Artificial Intelligence 187–188 (2012) 1–30

Contents lists available at SciVerse ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

Automated verification of state sequence invariants in general game playing

Sebastian Haufe [a], Stephan Schiffel [b], Michael Thielscher [c,*]

[a] Department of Computer Science, Dresden University of Technology, Germany
[b] School of Computer Science, Reykjavík University, Iceland
[c] School of Computer Science and Engineering, The University of New South Wales, Australia

ARTICLE INFO

Article history:
Received 13 April 2011
Received in revised form 8 March 2012
Accepted 6 April 2012
Available online 11 April 2012

Keywords:
General game playing
Knowledge representation
Answer set programming

ABSTRACT

A *general* game player is a system that can play previously unknown games given nothing but their rules. Many of the existing successful approaches to general game playing require to generate some form of game-specific knowledge, but when current systems establish knowledge they rely on the approximate method of playing random sample matches rather than formally proving knowledge. In this paper, we present a theoretically founded and practically viable method for automatically verifying properties of games whose rules are given in the general Game Description Language (GDL). We introduce a simple formal language to describe game-specific knowledge as *state sequence invariants*, and we provide a proof theory for verifying these invariants with the help of Answer Set Programming. We prove the correctness of this method against the formal semantics for GDL, and we report on extensive experiments with a practical implementation of this proof system, which show that our method of formally proving knowledge is viable for the practice of general game playing.

1. Introduction

General Game Playing is concerned with the development of systems that understand the rules of previously unknown games and learn to play these games well without human intervention. Today considered to be a Grand Challenge for Artificial Intelligence [14], this endeavour requires the combination of methods from a variety of sub-disciplines, including reasoning, search, game playing, and learning [29,28]. Broad interest in this research area was sparked by the inauguration of the annual AAAI General Game Playing Competition, for which Michael Genesereth and his Stanford Logic Group developed the general Game Description Language (GDL) [25]. With its recent extension to incomplete information games [37], the description language allows to formalise and communicate the rules of arbitrary finite n-player games to a general game-playing system. GDL rules are logical axioms, and a plain, Prolog-like inference mechanism suffices for a basic player to be able to make legal moves [14].

1.1. The value of knowledge

A general game-playing system can solve simple games by brute-force search. Moreover, recent research has shown that Monte Carlo-based methods provide a successful form of selective blind search to play arbitrary unknown games without the need to learn an explicit game-specific strategy [4,26,19].

* Corresponding author.
E-mail addresses: sebastian.haufe@gmx.net (S. Haufe), stephans@ru.is (S. Schiffel), mit@cse.unsw.edu.au (M. Thielscher).

0004-3702/$ – see front matter
http://dx.doi.org/10.1016/j.artint.2012.04.003

Moving from blind to informed search, however, is a great endeavour in general game playing as it requires a player to automatically analyse the bare rules of previously unknown games with the goal to extract and exploit game-specific knowledge. The value of gaining knowledge is recognised across a variety of approaches to general game playing, as the following examples demonstrate.

1. Kuhlmann et al. [22] show how search heuristics can be created on the basis of structural elements such as boards, markers and pieces. These elements are identified by finding *invariants*. For example, a ternary expression in the game description, say `cell(X,Y,Piece)`, is considered to denote a two-dimensional board provided it has one *output* argument, that is, which cannot have two different values simultaneously.
2. Clune [6] shows how automatically generated evaluation functions are improved by determining the stability of candidate features. An example of a stable property are corner stones in the game of Othello, which will not change once they have been placed.
3. Further structural knowledge that proves useful for better game play is identified in [34,35]. For example, the second argument in an expression `money(Player,Amount)` is considered to denote a *quantity* if this argument is unique and ordered by a transitive and antisymmetric relation. Knowledge of this kind is used in [35] to create a goal distance measure based on Fuzzy Logic.
4. Even approaches that originate in pure blind Monte Carlo search have been shown to profit from the use of knowledge: Finnsson and Björnsson [9] demonstrate how the identification of different piece *types* in chess-like games allows to determine the relative importance of state features, which can then be used to bias move selection during random search.

While successful general game-playing systems like the aforementioned rely on the ability to acquire game-specific knowledge, none of them actually attempt to prove it. Rather they generate random sample matches to test whether a property is violated at some point, and then rely on the correctness of this informed guess, as frankly admitted by Kuhlmann et al. [22]:

> We have mentioned several situations in which we needed to prove an invariant about states of the game. . . . Rather than proving these invariants formally, which would be time-consuming, our agent uses an approximate method to become reasonably certain that they hold. [22, p. 1460]

Of course this runs the risk of basing one's strategy on erroneous beliefs about a game, which may lead to serious blunders when a player cuts off an easy win or a straightforward loss from its search space, or when a player employs an inappropriate evaluation function. The purpose of this paper is to demonstrate that, contrary to widespread belief, it is viable for a general game-playing system to prove knowledge formally.

Moreover, assisting general game-playing systems with the acquisition of knowledge about a new game would not be the only benefit of having an automatic verification method in general game playing. Such a tool can also support game design. In practice it frequently happens that a new game is specified by a set of rules that are syntactically valid but erroneous in that they do not describe the exact intended game. Later (in Section 3.1) we will give an illustrative example of a defective game description whose problems were not detected until the game was actually used at the 2006 AAAI Competition, which caused quite some disturbance among the participants and the organisers alike. The availability of an automated proof method would enable game designers to ensure that their game specifications satisfy basic desired properties, such as that two pieces can never occupy the same square, that players are never left without a legal move in nonterminal states, or that a game designed to be turn-taking and zero-sum does indeed have these properties.

