Solving Two-player Games with QBF Solvers in General Game Playing

Yifan He
UNSW Sydney
Australia
harryheyifan99@gmail.com

Abdallah Saffidine
UNSW Sydney
Australia
abdallah.saffidine@gmail.com

Michael Thielscher
UNSW Sydney
Australia
mit@unsw.edu.au

ABSTRACT

Game solving is a relatively less explored area in general game playing. This paper introduces a translation from the Game Description Language GDL to Quantified Boolean Formulas (QBF) that lets us leverage QBF solvers to compute winning strategies in two-player games described in GDL. We implement this approach and measure the computation time needed by state-of-the-art QBF solvers on a range of two-player zero-sum turn-taking games. We introduce a variety of optimizations to the translation and evaluate them experimentally. Our empirical analysis establishes that our proposed approach is suitable for solving small games and can potentially help general game players evaluate endgame positions.

KEYWORDS

General Game Playing; Quantified Answer Set Programming

ACM Reference Format:
Yifan He, Abdallah Saffidine, and Michael Thielscher. 2024. Solving Two-player Games with QBF Solvers in General Game Playing. In *Proc. of the 23rd International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2024), Auckland, New Zealand, May 6 – 10, 2024*, IFAAMAS, 9 pages.

1 INTRODUCTION

General game players are systems that can play strategic games based on game descriptions supplied at runtime. These systems are generally unaware of the game rules until the competition starts. GDL, the general Game Description Language, is a widely used language for describing the rules of games [9]. Various approaches to general game playing have been developed in the past, most notably Minimax with automatically generated heuristics [28], MCTS [4], SCSP [17], and Deep Reinforcement Learning [13]. However, all of these approaches target game-*playing* whereas relatively few attempts were made in the direction of game-*solving* in general game playing. While it is challenging to design a solver capable of solving all the games that can be described in GDL, it is interesting to investigate whether there are classes of games described in GDL that can be played perfectly. Since GDL is a logic language, the use of logical formula solvers to solve GDL games is a natural idea. One important work on game-solving used answer set programming (ASP) [8] to solve single-player games with perfect information [32]. This was achieved by converting single-player games described in GDL to ASP and applying the ASP-solver Clingo to compute solutions to these games.

Two-player, zero-sum turn-taking games form a broader class of games that has raised a lot of research interest over the years. Examples of this type of game include Breakthrough [14], Generalized Tic-Tac-Toe [6], Connect-4 [11], and Dots and Boxes [2]. One particularly interesting approach to solving this type of game is to encode the game by a QBF [5] and call a QBF solver [19, 24] to evaluate the resulting expression. It has been shown in Generalized Tic-Tac-Toe that QBF solvers can solve the game faster than proof number search solvers [6]. However, most existing QBF encodings target specific games such as Connect-4 [11] or a specific class of games such as positional games [22]. One piece of work similar to ours is the concise translation from two-player zero-sum turn-taking board games that are described in Board-game Domain Definition Language (BDDL) to QBF [30]. Although that translation covers both positional games and some non-positional games, it can only convert a subclass of board games to QBF and is therefore far from covering all two-player zero-sum turn-taking games with perfect information.

In this paper, we present a translation from GDL to QBF that applies to all two-player zero-sum turn-taking GDL games. Instead of converting from GDL to QBF directly, we translate from GDL to Quantified Answer Set Programming (QASP), and use existing methods [7] to convert the resulting QASP to QBF. Furthermore, we develop a variety of optimizations to the translation. We provide an empirical analysis that shows that our proposed approach is suitable for solving small games and can potentially help general game players evaluate endgame position.

The rest of the paper is structured as follows. In Section 2, we review some preliminaries on GDL and QASP. We then describe our GDL to QASP translation method (Section 3). In Section 4, we evaluate the efficiency of our translation on different families of benchmarks. We conclude in Section 5.

2 PRELIMINARIES

2.1 Game Description Language (GDL)

The Game Description Language (GDL) can be used to describe the rules of any finite perfect information game [32]. GDL describes game rules in normal logic program syntax similar to Prolog [34]. There are preserved keywords to describe the different elements of a game [9, 10]; these keywords are listed in Table 1. There are some further restrictions for a set of GDL rules to be a valid game description. Firstly, `role` can appear only in facts; `init` and `next` can only appear as heads of rules; and `true` and `does` only appear in rule bodies. Secondly, `init` is not connected to `true`, `does`, `legal`, `next`, `terminal`, or `goal` in the predicate dependency graph [9];

Proc. of the 23rd International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2024), N. Alechina, V. Dignum, M. Dastani, J.S. Sichman (eds.), May 6 – 10, 2024, Auckland, New Zealand. © 2024 International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org).

role(R)	R is a player
init(F)	F holds in the initial position
true(F)	F holds in current position
next(F)	F holds in the next position
legal(R,M)	M is a valid action for player R
does(R,M)	Player R performs action M
terminal	The current position is terminal
goal(R,V)	Player R gets point V in current position
distinct(X,Y)	X and Y are syntactically different
input(R,A)	Action A is in the move domain of player R
base(P)	P is a base proposition

Table 1: GDL keywords

moreover, there is no path from does to any of `legal`, `terminal`, and `goal` in the predicate dependency graph. Finally, a GDL game description must be stratified, allowed, and well-formed [9]. A stratified logic program is known to admit a unique stable model [32].

GDL rules can be interpreted as a multiagent state transition system of the given game. Let G be a valid game description and Σ the set of ground terms in G. Suppose that $S = \{f_1, f_2, ..., f_n\} \subseteq \Sigma$ is any given position and $A = \{r_1, ..., r_k\} \to \Sigma$ any function that assigns a move to each of the k players. Position S can be encoded as a set of facts using keyword *true*: $S^{true} = \{true(f_1)., true(f_2), ..., true(f_n).\}$, and the joint actions by the k players are encoded as a set of facts using keyword *does*: $A^{does} = \{does(r_1, A(r_1))., ..., does(r_k, A(r_k)).\}$. The state transition system is then obtained as follows.

Definition 1 ([29]). *Let G be a GDL specification whose signature determines the set of ground terms Σ. Let 2^Σ be the power set of Σ. The semantics of G is the state transition system (R, S_1, T, l, u, g) where:*

- *$R = \{r \in \Sigma \mid G \models$ `role(r)`$\}$ (the players);*
- *$S_1 = \{f \in \Sigma \mid G \models$ `init(f)`$\}$ (the initial position);*
- *$T = \{S \in 2^\Sigma \mid G \cup S^{true} \models$ `terminal`$\}$ (the terminal positions);*
- *$l = \{(r, a, S) \mid G \cup S^{true} \models$ `legal(r,a)`$\}$, where $r \in R$, $a \in \Sigma$, and $S \in 2^\Sigma$ (the legality relation);*
- *$u(A, S) = \{f \in \Sigma \mid G \cup S^{true} \cup A^{does} \models$ `next(f)`$\}$, for all $A : (R \to \Sigma)$ and $S \in 2^\Sigma$ (the update function);*
- *$g = \{(r, v, S) \mid G \cup S^{true} \models$ `goal(r,v)`$\}$, where $r \in R$, $v \in \mathbb{N}$, and $S \in 2^\Sigma$ (the goal relation).*

Based on the above definition, we represent a *valid game playing sequence* of n steps as follows.

$$S_1 \xrightarrow{A_1} S_2 \xrightarrow{A_2} ... S_n \xrightarrow{A_n} S_{n+1}$$

In the above game sequence, we write $S_i \xrightarrow{A_i} S_{i+1}$ if $S_i \notin T$ and all moves are legal in the state in which they are taken, that is, $(r, A(r), S_i) \in l$ for each $r \in R$. We say a valid game playing sequence *terminates in n steps* if $S_{n+1} \in T$ [33].

