Properties

Two properties of a state are mutex if and only if there is no reachable state where they are both true or both false. As mentioned earlier, in tic-tac-toe, there are two properties that keep track of whose turn it is: $control(xplayer)$ and $control(oplayer)$. To automatically identify mutex properties in a game, we perform two steps: we first generate candidate pairs of properties and then we use induction to show that the properties are actually mutex, once again using standard first-order reasoning since we only have partial state descriptions.

Computing the candidate mutex pairs is similar to computing the abstract states that have legal moves. We rewrite $\exists x.next(x)$ in terms of *true*. That is, we compute the conditions under which something is true in the next state. By writing the result in disjunctive normal form, we get a series of conjunctions of *true* literals, each of which implies $\exists x.next(x)$.

In tic-tac-toe, we compute the following disjunction.

$$\begin{gathered} next(control(xplayer)) \Leftarrow true(control(oplayer)) \\ \vee \\ next(control(oplayer)) \Leftarrow true(control(xplayer)) \end{gathered}$$

Thus, we conjecture that $control(xplayer)$ and $control(oplayer)$ are mutex.

To determine whether a candidate pair is actually mutex, we attempt a proof by induction. Showing they are mutex in the initial state is easy enough. The inductive step assumes the two properties are mutex, i.e. one of them is true and the other is false, and then demonstrates that regardless how the players act, in the resulting state one of the properties is true and one of the properties is false.

In tic-tac-toe, the inductive step has two steps. First, assume $true(control(xplayer)) \wedge \neg true(control(oplayer))$; then compute all the possible resulting (abstract) states, using first-order techniques. Check that in all of them $true(control(xplayer))$ and $true(control(oplayer))$ are neither both true nor both false. Then, perform the second step, where we start by assuming $true(control(oplayer)) \wedge \neg true(control(xplayer))$. If both steps succeed, the inductive step and the inductive proof have both been completed.

6 Conclusion

A game is playable if and only if in every reachable, nonterminal state, every player has at least one legal move. The major difficulty in proving playability is in dealing with the fact that some states are reachable while others are not. With enough computational power, every GDL game can be checked for playability by brute force – by simply walking over all the reachable, nonterminal states and checking whether each player has a legal move in each of them. Because this approach is impractical for all but a small fraction of games, we must explore other techniques.

Our approach is to prove playability by induction, and in so doing construct an abstract version of the game such that if it is playable, the original game must also be playable. One of the difficulties that arises because of this approach is that we need to reason about incomplete knowledge of states of the game. This is problematic because GDL assumes knowledge of the game state is always complete. In order to perform reasoning about incomplete states, we therefore attempt to translate the GDL game description into first-order logic, which is not always possible in the presence of recursion.

A web implementation of the algorithm described in this document has been in existence for over a year; it was installed just before the first annual AAAI general game playing competition in the summer of 2005. This document is an extension of the explanatory note found on that site: http://games:4000/analyzer/analyzer?.

References

[LHG06] Nathaniel Love, Timothy Hinrichs, and Michael Genesereth. Game description language specification. *Stanford Logic Group Technical Report LG-2006-01*, 2006.

[McI98] Sheila McIlraith. Logic-based abductive inference. *Stanford Knowledge Systems Lab Technical Report KSL98-19*, 1998.