The first method of automatically proving properties for general games has been given in [32]. Yet their approach requires to search the entire set of reachable positions in a game. This renders the method unsuited for both practical play and game design, because in either case the interest lies in games that are far too complex to be searched completely in reasonable time.

1.2. Overview of results

In this paper, we present the first practical method of rigorously proving game-specific knowledge given nothing but the formal rules of a game. Our approach allows systems to automatically verify *state sequence invariants*. These are temporally extended yet local properties of games that can be proved by induction rather than by complete search.[1] An example of a simple state sequence invariant is, "for all `X`,`Y` there is a unique `Piece` such that `cell(X,Y,Piece)` is true." An example of a state sequence invariant that refers to two consecutive states is, "a corner of an Othello board that is occupied by a light (dark) piece will be occupied by a light (dark) piece in the next state." All examples of game-specific knowledge mentioned above in Section 1.1 can be expressed as state sequence invariants.

Our specific contributions can be summarised as follows.

[1] *Local* means properties that can be established without searching through a large part of the entire game tree, and *temporally extended* means properties that concern one or more successive game states. An example of a *global* property would be the existence of a winning strategy for a player, which in general requires to search much—if not all—of the game tree.

1. We define syntax and semantics of a simple formal language for game-specific knowledge, for which we combine elements from the Game Description Language and Temporal Logic.
2. We define a new, formal semantics for GDL by which game descriptions are interpreted as labelled state transition systems.
3. Using the paradigm Answer Set Programming [13,12], we develop a proof theory for verifying game-specific knowledge against a given GDL specification, and we formally prove this method to be correct.
4. Because the practice of general game playing typically requires a player to search through large sets of potentially valid and useful properties, we extend our basic proof method so as to allow a general game player to systematically search and verify (or reject) multiple properties at once.
5. We report on systematic experiments with an implementation of our method, for which we have integrated a state-of-the-art Answer Set Solver [10] with the successful knowledge-based general game player described in [35].

Before we proceed we stress that in this paper we are only concerned with automatically *proving* properties. Learning what type of properties may actually be worth proving is beyond the scope of this paper, and we refer to the literature mentioned in Section 1.1 [6,9,22,35] and other publications [7,17], which comprise an extensive body of work on various types of game-specific knowledge that helps a general game player find good heuristics.

The paper proceeds as follows. In the next section, we recapitulate the basic syntax of GDL. Thereafter, in Section 3, we introduce a formal language for state sequence invariants, and we define a new, state transition-based semantics for GDL. In Section 4, we develop a proof theory for automatically verifying potential invariants against GDL game descriptions. In Section 5, we present the aforementioned extensions to the method. In Section 6, we discuss our implementation in detail and report on systematic experiments that demonstrate the practical viability of our approach. We conclude in Section 7.

2. Formalisation of games: the game description language

While finite state machines are the natural model for finite multiplayer games, most games have huge state spaces and therefore cannot be directly specified as a finite state machine in practice. This motivated the development of the general Game Description Language (GDL) [14], which can be used to provide a fully axiomatic, compact description of any finite and deterministic n-player game ($n \geqslant 1$) with perfect information. On the one hand, the language is declarative and easy to understand and use by humans; on the other hand, it can be processed fully automatically by a general game-playing system. The syntax follows that of *normal logic programs* (see e.g., [23]):

Definition 1. A *term* is either a variable, or a function symbol applied to terms as arguments (functions with no arguments are called *constants*).

An *atom* is a predicate symbol with terms as arguments.

A *literal* is an atom or its negation.

A *clause* is of the form $h :- b_1, \ldots, b_n .$, where h (the *head*) is an atom and $b_1, \ldots, b_n$ (the *body*) are literals ($n \geqslant 0$), with the meaning that $b_1, \ldots, b_n$ together imply h.

A *logic program* is a finite set of clauses.

A word on the notation: In this paper we will be largely concerned with embedding GDL game descriptions into answer set programs. For this reason we will use the standard Prolog syntax for GDL, where variables are indicated by uppercase letters (see Fig. 1 for an example). This is in contrast to the more customary KIF-notation of GDL introduced in [14]. The KIF-syntax also allows disjunctions in clause bodies, but these can be easily transformed into normal logic program clauses [23]. Throughout the paper, we will use "clause" and "(game) rule" interchangeably.

As a language especially designed for game descriptions, GDL uses a few pre-defined predicate symbols shown in Table 1. A further standard predicate is **distinct** `(X,Y)`, which means syntactic inequality of the two arguments and which can only appear in the body of a clause.[2]

GDL-II GDL has recently been extended to games with randomness and imperfect information [37,38] using these two additional keywords:

- Constant **random** is a pre-defined role that models Nature and plays randomly;
- Predicate **sees** `(R,P)`—to be read as: role `R` perceives `P`—is used to control the information that players have about the game state.

Perfect-information games can be expressed in GDL-II by the general rule

sees`(R,move(R2,M))` `:-` **role**`(R)`, **does**`(R2,M)`.

[2] The semantics of this predicate is given by tacitly assuming the unary clause `distinct`(s,t)., for every pair s,t of syntactically different ground terms.