Because of the similarities of syntax and semantics, Answer Set Programming [8] is a natural choice for reasoning about the properties of a GDL game. To this end, GDL rules are often extended by a "time" dimension. The *Temporal Extension* of a GDL game is defined as follows.

Definition 2 ([32]). *Let G be a valid GDL description. The* Temporal Extension of G *is an Answer Set Program denoted by Ext(G) which is obtained from G as follows.*

(1) *Change all* `distinct(X,Y)` *to* X `!=` Y.
(2) *Change all* `true(`ψ`)` *to* `true(`ψ`,T)`, *all* `next(`ψ`)` *to* `true(`ψ`,T+1)`, *and all* `init(`ψ`)` *to* `true(`ψ`,1)`.
(3) *For all atoms of form* $\mathtt{p}(t_1, t_2, ...t_n)$ *such that* $\mathtt{p} \notin \{$`true, init, next, role, distinct`$\}$ *and there exists a path from* does *or* `true` *to* p *in the predicate dependency graph, change it to* $\mathtt{p}(t_1, t_2, ...t_n, T)$. *In particular, if* $\mathtt{p} \in \{$`goal, terminal, legal`$\}$, p *is always extended with the time dimension.*
(4) *For each rule that originally has* `next` *in the head, add* `mtdom(T)` *to the body. For each rule whose head is not* `next` *but has been extended by a "time" dimension, add the literal* `tdom(T)` *to the body.*

We also extend the definition of A^{does} to

$$A^{does}(i) = \{does(r_1, A(r_k), i), ..., does(r_k, A(r_k), i)\}$$

In the above definition, the predicate `tdom` represents the time domain of the game states while `mtdom` represents the timestamps in which the players are making moves. For the rest of the paper, we assume the domain of `tdom` is $\{1...T_{max}+1\}$ while the domain of `mtdom` is $\{1...T_{max}\}$, where T_{max} is the intended depth of the game. The reason for the range of `mtdom` to be one less than `tdom` is that, for GDL, the action taken at timestamp t can only "affect" predicates defining termination and outcome of a game (i.e., `terminal` and `goal`) at time $t+1$ but not at time t itself.

Note that the program $Ext(G)$ is stratified whenever G is. The following theorem shows the semantics equivalence of $Ext(G)$ and valid game playing sequences in the original game description G.

Theorem 1 ([33]). *Consider a GDL description G with semantics (R, S_1, T, l, u, g) and a sequence*

$$S_1 \xrightarrow{A_1} S_2 \xrightarrow{A_2} ...S_n \xrightarrow{A_n} S_{n+1}.$$

Let $P = Ext(G) \cup A_1^{does}(1) \cup \ldots \cup A_n^{does}(n)$. Then, for any predicate symbol p in the game description G and for all $1 \le i \le n+1$, such that p is not `init` *or* `next` *and p does not depend on* does *in the predicate dependency graph, we have 1) $S_i = \{f \mid P \models true(f, i)\}$, and 2) $G \cup S_i^{true} \models p(t)$ iff $P \models p(t, i)$. In particular, since* `legal` *does not depend on* does, *$G \cup S_i^{true} \models legal(r, a)$ iff $P \models legal(r, a, i)$.*

2.2 Quantified Answer Set Programming

Similar to the difference between QBF [5] and SAT [21], allowing quantifiers in ASP programs can provide a more expressive language. The resulting language is called Quantified Answer Set Programming (QASP). Its semantics is defined as follows.

Definition 3 ([7]). *Suppose P is a logic program with ground atoms $\mathcal{A}$. A* quantified logic program *over $\mathcal{A}$ has the form*

$$Q_1\ X_1\ ...\ Q_n\ X_n\ P$$

where X_i are pairwise disjoint subsets of $\mathcal{A}$, every Q_i is either $\exists$ or $\forall$, and P is a logic program over $\mathcal{A}$. For convenience, let us define $fix(X, Y)$, where $Y \subseteq X \subseteq \mathcal{A}$, as the logic program

$$\{:\text{-}\ not\ x. \mid x \in Y\} \cup \{:\text{-}\ x. \mid x \in X \setminus Y\}.$$

A normal logic program P is satisfiable iff it has a stable model. Satisfiability of a QASP is recursively defined as follows.

(1) *If the QASP has form $\exists\ X\ P$ (resp. $\forall\ X\ P$), the program is satisfiable iff there exists (resp. for all) $Y \subseteq X$ such that the program $P \cup fix(X, Y)$ is satisfiable.*

(2) *If the QASP has form* $\exists\ X\ \mathbf{Q}\ P$ *(resp.* $\forall\ X\ \mathbf{Q}\ P$*), the program is satisfiable iff there exists (resp. for all)* $Y \subseteq X$ *such that the program* $\mathbf{Q}\ (P \cup fix(X, Y))$ *is satisfiable.*

QASP has the same expressive power as QBF, which allows us to model many two-player games. Because of the existing work on converting logic programs to propositional formulas such as *lp2sat* [15], a QASP can be solved by converting it to an equal-satisfiable QBF expression using tools like *qasp2qbf* [7] and calling a QBF solver to evaluate the satisfiability of the converted expression. The correctness of such an approach has been proved [7].

GDL and QASP use stable models for the semantics while QBF is based on classical models [7]. Converting directly from GDL to QBF is therefore challenging as it would require some form of completion technique [1, 15]. Thanks to existing work on converting from QASP to QBF [7], we do not need to deal with the completion task from scratch as long as we can convert from GDL to QASP.

3 TRANSFORMING GDL GAMES INTO QASP

We discuss how to transform the GDL description G for a two-player zero-sum turn-taking game to a QASP program. First, we need to map G to the temporal extended ASP program $Ext(G)$ according to Definition 2 and the domain of `mtdom` is $1 \ldots T_{max}$ while the domain of `tdom` is $1 \ldots T_{max} + 1$. For simplicity, let us assume the two players are called x and o respectively, and let g_{max} be the value a player would achieve if it wins the game (in most game descriptions, $g_{max} = 100$). The goal is to ensure that the QASP formula is true if, and only if, the original GDL game is x-winnable. We give a formal definition of two-player zero-sum turn-taking games and x-winnability in the context of GDL.

Definition 4. *A GDL game* (R, S_1, T, l, u, g) *is a* two-player zero-sum game *iff for any valid play sequence*

$$S_1 \xrightarrow{A_1} \ldots \xrightarrow{A_n} S_{n+1}$$

that terminates in n steps, for $(\mathtt{x}, v_x, S_{n+1}) \in g$ *and* $(\mathtt{o}, v_o, S_{n+1}) \in g$ *we have that* $v_x + v_o = g_{max}$*. A game is two-player zero-sum* turn-taking *iff it is a two-player zero-sum game and at any valid non-terminal game position, at most one player has more than one legal move.*

Definition 5. *A two-player zero-sum turn-taking GDL game G with semantics* (R, S_1, T, l, u, g) *is* x-winnable within N steps at the position S *iff the following recursive definition holds:*

(1) $S \in T$, $N \geq 0$, *and* $(\mathtt{x}, g_{max}, S) \in g$ *or,*
(2) $S \notin T$, $N > 0$, *and there exists* a_x *such that* $(\mathtt{x}, a_x, S) \in l$ *and for all* a_o *such that* $(\mathtt{o}, a_o, S) \in l$*, the game G at position* $u(\{a_x, a_o\}, S)$ *is x-winnable within* $N - 1$ *steps.*

In the latter case, such an action a_x *is called a* winning action for x at state S.