Table 1
The keywords of GDL.

role (R)	R is a player
init (F)	F holds in the initial position
true (F)	F holds in the current position
legal (R,M)	player R has legal move M
does (R,M)	player R does move M
next (F)	F holds in the next position
terminal	the current position is terminal
goal (R,N)	player R gets goal value N

According to this clause, each player R always "sees" any move M by any player R2 and, hence, has complete state knowledge throughout a game given that each game description provides a complete description of the initial position.

Example *(The board game Quarto).* The two-player turn-taking game of "Quarto" [20] is played on a 4×4 game board. It uses 16 different pieces, one for each combination of four characterising binary attributes (e.g. short/tall, red/blue, etc.). Initially, the board is empty and one player starts by selecting one of the pieces for placement by the other player. The players take turns repeating this procedure with yet unplaced pieces until either no more pieces are available (in which case the game ends in a draw) or one player wins by having completed a horizontal, vertical or diagonal line of four pieces with at least one shared attribute (e.g., they are all red). A complete GDL specification for Quarto is shown in Fig. 1. The two players are called *r1*, *r2*, and the 16 pieces are represented by constants *p0000*, *p0001*, *p0010*, ..., *p1111*, where each bit position stands for one of the attributes. The actions are

- *select*(P): piece P gets selected for placement,
- *place*(P, X, Y): piece P is placed on free board cell (X, Y), and
- *noop*: an action without effect, performed by the player who currently has no control.

The game positions are represented using these state components, henceforth called *fluents*:

- *cell*(X, Y, P): board cell (X, Y) contains piece P (where $P = b$ for blank cells),
- *pool*(P): piece P is available for selection,
- *sctrl*(R): role R currently has control to select a piece,
- *pctrl*(R): role R currently has control to place a piece, and
- *selected*(P): the last action has been to select piece P.

Lines 1 and 2 in Fig. 1 define the names of the two players and fluents that compose the initial state. Clauses 4–6 for predicate **legal** (R,M) define the preconditions for player R to take move M relative to what is **true** in the current position: A player can select a piece from the pool (line 4) and place a selected piece on a blank cell (line 5) when he has control to do so; otherwise the player can only do *noop*, a move without effect[3] (line 6).

Clauses 8–15 for predicate **next** (F) provide a complete axiomatisation of all fluents F that compose a successor state relative to what is **true** in the current position and what each player **does**. Specifically, two frame axioms say respectively that any piece P remains in the pool if it is not being selected (lines 8–9) and that all cells X,Y keep their mark P unless a player decides to place a piece on that very cell (lines 12 and 13). As for the actual effects of the moves, line 10 says selecting a piece P causes selected(P) to become true in the next state; line 11 says that when placing a piece on a cell, then the contents of this cell changes accordingly; and lines 14 and 15 define the progression of the two control fluents.

According to line 17, the two players see each other's moves, which induces perfect information. A state is terminal if either there is a line of pieces with a common attribute (line 19, where the winning criterion is encoded by the auxiliary predicate line) or the board has no empty position (line 20, in conjunction with clause 28). The player who completes a line wins the game with maximal payoff 100 (line 22) and leaves his opponent with minimal payoff 0 (line 24). A draw is indicated by both players obtaining payoff 50 in case of a completely filled board where the winning criterion is not satisfied (line 23).

According to the informal semantics given in [14,25], a GDL specification G is to be understood as follows.

1. The derivable instances of **role** (R) define the players.
2. The initial state is composed of the derivable instances of **init** (F).

[3] This is the usual way of modelling turn-taking games in GDL, which assumes all players to move simultaneously.

```
role(r1).     role(r2).     init(cell(1,1,b)).  ...  init(cell(4,4,b)).
init(sctrl(r1)).            init(pool(p0000)).  ...  init(pool(p1111)).

legal(R,select(P))    :- true(sctrl(R)), true(pool(P)).
legal(R,place(P,X,Y)) :- true(pctrl(R)), true(selected(P)), true(cell(X,Y,b)).
legal(R,noop)         :- role(R), not true(sctrl(R)), not true(pctrl(R)).

next(pool(P))     :- true(pool(P)), not does(r1,select(P)),
                                    not does(r2,select(P)).
next(selected(P)) :- does(R,select(P)).
next(cell(X,Y,P)) :- does(R,place(P,X,Y)).
next(cell(X,Y,P)) :- true(cell(X,Y,P)), does(R,place(Q,X1,Y1)), !=(X,Y,X1,Y1).
next(cell(X,Y,P)) :- true(cell(X,Y,P)), does(R,select(Q)).
next(sctrl(R))    :- true(pctrl(R)).
next(pctrl(R1))   :- true(sctrl(R2)), otherrole(R1,R2).

sees(R,move(R2,M)) :- role(R), does(R2,M).

terminal :- line.
terminal :- not boardopen.

goal(R,100) :- line, placedlast(R).
goal(R, 50) :- not line, not boardopen, role(R).
goal(R,  0) :- line, otherrole(R,R1), placedlast(R1).

placedlast(R) :- true(sctrl(R)).

boardopen :- true(cell(X,Y,b)).

line :- row.
line :- column.
line :- diagonal.

row :-      true(cell(1,Y,P1)), true(cell(2,Y,P2)),
            true(cell(3,Y,P3)), true(cell(4,Y,P4)), sameattr(P1,P2,P3,P4).
column :-   true(cell(X,1,P1)), true(cell(X,2,P2)),
            true(cell(X,3,P3)), true(cell(X,4,P4)), sameattr(P1,P2,P3,P4).
diagonal :- true(cell(1,1,P1)), true(cell(2,2,P2)),
            true(cell(3,3,P3)), true(cell(4,4,P4)), sameattr(P1,P2,P3,P4).
diagonal :- true(cell(1,4,P1)), true(cell(2,3,P2)),
            true(cell(3,2,P3)), true(cell(4,1,P4)), sameattr(P1,P2,P3,P4).

sameattr(P1,P2,P3,P4) :- nthbit(N,P1,Bit), nthbit(N,P2,Bit),
                         nthbit(N,P3,Bit), nthbit(N,P4,Bit).

!=(X1,Y1,X2,Y2) :- index(X1), index(Y1), index(X2), index(Y2), distinct(X1,X2).
!=(X1,Y1,X2,Y2) :- index(X1), index(Y1), index(X2), index(Y2), distinct(Y1,Y2).

nthbit(1,p0000,0).     index(1).     otherrole(r1,r2).
nthbit(2,p0000,0).     index(2).     otherrole(r2,r1).
...                    index(3).
nthbit(4,p1111,1).     index(4).
```