For the second item, it seems we are making the strong assumption assuming that in every round, the x player is taking the action before the o player. However, the outcome of the game would be the same if we let player o choose the action before player x because, in turn-taking games, there exists at least one player who has no more than one legal action. Hence, the quantification order of the actions at the same turn cannot affect the game result under optimal play.

Based on these definitions, we will first provide a general QASP encoding (GE) that can be applied to check the x-winnability of any two-player zero-sum turn-taking games. In section 3.3 we will then describe a more concise encoding for cases where the order of play is fixed, that is, if we can infer which player's turn it is in every round before the game starts (SE).

3.1 General Encoding (GE)

We introduce the predicate `terminated(T)`, meaning the game ended at or before timestamp T, along with the following two rules.

$$terminated(T)\ \texttt{:-}\ terminal(T). \tag{1}$$

$$terminated(T+1)\ \texttt{:-}\ terminated(T),\ tdom(T+1). \tag{2}$$

To ensure that in non-terminal states both players are making exactly one legal move per turn, we introduce the following rules.

$$\texttt{:-}\ does(P, M, T),\ not\ legal(P, M, T). \tag{3}$$

$$1\ \{does(P, M, T) : mdom(P, M)\}\ 1\ \texttt{:-}\ mtdom(T),\ role(P),\ not\ terminated(T). \tag{4}$$

$$\texttt{:-}\ terminated(T),\ does(P, M, T). \tag{5}$$

$$\{moveL(L, T) : ldom(L)\}\ \texttt{:-}\ mtdom(T). \tag{6}$$

For each $\mathtt{m_i}$ that is in the move domain of player o,

$$does(o, \mathtt{m_i}, T)\ \texttt{:-}\ legal(o, \mathtt{m_i}, T),\ not\ terminated(T),\ moveL(\rho_1, T), \ldots, moveL(\rho_j, T),\ not\ moveL(\mu_1, T),\ \ldots,\ not\ moveL(\mu_k, T). \tag{7}$$

where $\rho_1, \ldots, \rho_j$ are the 1 bits in the binary representation of $i - 1$, and $\mu_1, \ldots, \mu_k$ are the 0 bits in the binary representation of $i - 1$.

For this set of rules, (6) and (7) constitute a logarithmic encoding of the actions of player o that uses the idea of the so-called corrective encoding of propositional games [22]. `mdom(P,M)` encodes that `M` is in the move domain of the player `P`. This domain can be derived by the rules defining the `input` predicate (cf. Table 1) or, if not given, can be calculated and simplified with the help of the domain dependency graph [10] (chapter 14). `ldom` is the logarithmic move domain which represents the domain of the first parameter of `moveL`. Suppose the size of the move domain of player o is $|M|$, then `ldom` is defined over 1 to $|L| = \lceil \log_2 |M| \rceil$. We create $|M|$ rules of the form (7), one for each action m_i in the move domain. The logarithmic encoding ensures that o will make exactly one legal move before the terminal state without the need to introduce so-called cheating variables [11]. It is important to note that $|M|$ might be less than $2^{|L|}$, which means there might be some binary combinations of `moveL` that do not correspond to a legal action in the move domain. In this case, no action of player o can be generated by rule (7). However, (3) and (4) ensure that exactly one legal action of player o is generated in such a case.

Rule (5) is introduced to ensure that both players would not take any action after the game has terminated. Note that eliminating this rule would not affect the correctness of the encoding, but we believe that after the QASP is translated to QBF, this rule is converted to a set of binary clauses in the QBF which can trigger some unit propagation [5] for the QBF solver, which is beneficial.

The goal of player x is to achieve a utility of g_{max} after the game terminates. This is modeled with the following two rules.

$$:- terminated(T),\ not\ terminated(T-1),\ not\ goal(x, g_{max}, T). \quad (8)$$

$$:- terminated(1),\ not\ goal(x, g_{max}, 1). \quad (9)$$

It is possible that under some play sequences, the game does not terminate within T_{max} steps. We treat all these game sequences as a loss for player x:

$$:- 0\ \{terminated(T) : tdom(T)\}\ 0. \quad (10)$$

We denote the answer set program (1) $\cup$ ... $\cup$ (10) as P_{ge}.

3.2 Quantifier Prefix

The ground atoms of the program $Ext(G) \cup P_{ge}$ need to be correctly quantified in such a way that the QASP is satisfied if, and only if, the game G at the initial position is x-winnable. In most existing QBF encodings of specific games [6, 11, 22], the quantifier prefix of the QBF expression is of form

$$\exists E_0\ \forall U_1\ \exists E_1\ \forall U_2 ...\ \forall U_n\ \exists E_n$$

Here, n is the maximum number of rounds of the game, and the action variables of player x (resp. o) for round i are quantified by E_{i-1} (resp. U_i). A critical property of these game encodings is that the values of all non-action variables in $E_0 \cup \ldots \cup E_i$ can be uniquely determined by the assignment of action variables in $E_0 \cup \ldots \cup E_i \cup\ U_1 \cup \ldots \cup U_{i-1}$. This property ensures that the variables describing the state of the game for the first i rounds are fully determined by the actions in the first i rounds.

In handmade encodings, this property is normally satisfied based on human knowledge about the dependencies of variables in the QBF. In GDL, however, the set of ground atoms of the program $Ext(G)\ \cup\ P_{ge}$ changes across different games, which is why we need an automated quantifier construction method for the program $Ext(G)\ \cup\ P_{ge}$ that satisfies the property as long as G is a GDL description of a two-player zero-sum turn-taking game. Since $Ext(G)$ is a stratified program, there are many possible ways of quantifying $Ext(G)\ \cup\ P_{ge}$. In the following, we consider several ways in which a valid quantification method can be automatically generated.

To begin with, a naive method is obtained by quantifying does and moveL according to the temporal order while placing all the other atoms in the innermost existential quantifier block. This is formalized by the following definition.

Definition 6 (Naive Quantification Method). *Given a two-player zero-sum turn-taking game G described in GDL and the maximum number T_{max} of steps allowed in the game. Suppose $\mathcal{A}$ is the set of ground atoms of the program $Ext(G)\ \cup\ P_{ge}$ and the quantifier prefix $\boldsymbol{Q}_n$ of the program is of the form*

$$\boldsymbol{Q}_n = \exists E_0\ \forall U_1\ \exists E_1\ \forall U_2 \ldots\ \forall U_{T_{max}}\ \exists E_{T_{max}}$$

Then, for each $a\ \in\ \mathcal{A}$, the quantifier block it belongs to is determined by the following four rules:

1. *If $a = \texttt{moveL}(L, t)$ for some l, t then $a \in U_t$.*
2. *If $a = \texttt{does}(x, M, t)$ for some M, t then $a \in E_{t-1}$.*
3. *If $a = \texttt{does}(o, M, t)$ for some M, t then $a \in E_t$.*
4. *Otherwise, $a \in E_n$.*

For performance reasons, the ground atoms of the program should be quantified as early as possible; in existing QBF encodings of specific games [6, 11, 22], variables are always quantified in the earliest possible quantifier block. Using the semantics of GDL and the consequences of Theorem 1, once we fix the values of does and moveL with timestamp no greater than T, we can consider any ground atom in $Ext(G) \cup P_{ge}$ that is not an instance of does or moveL and is extended by a timestamp $t \leq T$. Whether this atom is in a stable model of $Ext(G) \cup P_{ge}$ can be uniquely determined by the values of does and moveL. One natural quantification method using this observation is to group the ground atoms by "timestamp".