Fig. 1. A GDL specification of the game Quarto.

3. In order to determine the legal moves of a player in any given state, this state has to be encoded first, using the keyword **true**. More precisely, let $S = \{f_1, \ldots, f_n\}$ be a finite state (e.g., the derivable instances of **init** (F) at the beginning), then G is extended by the unary clauses

   true(f_1).
   ...
   true(f_n).

   Those instances of **legal** (R,A) which are derivable from this extended program define all legal actions A for player R in state S.

4. In the same way, the clauses for **terminal** and **goal**(R,N) define termination and outcome (i.e., a goal value N for player R) *relative* to the encoding of a given state.
5. Determining a position update requires the encoding of the current position along with clauses representing a joint move. Specifically, if players $r_1, \ldots, r_k$ make moves $a_1, \ldots, a_k$, then

   $$\textbf{does}(r_1, a_1)\,.$$
   $$\ldots$$
   $$\textbf{does}(r_k, a_k)\,.$$

   must be added to G, and then the derivable instances of **next**(F) compose the updated state.
6. In the same way, the derivable instances of **sees**(R,P) describe all players' percepts.

This informal description will be made precise in Section 3.2. We complete this introduction by recalling the restrictions that have been imposed on GDL in [25] in order to ensure that any valid rule description can be unambiguously interpreted as a game.

Definition 2. The *dependency graph* for a logic program G is a directed, labelled graph whose nodes are the predicate symbols that occur in G and where there is a *positive* edge $p \xrightarrow{+} q$ if G contains a clause $p(\vec{s}) :- \ldots, q(\vec{t}), \ldots.$, and a *negative* edge $p \xrightarrow{-} q$ if G contains a clause $p(\vec{s}) :- \ldots, \textbf{not}\, q(\vec{t}), \ldots.$ We say p *depends on* q in G if there is a path from p to q in the dependency graph of G.

A *valid GDL specification* is a finite set of clauses G where

- **role** only appears in facts (i.e., clauses with empty body) or in the body of clauses;
- **init** only appears as head of clauses and does not depend on any of **true**, **legal**, **does**, **next**, **sees**, **terminal**, or **goal**;
- **true** only appears in the body of clauses;
- **does** only appears in the body of clauses, and none of **legal**, **terminal**, or **goal** depends on **does**;
- **next** and **sees** only appear as head of clauses.

Moreover, the description G and its dependency graph Γ must satisfy the following.

1. There are no cycles involving a negative edge in Γ, that is, G must be *stratified* [1,11];
2. Each variable in a clause occurs in at least one positive atom in the body, that is, G must be *allowed* [24];
3. If p and q occur in a cycle in Γ and G contains a clause

   $$p(s_1, \ldots, s_m) :- b_1(\vec{t}_1), \ldots, q(v_1, \ldots, v_k), \ldots, b_n(\vec{t}_n)$$

   then for every $i \in \{1, \ldots, k\}$,
   - v_i is variable-free, or
   - v_i is one of $s_1, \ldots, s_m$, or
   - v_i occurs in some $\vec{t}_j$ $(1 \leqslant j \leqslant n)$ such that b_j does not occur in a cycle with p in Γ.

The last condition imposes a restriction on the combination of function symbols and recursion to ensure finiteness and decidability in all cases.

It is straightforward to verify that the rules in Fig. 1 satisfy all requirements of a valid GDL description. The imposed restrictions on the keywords are a consequence of their use to define the semantics of a GDL game. If, for example, **legal** were allowed to depend on **does**, as in

```
legal(R1,select(P)) :- otherrole(R1,R2), not does(R2,select(P)).
```

then a player would not be able to decide whether a move is legal based on his knowledge of the current position alone. To convey an intuitive understanding of the necessity of the further restrictions, some examples of invalid GDL specifications follow.

Stratification. Non-stratified rules, as in

```
boardopen   :- not boardclosed.
boardclosed :- not boardopen.
```

may not admit a unique (i.e., unambiguous) model.

Allowedness Clauses that are not allowed, as in

```
next(selected(P)) :- does(R,select(Q)).
```

may induce positions that are composed of an unbounded number of fluents.

Recursion restriction. Clauses that do not obey the recursion restriction, as in

```
placedlast(selected(X)) :- placedlast(X).
```

may allow terms to grow to unbounded size by recursion.