Definition 7 (Time Based Quantification Method). *Given a two-player zero-sum turn-taking game G described in GDL and the maximum number T_{max} of steps allowed in the game. Suppose $\mathcal{A}$ is the set of ground atoms of the program $Ext(G)\ \cup\ P_{ge}$ and the quantifier prefix $\boldsymbol{Q}_t$ of the program is of the form*

$$\boldsymbol{Q}_t = \exists E_0\ \forall U_1\ \exists E_1\ \forall U_2 \ldots\ \forall U_{T_{max}}\ \exists E_{T_{max}}$$

Then, for each $a\ \in\ \mathcal{A}$, the quantifier block it belongs to is determined by the following four rules:

1. *If $a = \texttt{moveL}(L, t)$ for some L, t then $a \in U_t$.*
2. *If $a = \texttt{does}(x, M, t)$ for some M, t then $a \in E_{t-1}$.*
3. *If a is not of form $\texttt{moveL}(L, t)$ or $\texttt{does}(x, M, t)$ and is extended by a timestamp t, then $a \in E_t$.*
4. *Otherwise, $a \in E_0$.*

While an improvement over the naive quantification method, the time-based one is still not ideal. For example, consider the predicate legal. In a valid game description, legal at timestamp T does not depend on the values of does at time T. In the time-based quantification method, legal is always quantified later than does with the same timestamp. We believe that it will be beneficial if we use this dependency information and quantify legal before does. Note that the same analysis also applies to predicates like goal, terminal and true. In order to addresses this issue, we introduce a quantification method based on atom dependency.

Definition 8 (Atom dependency). *Consider a ground logic program of P with grounding $\mathcal{A}$ in smodels format [31], then for two atoms $p, q\ \in\ \mathcal{A}$ we say that q depends on p iff the following recursive definition holds: Program P contains a rule such that the variable q appears in the head of the rule and the variable p appears in the body of the rule; or, there exists an atom $z \in\ \mathcal{A}$ such that q depends on z and z depends on p. We denote that q depends on p by $p \rightarrow\ q$, and that q does not depend on p by $p \not\rightarrow\ q$.*

Definition 9 (Dependency Based Quantification Method). *Given a two-player zero-sum turn-taking game G described in GDL with semantics (R, S_1, T, l, u, g) such that $R = \{x, o\}$ and the maximum number T_{max} of steps allowed in the game. Suppose $\mathcal{A}$ is the set of ground atoms of the program $Ext(G)\ \cup\ P_{ge}$ and the quantifier prefix $\boldsymbol{Q}_d$ of the program is of form*

$$\boldsymbol{Q}_d = \exists E_0\ \forall U_1\ \exists E_1\ \forall U_2 \ldots\ \forall U_{T_{max}}\ \exists E_{T_{max}}$$

Then, for each $a\ \in\ \mathcal{A}$, the quantifier block it belongs to is determined by the following three rules:

1. *If $a = \texttt{moveL}(L, t)$ for some L, t then $a\ \in\ U_t$.*
2. *If $a = \texttt{does}(x, M, t)$ for some M, t then $\texttt{a}\ \in\ E_{t-1}$.*
3. *Otherwise, $a \in E_t$ with t the maximum $1 \leq t \leq T_{max}$ such that $\texttt{moveL}(L, t) \rightarrow a$ for some $\texttt{moveL}(L, t) \in \mathcal{A}$. If no such t exists then $a \in E_0$.*

The correctness of the encoding GE is given by the following theorems. Here, Theorem 2 shows that every valid game-playing sequence with length no greater than T_{max} such that player x wins corresponds to some stable models of the answer set program $Ext(G) \cup P_{ge}$. Theorem 3 shows that every stable model of the answer set program $Ext(G) \cup P_{ge}$ corresponds to a valid game-playing sequence with length no greater than T_{max} and player x wins. Theorem 5 shows the correctness of the overall encoding. Theorem 2 and 3 are similar to Theorems 2, 3 in the single-player paper [32], hence we follow the notations defined in that paper.

Theorem 2. *Given a two-player zero-sum turn-taking game G described in GDL with semantics (R, S_1, T, l, u, g) such that $R = \{x, o\}$ and the maximum number T_{max} of steps allowed in the game. Suppose there is a valid playing sequence of length $0 \leq n \leq T_{max}$*

$$S_1 \xrightarrow{A_1} S_2 \xrightarrow{A_2} \ldots S_n \xrightarrow{A_n} S_{n+1}$$

such that $S_{n+1} \in T$, and $(r, g_{max}, S_{n+1}) \in g$. Then, $Ext(G) \cup P_{ge}$ admits at least one answer set such that

$$does(x, A_1(x), 1),\ does(o, A_1(o), 1),\ does(x, A_2(x), 2),$$
$$does(o, A_2(o), 2), \ldots, does(x, A_n(x), n),\ does(o, A_n(o), n)$$

are the only positive instances of does.

Proof. Because of the consequence of Theorem 1, the fact that the program $Ext(G)$ is stratified, and the fact that only S_{n+1} is a terminal state, there must be a stable model for the program $Ext(G) \cup (1) \cup (2) \cup (3) \cup (4) \cup (5) \cup (10)$ that admits the above set of does as the only positive instances of this predicate in the model. Note that because of the logarithmic encoding of the actions of player o, for each of that player's legal action there exists a binary combination of moveL that would derive the legal action, hence this stable model also satisfies rules (6) and (7). Since S_{n+1} is a terminal position with x as the winner, this stable model moreover satisfies rules (8) and (9); hence, it is also a stable model of program $Ext(G) \cup P_{ge}$. □

Theorem 3. *Given a two-player zero-sum turn-taking game G described in GDL with semantics (R, S_1, T, l, u, g) such that $R = \{x, o\}$ and the maximum number T_{max} of steps allowed in the game. Suppose the program $Ext(G) \cup P_{ge}$ admits an answer set $\mathcal{M}$, then there exists $0 \leq n \leq T_{max}$ such that*

$$does(x, A_1(x), 1),\ does(o, A_1(o), 1),\ does(x, A_2(x), 2),$$
$$does(o, A_2(o), 2), \ldots, does(x, A_n(x), n),\ does(o, A_n(o), n)$$

are the only positive instances of does *in $\mathcal{M}$, and there is a valid game playing sequence*

$$S_1 \xrightarrow{A_1} S_2 \xrightarrow{A_2} \ldots S_n \xrightarrow{A_n} S_{n+1}$$

such that $S_{n+1} \in T$,and $(r, g_{max}, S_{n+1}) \in g$.

Proof. Firstly note that does only appears in the head of rules (4) and (7) of $Ext(G) \cup P_{ge}$. Because of rules (1), (2), (4), (10) and the logarithmic encoding of the actions of player o such that the legal action of player o can always be expressed by a binary combination of *moveL*, we conclude that there exists a sequence of actions $A_1^{does}(1), \ldots, A_n^{does}(n)$ $(0 \leq n \leq T_{max})$ in the stable model $\mathcal{M}$ such that these are the only positive instances of does. Because of the consequence of Theorem 1 and rules (4) and (10), there must exist states $S_1, \ldots, S_{n+1}$ such that $S_1 \xrightarrow{A_1} S_2 \xrightarrow{A_2} \ldots S_n \xrightarrow{A_n} S_{n+1}$, and $S_{n+1} \in T$. Because of rules (8) and (9), $(r, g_{max}, S_{n+1}) \in g$. □

Theorem 4. *Given a two-player zero-sum turn-taking game G described in GDL with semantics (R, S_1, T, l, u, g) such that $R = \{x, o\}$ and the maximum number T_{max} of steps allowed in the game. For any valid play sequence*

$$S_1 \xrightarrow{A_1} S_2 \xrightarrow{A_2} \ldots S_i \xrightarrow{A_i} \ldots$$

such that $i \leq T_{max}$, the ASP program

$$P = Ext(G) \cup P_{ge} \cup A_1^{does}(1) \cup \ldots \cup A_{i-1}^{does}(i-1)$$

has the following properties:

1. *For any stable model $\mathcal{M}$ of P, $S_i = \{f \mid true(f, i) \in \mathcal{M}\}$.*
2. *For any stable model $\mathcal{M}$ of P, $G \cup S_i^{true} \models legal(r, a)$ iff $legal(r, a, i) \in \mathcal{M}$.*
3. *For any a_x such that $G \cup S_i^{true} \models legal(x, a_x)$ and any a_o such that $G \cup S_i^{true} \models legal(o, a_o)$, both a_x and a_o are in the move domain and there exists a binary combination of $moveL(L, i)$ that can generate $does(o, a_o, i)$.*
4. *If P does not have a stable model, then S_i is not x-winnable within $T_{max} - i + 1$ steps.*

Proof (Sketch). Firstly, it is trivial to see that if we add rule (1) and (2) to $Ext(G)$, Theorem 1 still holds. Using the fact that only does and moveL appear in the head of $(3) \cup (4) \cup \ldots \cup (10)$, properties 1 and 2 are a direct consequence of Theorem 1. Since the move domain is a superset of all possible actions of both players at all positions, it is a superset of all the legal actions at position S_i, hence both a_x and a_o are in the move domain. The logarithmic encoding ensures that any action in the move domain can be expressed by a binary combination of moveL, and the legal action of player o at position S_i can definitely be expressed as a binary combination of moveL as well. Hence, property 3 holds. Finally, we prove the contrapositive statement of property 4. If S_i is x-winnable within $T_{max} - i + 1$ steps, then there must exist a valid play sequence of no more than $T_{max} - i + 1$ steps with x achieving the value of g_{max} at the end. This means that there exists a valid play sequence with no more than T_{max} steps with $A_1^{does}(1), \ldots, A_{i-1}^{does}(i-1)$ as the first $i-1$ actions such that x achieves the value of g_{max} at the end. Using Theorem 2, we know that P must have at least one stable model. This establishes property 4. □

Theorem 5. *Given a two-player zero-sum turn-taking game G described in GDL with semantics (R, S_1, T, l, u, g) such that $R = \{x, o\}$ and the maximum number T_{max} of steps allowed in the game. Let $\mathbf{Q}$ be the quantifier prefix of the program $Ext(G) \cup P_{ge}$ constructed using either Definition 6, 7 or 9. The QASP program*

$$\mathbf{Q}\ Ext(G) \cup P_{ge}$$

is satisfiable iff the game G at position S_1 is x-winnable within T_{max} steps.

Proof (Sketch). We only show the case when $\mathbf{Q}$ is constructed based on the dependency-based quantification method according to Definition 9. Rules 1 and 2 in this definition ensure that the quantification order of the action atoms in P matches the temporal order of the original game G. The third rule ensures that no atom in $Ext(G) \cup P_{ge}$ that depends on an action is quantified before that

action atom. Considering the definition of x-winnable, this theorem is then a direct consequence of Theorems 2, 3 and 4. □

Thanks to Theorem 5 and the existing technique of converting any QASP program to an equal-satisfiable QBF expression [7], we can use a QBF solver to solve all two-player zero-sum turn-taking games described in GDL.

3.3 Strictly Turn-Taking Encoding (SE)

For many turn-taking games, which player's turn it is solely depends on the round number. For example, in Connect-4 [11], we know that player x (resp. o) is taking turn at the odd (resp. even) steps. In other games, however, such as Dots and Boxes [2], whose turn it is is not determined by the round of the game but the game state. We call turn-taking games in which the player taking turn is only related to the step of the game as *strictly turn-taking* games. Note that in the encoding GE, we create an existential quantifier block and a universal quantifier block for each round of the game. For strictly turn-taking games, we only need a single existential or universal quantifier block for each step. For example, in Connect-4, player x takes turn in the first step while player o has only one legal action noop. Generally speaking, we can use the following rule to force player o into taking the only available action in this round, and so the universal quantifier block U_1 can be removed:

$$does(o, M, 1) \text{ :- } legal(o, M, 1),\ not\ terminated(1). \qquad (11)$$

With such a technique, we can obtain an encoding with *fewer* quantifier blocks when the game is strictly turn-taking provided that we can determine the turn-taking player for each step of the game. For general GDL games, this can be determined with the help of automated theorem proving methods [33].

4 EXPERIMENTAL RESULTS

We evaluate different variants of the translation method on a number of popular two-player GDL games. We consider both games that are strictly turn-taking and games that are not: Generalized 4×4 Tic-Tac-Toe (GT-1-1, and GT-2-2) [6], Connect-3 (C-3) [11], Connect-4 (C-4) [11], Breakthrough (BT) [14], and Dots and Boxes (D&B) [2]. We test the efficiency of our translation method on different configurations of the games. For C-3, C-4, BT, and D&B, different configurations refer to different board sizes whereas in GT-1-1 and GT-2-2, different configurations indicate different winning domino shapes [6]. The GDL rules of these games are created and modified based on the GGP Base repository [12]. Among these games, C-3, C-4, BT, and GT-1-1 are not only strictly turn-taking but in fact, the two players take turns alternatingly. GT-2-2 is strictly turn-taking too, but both players take two consecutive turns. As mentioned above, Dots and Boxes is not strictly turn-taking.

Valid GDL games must be finite and terminating. We employ iterative deepening to solve any game G [22, 30] by gradually increasing the depth considered in our encoding, T_{max}, and checking if G is x-winnable within T_{max} steps from the initial position. For a game G, we denote by μ_G the length of the longest valid playing sequence with player x as the winner.

Iterative deepening is complete and terminates as long as we are provided with μ_G: If G is not x-winnable for any depth not greater than μ_G then it is not x-winnable at all. Computing μ_G for some GDL games can be automated [33], but this is beyond the scope of this paper and so we used human domain knowledge [30] to determine it.