3. Sequence invariants over game descriptions

3.1. The importance of sequence invariants in GDL

Recall the Quarto rules in lines 20 and 28, respectively, from the game description in Fig. 1 (page 5):

```
terminal  :- not boardopen.
boardopen :- true(cell(X,Y,b)).
```

Suppose these two clauses were replaced by

```
terminal :- not true(cell(X,Y,b)), index(X), index(Y).
```

At first glance, this seems not to alter the meaning, namely, that the game terminates if there is no blank cell. In fact, however, there is a crucial difference regarding the implicit quantification of the variables `X` and `Y`. While the original two clauses imply **terminal** if there do *not* exist `X` and `Y` such that **true**`(cell(X,Y,b))`, the alternative rule implies **terminal** if there do exist `X` and `Y` such that *not* **true**`(cell(X,Y,b))`. The first placement of a piece at *any* cell yields a state which satisfies the body of one ground instance of the alternative clause (as the marked cell is not blank anymore) and hence untruly renders this state terminal, whereas the original clauses imply termination only when *all* cells are marked. The organisers of the General Game Playing Competition in 2006 used a GDL specification for the game of Othello [16] with a similar defect, which caused quite some disturbance, first among the participants and then among the organisers themselves. A proof system that allows to formally verify game descriptions would have been of invaluable assistance to the game designers in order to prevent such mishaps. The bug that we just introduced in the Quarto game description, say, would be immediately detected when attempting to prove the following intended property:

If there is a blank cell and no completed line, then Quarto is not terminated. (1)

In addition to assisting the game design, a proof system can also help a general game-playing system to discover valuable information about a previously unknown game. This information can then be used, for example, to choose an appropriate search algorithm or to construct a suitable, game-dependent heuristic. In the following we will motivate a class of game properties which allows an efficient verification and is expressive enough to comprise many interesting properties of a game description, including the previously mentioned one.

To begin with, we consider the class of properties which make statements about single states of a game, which we will call *state invariants*. They are "local", which means that they can be verified for all reachable states by an analysis of the GDL rules rather than by a complete search through the whole game tree. This covers many interesting properties, including (1). As another example, the Quarto property

Each cell contains at most one piece. (2)

allows a general game player to infer the existence of a board structure, which is valuable knowledge to construct a good heuristics for playing the game [22,6,35]. However, many interesting properties cannot be expressed by referring to a single state. Consider, for example,

If no player can place a piece now, then in the next state one player can do so. (3)

This property is not a state invariant due to the inherent reference to subsequent game states. However, it can be seen as a state *sequence* invariant with degree 1, meaning that its formulation requires a "lookahead" of exactly one joint move.

In the following we will define a formal language over the syntax of GDL that allows the formulation of state sequence invariants. The language is restricted in that no infinite sequences and no quantification over sequences is allowed, which turns out to be a beneficial tradeoff between expressibility and efficient verifiability. A simple and elegant way to obtain such a language is by extending GDL with the unary operator "$\bigcirc$" borrowed from Temporal Logic (see, for example, [21]) to refer to successor game states.

Definition 3. We define $\mathcal{P}$ to be the set of all ground atoms $p(\vec{t})$ using the predicate and function symbols of a valid GDL specification G such that $p \notin \{\textbf{init}, \textbf{next}\}$ and p does not depend on **does** in G. Then the set $\mathcal{V}_G$ of (*state*) *sequence invariants over* G is the smallest set with

- $\mathcal{P} \subseteq \mathcal{V}_G$;

- Let $\varphi[\vec{X}]$ denote a formula obtained from φ by replacing arbitrary ground terms with variables from $\vec{X} = (X_1, \ldots, X_k)$. Furthermore, let $D_{\vec{X}} = D_{X_1} \times \cdots \times D_{X_k}$ for *finite* sets (of *domain elements*) $D_{X_1}, \ldots, D_{X_k}$.
  If $\varphi, \varphi_1, \varphi_2 \in \mathcal{V}_G$, then also the following are in $\mathcal{V}_G$:
  - $\neg\varphi$, $\varphi_1 \wedge \varphi_2$, and $\varphi_1 \vee \varphi_2$;
  - $(\exists \vec{X} : D_{\vec{X}})\varphi[\vec{X}]$;
  - $(\exists_{l..u} \vec{X} : D_{\vec{X}})\varphi[\vec{X}]$, for each $l \in \mathbb{N}$ and $u \in \mathbb{N} \cup \{\infty\}$ s.t. $l \leqslant u$;
  - $\bigcirc\varphi$.

We also define, over the syntax tree t_φ of $\varphi \in \mathcal{V}_G$, the *degree of* φ, denoted $\deg(\varphi)$, to be the maximal number of $\bigcirc$-occurrences on paths from the root of t_φ to its leaves.

Since predicates over **init** and **next** are excluded, the unary predicate **true** provides the only means for referring to fluents and thus to states, which keeps the language clear and simple. Predicates that depend on **does** are excluded for technical reasons which will be pointed out at the end of Section 4.1. We allow restricted quantification, by explicit specification of a *finite* domain for each variable; and we use counting quantifiers of the form $(\exists_{l..u} \vec{X} : D_{\vec{X}})\varphi$ to give a lower (l) and upper (u) bound for the number of ground instances $\vec{t}$ for a vector of variables $\vec{X}$ such that $\varphi[\vec{X}/\vec{t}]$ is true. Here, $\varphi[\vec{X}/\vec{t}]$ denotes the formula which is obtained from φ by replacing all variables in $\vec{X}$ with the respective instances in $\vec{t}$. If $u = \infty$ then there is no upper bound. Modality $\bigcirc\varphi$ means "φ holds next," and the degree of a formula is the maximal "nesting" of this modal operator. The binary connective $\supset$ as well as quantifier $(\forall \vec{X} : D_{\vec{X}})$ are defined via the usual macros, and the terms "state sequence invariant" and "formula" will be used interchangeably. In the remainder, φ, ψ, and ρ (possibly with subscripts) are always used to refer to state sequence invariants.

As an example, consider the previously mentioned property (1) (page 7). Denoting the set of board indices by $I = \{1, 2, 3, 4\}$, it can be formulated via the following formula of degree 0[4]:

$$\big((\exists X, Y : I)\ \mathit{true}\big(\mathit{cell}(X, Y, b)\big) \wedge \neg \mathit{line}\big) \supset \neg \mathit{terminal}.$$

Property (2) can be formulated via a formula of degree 0, too, if we denote the set of pieces by $D_P = \{\mathit{p0000}, \mathit{p0001}, \ldots, \mathit{p1111}\}$:

$$(\forall X, Y : I)\ (\exists_{0..1} P :\ D_P)\ \mathit{true}\big(\mathit{cell}(X, Y, P)\big). \tag{4}$$

Property (3), however, refers to two consecutive states and hence requires a formula of degree 1:

$$\neg\big(\exists R : \{\mathit{r1}, \mathit{r2}\}\big)\ \mathit{true}\big(\mathit{pctrl}(R)\big) \supset \bigcirc\big(\exists R : \{\mathit{r1}, \mathit{r2}\}\big)\ \mathit{true}\big(\mathit{pctrl}(R)\big). \tag{5}$$

3.2. A formal semantics for the game description language

The formal treatment of state sequence invariants requires a formal semantics for GDL, which we define next, making precise what is only informally described in [25]. Our semantics is based on viewing a GDL game description as an *answer set program* (ASP), for which we can use the notion of an *answer set*, which provides a modern way of characterising the models of logic programs with negation [5,2,13] (see, e.g., [12] for a general introduction to ASPs and answer sets).

Definition 4. (See [13].) Given a set of clauses G and a set of ground atoms $\mathcal{A}$, let $G^{\mathcal{A}}$ be the set of negation-free implications, $h :\!- b_1, \ldots, b_k .$, obtained by taking all ground instances of clauses in G and

- deleting all clauses with a negative body literal **not** b_i such that $b_i \in \mathcal{A}$,
- deleting all negative body literals from the remaining clauses.

Then $\mathcal{A}$ is an *answer set* for G if and only if $\mathcal{A}$ is the least model for $G^{\mathcal{A}}$.

A useful property of answer sets is to provide a unique model whenever the underlying program is stratified [13], as is always the case with valid GDL game descriptions (cf. Definition 2). In the following, by $G \vdash p$ we denote that ground atom p is contained in this unique answer set for a stratified set of clauses G.

Any game description in GDL contains a finite set of function symbols, including constants, which implicitly determines a (usually infinite) set of ground terms Σ. As indicated in Section 2, interpreting a GDL specification requires to encode positions and joint moves as logic program facts. To this end, we introduce two abbreviations: $S^{\mathtt{true}}$, where $S = \{f_1, \ldots, f_n\} \subseteq \Sigma$ is a finite set of ground terms; and $A^{\mathtt{does}}$, where $A : \{r_1, \ldots, r_k\} \mapsto \Sigma$ is an assignment of moves to players. These two sets of atoms are defined as follows.

[4] In quantifiers, when two variables X and Y have the same domain D, we abbreviate $(X, Y) : D \times D$ by $X, Y : D$.

$$\begin{aligned} S^{\mathtt{true}} &\stackrel{\text{def}}{=} \big\{\mathbf{true}(f_1)\,.,\ldots,\ \mathbf{true}(f_n)\,.\big\},\\ A^{\mathtt{does}} &\stackrel{\text{def}}{=} \big\{\mathbf{does}\big(r_1, A(r_1)\big)\,.,\ldots,\mathbf{does}\big(r_k, A(r_k)\big)\,.\big\}. \end{aligned} \tag{6}$$

We are now prepared to formally define how each valid GDL specification determines a unique state transition system as the underlying game model.

Definition 5. The *semantics* of a valid GDL specification G is given by this state transition system $(R, S_{init}, T, l, u, g)$:

- $R = \{r : G \vdash \mathbf{role}(r)\}$—the *roles*;
- $S_{init} = \{f : G \vdash \mathbf{init}(f)\}$—the *initial* position;
- $T = \{S : G \cup S^{\mathtt{true}} \vdash \mathbf{terminal}\}$—the *terminal* positions;
- $l = \{(r, a, S) : G \cup S^{\mathtt{true}} \vdash \mathbf{legal}(r, a)\}$—the *legality* relation;
- $u(A, S) = \{f : G \cup A^{\mathtt{does}} \cup S^{\mathtt{true}} \vdash \mathbf{next}(f)\}$—the *update* function;
- $\mathcal{I}(A, S) = \{(r, p) : G \cup A^{\mathtt{does}} \cup S^{\mathtt{true}} \vdash \mathbf{sees}(r, p)\}$—the *information* relation;
- $g = \{(r, v, S) : G \cup S^{\mathtt{true}} \vdash \mathbf{goal}(r, v)\}$—the *goal* relation;

for all finite subsets $S \subseteq \Sigma$ and assignments $A : R \mapsto \Sigma$ where Σ is the set of all ground terms that can be built with the function symbols (including the constants) that occur in G.