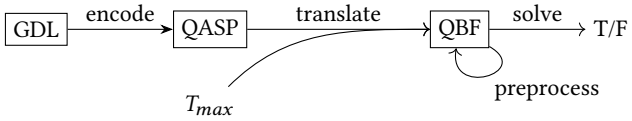

Figure 1: Solving games with QBF

Our overall approach is given in Figure 1.[1] The GDL to QASP translation is done with one of the following encodings: *naive quantification (GN)*, *time-based quantification (GT)*, *dependency-based quantification (GD)* and *dependency-based SE quantification (SD)* when applicable. We use *qasp2qbf* [7] for the QASP to QBF translation. For the QBF preprocessing, we use *bloqqer* [3]. Finally, we solve the preprocessed formulas with state-of-the-art QBF solvers *Caqe* [24] and *DepQBF* [19]. *Caqe* is an expansion-based QBF solver that uses the technique of counterexample guided abstraction refinement (CEGAR) [24] while *DepQBF* is a search-based QBF solver that applies the QCDCL algorithm [19]. Preliminary experiments showed that alternative implementations for preprocessors (*HQSPre* and *QratPre++* [20, 35]) and solvers (*Qute* and *RAReQS* [16, 23]) lead to drastically worse performance. For the sake of clarity, they are therefore omitted from our formal experimental analysis.

For comparison purposes, we include the performance of *Caqe* and *DepQBF* on instances generated by the existing general translation method from BDDL to QBF (BL) [30]. We also add a comparison with a Minimax solver with transposition tables [26] (Minx) implemented in C++ that uses Prolog [34] as the GDL reasoner for legal actions [27]. All the experiments were run on a Latitude 5430 laptop with a solving time limit of 1000 seconds and a preprocessing time limit of 500 seconds. None of the instances we tested timed out during the translation and preprocessing phase.

In Table 2, we record the value of μ_G and the smallest depth T_{max} of the game in **bold** if the game can be proved to be x-winnable within T_{max} steps by any solver under any encoding within the solving time limit. If the game is not x-winnable at all depths, and the game can be proved not to be x-winnable within μ_G steps by any solver under any encoding in the solving time limit, we let $T_{max} = \mu_G$ and record T_{max} in plain format. For games that can neither be proved to be x-winnable at some depth nor be proved not to x-winnable for all depth within the solving time limit, we record the maximum refuted depth T_{max} [30] (i.e., the maximum T_{max} such that the game is proved not to be x-winnable at depth T_{max} by any solver under any encoding within the solving time limit) of the unsolved games in *italic* format.

In Table 2, we also record the solving time of *Caqe* and *DepQBF* for each game at depth T_{max} under each translation method and the solving time of the minimax solver. We also record the *bloqqer* preprocessing time (in seconds) for instances generated by the method GD in column Bloq.

[1]The interested reader can find the code used in our experiments here: https://github.com/hharryyf/gdl2qbf

					DepQBF					Caqe					Minx
Game	**Config**	μ_G	T_{max}	**Bloq**	**GN**	**GT**	**GD**	**SD**	**BL**	**GN**	**GT**	**GD**	**SD**	**BL**	
	2×5	21	21	5.72	0.97	1.09	0.74	0.74	*	26.02	0.41	**0.13**	0.17	*	0.36
	2×6	29	**15**	5.34	10.45	7.54	7.15	8.13	*	62.94	14.93	6.44	8.18	*	**2.86**
BT	3×4	19	19	9.39	0.62	0.38	0.59	0.46	*	12.26	0.52	**0.11**	0.12	*	1.09
	3×5	31	*19*	19.71	*	*	*	*	*	*	*	827.60	655.64	*	**92.41**
	4×4	25	25	50.50	370.27	200.19	109.23	69.46	*	*	131.09	19.13	**17.37**	*	106.20
	4×4	15	**9**	0.61	0.07	0.06	**0.03**	0.05	1.17	0.08	0.08	0.06	0.07	11.56	0.21
C-3	5×5	25	**9**	0.83	0.16	0.12	**0.09**	0.16	69.01	0.11	0.32	0.29	0.27	339.30	0.75
	6×6	35	**9**	1.26	**0.15**	0.41	0.31	0.37	392.07	0.49	1.04	0.57	0.45	709.80	1.37
	4×4	15	15	0.91	0.53	0.46	0.57	0.63	189.97	2.03	0.46	**0.30**	0.37	327.94	1.42
C-4	5×5	25	*21*	2.52	646.22	665.92	370.33	563.55	*	*	370.83	**135.52**	167.21	*	517.50
	6×6	35	*19*	3.80	*	*	*	*	*	*	*	593.76	**381.30**	*	*
	elly	15	**7**	3.65	1.57	1.75	3.26	1.71	5.40	3.44	1.79	**0.73**	1.21	70.89	9.75
	fat.	15	15	15.22	317.69	284.75	**188.89**	237.42	*	*	*	396.69	372.24	*	307.38
GT-1-1	knob.	15	15	15.53	616.85	546.85	**363.81**	664.69	*	*	*	690.24	*	*	*
	skin.	15	15	14.29	340.83	480.12	380.18	496.28	*	*	*	*	*	*	**206.59**
	tip.	15	**9**	5.98	12.31	6.06	11.01	10.12	35.48	6.03	16.33	**2.44**	3.43	756.81	30.94
	elly	14	**6**	2.63	0.08	0.10	0.18	0.08	n/a	0.17	0.08	0.07	**0.03**	n/a	0.23
	fat.	14	14	12.77	327.41	285.92	**148.66**	285.90	n/a	*	781.80	313.55	208.85	n/a	*
GT-2-2	knob.	14	**6**	2.36	0.07	0.11	0.13	0.09	n/a	0.08	0.11	0.04	**0.03**	n/a	0.83
	skin.	14	14	12.12	798.62	*	**378.89**	872.61	n/a	*	*	548.99	633.94	n/a	662.32
	tip.	14	**6**	2.57	0.06	0.05	0.09	0.11	n/a	0.16	0.08	0.06	**0.04**	n/a	3.87
D&B	2×2	12	**12**	1.57	3.48	11.57	5.13	n/a	n/a	47.75	19.58	5.89	n/a	n/a	**0.63**
	2×3	17	17	5.35	*	*	*	n/a	n/a	*	*	599.74	n/a	n/a	**15.06**

Table 2: Solving time (in seconds) for DepQBF and Caqe under different translation and quantification methods (GN, GT, GD, SD, BL). The solving time for a minimax + transposition table solver on the original GDL game is also included. * means the solver timed out after 1000 seconds. n/a means the game is not strictly turn-taking or the game is not available in the BDDL project.

	GD-plain	GD-bloq	SD-plain	SD-bloq	BL-plain	BL-bloq
# var	29957	1072	17525	988	2366	431
# ∀	95	41	45	42	67	65
# cl	93986	10459	51107	9744	6875	3715
# qb	39	19	19	19	21	21
DepQBF (s)	557.79	0.59	403.00	0.46	*	*
Caqe (s)	150.36	0.11	29.46	0.12	*	*

Table 3: Instance information for the game Breakthrough-3x4 of depth 19.