Example. The Quarto rules in Fig. 1 (page 5) entail the initial state

$$S_{init} = \big\{sctrl(r1), pool(p0000), pool(p0001), \ldots, pool(p1111), cell(1,1,b), \ldots, cell(4,4,b)\big\}.$$

Adding $S_{init}^{\mathtt{true}}$ to the set of clauses allows to derive the legal moves of both players in S_{init}:

$$\big\{\big(r1, select(p0000), S_{init}\big), \ldots, \big(r1, select(p1111), S_{init}\big), (r2, noop, S_{init})\big\} \subseteq l.$$

Consider, say, $A = \{r1 \mapsto select(p0000), r2 \mapsto noop\}$, then further adding $A^{\mathtt{does}}$ to the logic program in Fig. 1 allows to infer the updated state, $u(A, S_{init})$:

$$\big\{pctrl(r2), selected(p0000), pool(p0001), \ldots, pool(p1111), cell(1,1,b), \ldots, cell(4,4,b)\big\}.$$

The syntactic restrictions imposed on valid GDL specifications justify the restriction to finite sets $S^{\mathtt{true}}$ and $A^{\mathtt{does}}$, as the following proposition shows.

Proposition 6. *Suppose G is a valid GDL specification, then*

1. $\{r : G \vdash \mathbf{role}(r)\}$ *is finite.*
2. $\{f : G \vdash \mathbf{init}(f)\}$ *is finite.*
3. $\{f : G \cup A^{\mathtt{does}} \cup S^{\mathtt{true}} \vdash \mathbf{next}(f)\}$ *is finite.*

Proof.

1. By Definition 2, all clauses in G with keyword **role** in their head are facts. These facts must be variable-free since G is allowed according to Definition 2, and hence the unique answer set for G includes just these finitely many ground instances of **role**.
2. According to Definition 2, keyword **init** only appears as head of clauses in G. The recursion restriction in Definition 2 ensures that if **init** depends on other predicates then no clause for these predicates can introduce new function symbols through recursion. Hence, the arguments in derivable instances for **init** have bounded size. Moreover, these terms must be grounded in a finite set of ground facts since G is allowed. This implies that the unique answer set for G contains only finitely many ground instances of **init**.
3. According to Definition 2, keyword **next** only appears as head of clauses in G. The recursion restriction in Definition 2 ensures that no clause for predicates on which **next** depends can introduce new function symbols through recursion. Hence, the arguments in derivable instances for **next** have bounded size. Moreover, these terms must be grounded in a finite set of ground facts since $G \cup A^{\mathtt{does}} \cup S^{\mathtt{true}}$ is finite and allowed, given that $A^{\mathtt{does}}$ and $S^{\mathtt{true}}$ are finite sets of ground facts. This implies that the answer set for any $G \cup S^{\mathtt{true}} \cup A^{\mathtt{does}}$ contains only finitely many ground instances of **next**. □

According to this proposition, only states that are finite can be reached from the initial state in a game described by a valid GDL specification. It is worth noting, however, that this does not imply that the set of reachable states itself is finite. As a matter of fact, GDL is expressive enough to describe any Turing machine as a "game" using clauses like the following.

```
init(head_position(0)).
next(head_position(succ(X))) :- true(head_position(X)),
                                does(tm,move_forward).
```

This clause for **next** describes an unbounded growth of `head_position(succ`^{n}`(0))` through state transitions, which corresponds to a one-way infinite tape. Hence, reachability of states is generally undecidable in GDL.

3.3. A formal semantics for sequence invariants

In order to define the formal meaning of sequence invariants that have degree $n > 0$, we first need to introduce successive state transitions of length n. Single state transitions are based on the formal semantics of the GDL given in Definition 5 on page 9. There is a transition from state S to state S' if S is not terminal and can be updated to S' with respect to a move assignment A which comprises a legal move for each of the players. Successive state transitions are then composed of multiple single state transitions. This is formally stated as follows.

Definition 7. For the semantics $(R, S_{init}, T, l, u, g)$ of a valid GDL specification and arbitrary finite $S, S' \subseteq \Sigma$, we write $S \xrightarrow{A} S'$ if the following holds:

- $A : R \mapsto \Sigma$ is such that $(r, A(r), S) \in l$ for each $r \in R$,
- $S' = u(A, S)$, and
- $S \notin T$.

We call $S_0 \xrightarrow{A_0} S_1 \xrightarrow{A_1} \ldots \xrightarrow{A_{m-1}} S_m$ (where $m \in \mathbb{N}$) a (*state*) *sequence*, sometimes abbreviated as $(S_0, S_1, \ldots, S_m)$ when reference to $A_0, A_1, \ldots, A_{m-1}$ is not needed. The *length* of a sequence $(S_0, \ldots, S_m)$ is m. Moreover, a state S_m is called *reachable* if there is a sequence $(S_{init}, \ldots, S_m)$.