4.1 Comparing GE and SE

For strictly turn-taking games, theorem-proving methods can be applied to calculate the player whose turn it is at each step of the game [33]. As a result, we can design a more concise encoding for games that are strictly turn-taking. It might be expected that for these games, SE outperforms GE because it has fewer quantifier blocks. For the particular instance Breakthrough-3×4 at depth 19, we count the number of variables (# *var*), the number of clauses(# *cl*), the number of universal variables (# ∀), and the number of quantifier blocks (# *bl*) of the instance under the encoding and quantification methods *GD* and *SD* with (-bloq) and without (-plain) preprocessing. We observe from Table 3 that without preprocessing, the number of variables, the number of clauses, the number of universal variables, and the number of quantifier blocks in *GD-plain* are all approximately twice the ones in *SD-plain*. As a consequence, the instance created by *GD-plain* should be more difficult to solve than *SD-plain* for both *DepQBF* and *Caqe*. However, after bloqqer preprocessing, the instance created by *GD-bloq* has a similar size to that of *SD-bloq*, and both *DepQBF* and *Caqe* take a similar amount of time to solve these preprocessed instances. More importantly, the preprocessed instances are much simpler to solve than the unpreprocessed ones. A similar observation holds for other games. We observe from Table 2 that for both *DepQBF* and *Caqe*, instances created by *SD* are not simpler to solve than *GD* after the instances are preprocessed by *bloqqer*. Since *bloqqer* preprocessing is critical for the solver's performance, and calculating the player taking turns using the theorem-proving [33] method is time-consuming, our experimental results somewhat surprisingly suggest that the strictly turn-taking property is not beneficial for the solver's performance in practice thanks to the existence of QBF preprocessors.

4.2 Comparing Different Quantification Methods

We can observe from Table 2 that under GE, the three different quantification methods *GN*, *GT*, and *GD* have similar performance for the search-based QBF solver *DepQBF* [19]. By using any of

the three different quantification methods, *DepQBF* can solve most of the small-sized games in a feasible amount of time. However, for the CEGAR-based QBF solver *Caqe* [24], the solver's performance is closely related to the quantification method. *Caqe* can solve significantly more small-sized games when we apply GE with the dependency-based quantification method than the naive or the time-based one. In fact, for the 23 games we tested, *Caqe* plus the encoding and quantification method *GD* gives the best overall performance. Hence, we believe that the dependency-based quantification method is an important part of the encoding. A likely explanation for the performance difference between *GD* and others is that in *GD*, atoms like `legal` and `terminal` at timestamp T are quantified before `does` at timestamp T since neither predicate depends on `does` in a valid GDL description. By quantifying these atoms earlier than `does`, the solver is forced to fix the value of these predicates before branching on `does` at timestamp T. This can ensure that the QBF solver does not branch on a `does` variable that corresponds to an illegal action at timestamp T, thus pruning the search space. Our result that *GD* is more important to *Caqe* than to *DepQBF* could be explained by the effect of unit propagation [5] in *DepQBF*. Although in *GN* and *GT*, atoms like `legal` and `terminal` at timestamp T are quantified after `does` at timestamp T, their value may be uniquely derived by unit propagation after the `does` instances for timestamps 1 to $T-1$ have been assigned. In some cases, even if we apply the *GN* or *GT* quantification method, *DepQBF* would not branch on a `does` variable that corresponds to an illegal action. In comparison, *Caqe* is not a search-based solver, which is why such a simplification procedure is unlikely to happen, and as a result, *GD* is more important to the performance of *Caqe* than to *DepQBF*.

4.3 Comparison With Other Methods

We first compare the encoding and quantification method *GD* with existing general QBF encoding methods for turn-taking games. Prior to our work, there were no encoding methods that could convert all two-player zero-sum turn-taking games with perfect information that can be expressed in the general game description language GDL to QBF. One of the most general frameworks was the translation method to convert general two-player zero-sum turn-taking *board games* that can be described in the board-game specification language [30]. We can observe in Table 2 that for both *DepQBF* and *Caqe*, the instances created by our translation method *GD* are significantly easier to solve than the ones created by the BDDL method (BL). However, this does not necessarily imply that our translation method is strictly "better" than the BDDL-based translation. This is not just because we construct QBF instances from GDL as opposed to BDDL, but also because the method discussed in our paper only works for small-sized games when the grounding of the ASP program is of feasible size. The authors of the BDDL-based translation emphasized the aim to create a correct and concise translation from two-player zero-sum turn-taking board games to QBF that potentially works for large games too. We can observe from Table 3 that the size of the encoding created by the BDDL method is smaller than that resulting from our translation.

Next, we compare the encoding and quantification method GD with the minimax and transposition table solver. The result of this comparison is mixed. We observe from Table 2 that for games from the C-3, C-4, GT-1-1, and GT-2-2 families after the instances generated by *GD* are preprocessed by *bloqqer*, both *Caqe* and *DepQBF* use less time to solve the majority of the QBF instances than the minimax method to solve the original GDL games. For the BT family, *Caqe* can solve all the instances generated by the method *GD* and outperforms the minimax solver in around half of the instances. However, for the D&B family, the minimax solver outperforms our encoding method *GD* by 1 to 2 orders of magnitude. Note that the above comparison only considers the solving time of the QBF solvers on the preprocessed instances. In practice, the *bloqqer* preprocessing time should be considered as well. For the 23 games in the experiment, the *bloqqer* preprocessing time never exceeds 60 seconds. Even if the *bloqqer* preprocessing time is included (column Bloq in Table 2), the encoding and quantification method *GD* is more efficient for most games from the C-4, GT-1-1, and GT-2-2 families. Most importantly, although our translation method outperforms the minimax and transposition table solver in only some games, the performance of the QBF method is comparable with search-based game-solving methods. This means that our QBF-based method can be an alternative approach for general game players to evaluate endgame positions of small-sized games. This is in line with the conclusion of the work on solving single-player GDL games with ASP while generalizing it to two-player games [32].

5 CONCLUSION AND FUTURE WORK

Game solving in General Game Playing is a relatively less explored field. Before this paper, one of the first results in general game "solving" was to solve general single-player GDL games with ASP [32]. In this paper, we have described a translation method that generalizes the techniques for the single-player case to two-player zero-sum turn-taking perfect information games. We have developed a method of constructing the quantifier prefix of the program with the use of the atom dependencies. We have shown that with the help of existing tools like qasp2qbf [7], the QASP instances can then be converted to QBF and solved by state-of-the-art QBF solvers. Experimental results have shown that the efficiency of the state-of-the-art QBF solvers on these translated instances is acceptable when the size of the games is relatively small. Although this paper only focuses on the two-player case, our approach can be easily extended to solving multiplayer games under the paranoid assumption [25] (i.e., all the other players are cooperatively playing against a single agent). Our method can also be applied to estimating the maximin or minimax bound [18] of two-player zero-sum simultaneous move games.

One shortcoming of our method is that the translation size is related to the grounding of the temporal-extended answer set program, which can be very large for many games. Although we have shown in the experimental results that the encoding size is not directly related to the solving time, creating a more concise translation from GDL to QBF is important when the program's grounding is too large. One challenging future work is to see if the lifted-encoding techniques that have been used in the BDDL method [30] can be applied in the context of two-player zero-sum turn-taking GDL games as well to create a more concise translation in cases when the GDL specification has a large grounding.

REFERENCES

[1] Vernon Asuncion, Fangzhen Lin, Yan Zhang, and Yi Zhou. 2012. Ordered completion for first-order logic programs on finite structures. *Artificial Intelligence* 177 (2012), 1–24.

[2] Elwyn R Berlekamp. 2000. The dots and boxes game: sophisticated child's play.

[3] Armin Biere, Florian Lonsing, and Martina Seidl. 2011. Blocked Clause Elimination for QBF. In *Automated Deduction – CADE-23*, Nikolaj Bjørner and Viorica Sofronie-Stokkermans (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 101–115.

[4] Yngvi Bjornsson and Hilmar Finnsson. 2009. Cadiaplayer: A simulation-based general game player. *IEEE Transactions on Computational Intelligence and AI in Games* 1, 1 (2009), 4–15.

[5] Marco Cadoli, Andrea Giovanardi, and Marco Schaerf. 1998. An algorithm to evaluate quantified Boolean formulae. *AAAI/IAAI* 98 (1998), 262–267.

[6] Diptarama, Ryo Yoshinaka, and Ayumi Shinohara. 2016. QBF Encoding of Generalized Tic-Tac-Toe. In *Proceedings of the 4th International Workshop on Quantified Boolean Formulas (QBF 2016) co-located with 19th International Conference on Theory and Applications of Satisfiability Testing (SAT 2016), Bordeaux, France, July 4, 2016 (CEUR Workshop Proceedings, Vol. 1719)*, Florian Lonsing and Martina Seidl (Eds.). CEUR-WS.org, Bordeaux, France, 14–26. https://ceur-ws.org/Vol-1719/paper1.pdf

[7] Jorge Fandinno, François Laferrière, Javier Romero, Torsten Schaub, and Tran Cao Son. 2021. Planning with incomplete information in quantified answer set programming. *Theory and Practice of Logic Programming* 21, 5 (2021), 663–679.

[8] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. 2012. Answer set solving in practice. *Synthesis lectures on artificial intelligence and machine learning* 6, 3 (2012), 1–238.

[9] Michael Genesereth and Nathaniel Love. 2006. General game playing: Game description language specification. https://www.cs.uic.edu/~hinrichs/papers/love2006general.pdf

[10] Michael Genesereth and Michael Thielscher. 2014. General game playing. *Synthesis Lectures on Artificial Intelligence and Machine Learning* 8, 2 (2014), 1–229.

[11] Ian P. Gent and Andrew G. Rowley. 2003. Encoding Connect-4 using quantified Boolean formulae. In *2nd International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*. Kinsale, Ireland, 78–93.

[12] GGP. 2023. GGP Base Repository. http://games.ggp.org/base/.

[13] Adrian Goldwaser and Michael Thielscher. 2020. Deep Reinforcement Learning for General Game Playing. *Proceedings of the AAAI Conference on Artificial Intelligence* 34 (04 2020), 1701–1708. https://doi.org/10.1609/aaai.v34i02.5533

[14] Kerry Handscomb. 2001. 8× 8 game design competition: The winning game: Breakthrough... and two other favorites. *Abstract Games Magazine* 7 (2001), 8–9.

[15] Tomi Janhunen and Ilkka Niemelä. 2011. *Compact Translations of Non-disjunctive Answer Set Programs to Propositional Clauses*. Springer Berlin Heidelberg, Berlin, Heidelberg, 111–130. https://doi.org/10.1007/978-3-642-20832-4_8

[16] Mikoláš Janota, William Klieber, Joao Marques-Silva, and Edmund Clarke. 2016. Solving QBF with counterexample guided refinement. *Artificial Intelligence* 234 (2016), 1–25.

[17] Frédéric Koriche, Sylvain Lagrue, Éric Piette, and Sébastien Tabary. 2016. General game playing with stochastic CSP. *Constraints* 21 (2016), 95–114.

[18] Kevin Leyton-Brown and Yoav Shoham. 2022. *Essentials of game theory: A concise multidisciplinary introduction*. Springer Nature.

[19] Florian Lonsing and Uwe Egly. 2017. DepQBF 6.0: A Search-Based QBF Solver Beyond Traditional QCDCL. In *Automated Deduction – CADE 26*, Leonardo de Moura (Ed.). Springer International Publishing, Cham, 371–384.

[20] Florian Lonsing and Uwe Egly. 2019. QRATPre+: Effective QBF Preprocessing via Strong Redundancy Properties. In *Theory and Applications of Satisfiability Testing – SAT 2019*, Mikoláš Janota and Inês Lynce (Eds.). Springer International Publishing, Cham, 203–210.

[21] Joao Marques-Silva, Inês Lynce, and Sharad Malik. 2021. Conflict-driven clause learning SAT solvers. In *Handbook of Satisfiability*. IOS press, 133–182.

[22] Valentin Mayer-Eichberger and Abdallah Saffidine. 2020. Positional Games and QBF: The Corrective Encoding. In *Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12178)*, Luca Pulina and Martina Seidl (Eds.). Springer, Alghero, Italy, 447–463. https://doi.org/10.1007/978-3-030-51825-7_31

[23] Tomáš Peitl, Friedrich Slivovsky, and Stefan Szeider. 2019. Dependency learning for QBF. *Journal of Artificial Intelligence Research* 65 (2019), 181–208.

[24] Markus N. Rabe and Leander Tentrup. 2015. CAQE: A Certifying QBF Solver. In *2015 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, Austin, 136–143. https://doi.org/10.1109/FMCAD.2015.7542263

[25] Jahn-Takeshi Saito and Mark HM Winands. 2010. Paranoid proof-number search. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*. IEEE, Copenhagen, Denmark, 203–210.

[26] Jonathan Schaeffer. 1989. The history heuristic and alpha-beta search enhancements in practice. *IEEE transactions on pattern analysis and machine intelligence* 11, 11 (1989), 1203–1212.

[27] Stephan Schiffel and Yngvi Björnsson. 2014. Efficiency of GDL reasoners. *IEEE Transactions on Computational Intelligence and AI in Games* 6, 4 (2014), 343–354.

[28] Stephan Schiffel and Michael Thielscher. 2007. Fluxplayer: A Successful General Game Player. In *Proceedings of the 22nd National Conference on Artificial Intelligence - Volume 2 (AAAI'07)*. AAAI Press, Vancouver, British Columbia, Canada, 1191–1196.

[29] Stephan Schiffel and Michael Thielscher. 2010. A Multiagent Semantics for the Game Description Language. In *Agents and Artificial Intelligence*, Joaquim Filipe, Ana Fred, and Bernadette Sharp (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 44–55.

[30] Irfansha Shaik and Jaco van de Pol. 2023. Concise QBF Encodings for Games on a Grid (extended version). arXiv preprint 2303.16949. http://arxiv.org/abs/2303.16949

[31] Tommi Syrjänen. 2000. Lparse 1.0 user's manual. http://www.tcs.hut.fi/Software/smodels/lparse.ps

[32] Michael Thielscher. 2009. Answer Set Programming for Single-Player Games in General Game Playing. In *Logic Programming*, Patricia M. Hill and David S. Warren (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 327–341.

[33] Michael Thielscher and Sebastian Voigt. 2010. A Temporal Proof System for General Game Playing. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*, Maria Fox and David Poole (Eds.). AAAI Press, Atlanta, Georgia, USA, 1000–1005. https://doi.org/10.1609/aaai.v24i1.7646

[34] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. 2012. SWI-Prolog. *Theory and Practice of Logic Programming* 12, 1-2 (2012), 67–96.

[35] Ralf Wimmer, Sven Reimer, Paolo Marin, and Bernd Becker. 2017. HQSpre — An Effective Preprocessor for QBF and DQBF. In *Proceedings, Part I, of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 10205*. Springer-Verlag, Berlin, Heidelberg, 373–390. https://doi.org/10.1007/978-3-662-54577-5_21