Intuitively, a sequence invariant φ with degree n is true in a state S_0 if and only if all "relevant" sequences $(S_0, \ldots, S_m)$ satisfy φ. Clearly, all sequences with $m = n$ are relevant, and sequences where $m > n$ are irrelevant since they provide no more information (regarding φ) than their respective initial subsequences of length n. Also irrelevant are sequences with $m < n$ that can be extended by a legal transition, as they are contained in sequences with greater length. However, two types of sequences with $m < n$ cannot be extended and thus need to be considered:

- *Terminated Sequences* (i.e. that end in a terminal state). These are relevant for entailment, lest arbitrary formulas of the form $\psi \wedge \bigcirc \rho$ be considered true in any terminal state S_t regardless of the truth of ψ (since no sequence of length $\geqslant 1$ exists in S_t).
- *Nonplayable Sequences* (i.e. that end in a nonterminal state with no legal move for at least one player). Although they influence entailment, we neglect nonplayable sequences for the moment and defer the discussion on this issue to Section 5.2.

Terminated sequences could in principle be extended by a pseudo joint move ϵ that defines a transition from each terminal state S_t into S_t itself, that is, $S_t \xrightarrow{\epsilon} S_t$. Every terminated sequence could thus be extended to length n, which would allow to give a semantics for invariants over sequences of length n only. However, this has unintended side effects. For example, implications of the shape $\neg\varphi \supset \bigcirc\varphi$ (like, e.g., formula (5) (page 8)) for Quarto would always be false because they are never true in a terminal state S_t that satisfies $\neg\varphi$. Similar considerations with other pseudo continuations of terminal states lead to equally non-verifiable albeit possibly valid sequence invariants. The following definition of entailment takes into account all of our foregoing considerations.

Definition 8. Let G be a valid GDL specification. A sequence $(S_0, \ldots, S_m)$ is called *n-max* if it is of length n, or if it is shorter and ends in a terminal state. Let S_0 be a state and φ be a formula such that $\deg(\varphi) = n$. We say that S_0 *satisfies* φ (written $S_0 \vDash \varphi$) if for all n-max sequences $S_0 \xrightarrow{A_0} \ldots \xrightarrow{A_{m-1}} S_m$ $(m \leqslant n)$ we have that $(S_0, \ldots, S_m) \vDash \varphi$ as follows (where $0 \leqslant i \leqslant m$):

$(S_i, \ldots, S_m) \vDash p$	iff	$G \cup S_i^{\mathtt{true}} \vdash p \qquad (p \in \mathcal{P})$
$(S_i, \ldots, S_m) \vDash \neg\varphi$	iff	$(S_i, \ldots, S_m) \nvDash \varphi$
$(S_i, \ldots, S_m) \vDash \varphi_1 \wedge \varphi_2$	iff	$(S_i, \ldots, S_m) \vDash \varphi_1$ and $(S_i, \ldots, S_m) \vDash \varphi_2$
$(S_i, \ldots, S_m) \vDash \varphi_1 \vee \varphi_2$	iff	$(S_i, \ldots, S_m) \vDash \varphi_1$ or $(S_i, \ldots, S_m) \vDash \varphi_2$
$(S_i, \ldots, S_m) \vDash (\exists \vec{X} : D_{\bar{X}})\varphi[\vec{X}]$	iff	there is an $\vec{a} \in D_{\bar{X}}$ s.t. $(S_i, \ldots, S_m) \vDash \varphi[\vec{X}/\vec{a}]$
$(S_i, \ldots, S_m) \vDash (\exists_{l..u} \vec{X} : D_{\bar{X}})\varphi[\vec{X}]$	iff	there are $\geqslant l$ and $\leqslant u$ different $\vec{a} \in D_{\bar{X}}$ s.t. $(S_i, \ldots, S_m) \vDash \varphi[\vec{X}/\vec{a}]$
$(S_i, \ldots, S_m) \vDash \bigcirc\varphi$	iff	$i = m$ or $(S_{i+1}, \ldots, S_m) \vDash \varphi$

A crucial part here is $(S_i, \ldots, S_m) \vDash \bigcirc\varphi$ for $i = m$: in case we reach the end of a state sequence, every formula of the form $\bigcirc\varphi$ must be true. Together with the definition of an n-max sequence, this correctly grasps the intuition for

terminated sequences of length smaller than n, so that, for example, formula (5) on page 8 is clearly entailed in each terminal state. In general, $\bigcirc\varphi$ is considered true in every terminal state even if φ is inconsistent. In our setting this is perfectly acceptable as we are just interested in the truth of a formula in reachable states—all states beyond are irrelevant. It is worth mentioning that $\bigcirc\neg\varphi$ and $\neg\bigcirc\varphi$ are only equivalent for nonterminal states, whereas for every terminal state S_t we have that $(S_t) \models \bigcirc\neg\varphi$ but $(S_t) \not\models \neg\bigcirc\varphi$.

The following proposition relates sequences that are longer than the degree n of the formula to be proved to n-max sequences. This generalises formula entailment to a context with additional formulas that can have a higher degree. It is conditioned on the standard restriction to *playable* GDL games, meaning that every role has at least one legal move in every nonterminal reachable state [25].

Proposition 9. *Let G be a GDL specification, φ be a sequence invariant of degree n, $(S_0, \ldots, S_m)$ be an n-max sequence, and $\hat{n} \geqslant n$ arbitrary.*

1. *Let G be playable and state S_0 reachable. Then $(S_0, \ldots, S_m)$ can be extended to an $\hat{n}$-max sequence $(S_0, \ldots, S_m, \ldots, S_{\hat{m}})$.*
2. *For all $\hat{n}$-max sequences $(S_0, \ldots, S_m, \ldots, S_{\hat{m}})$ extended from $(S_0, \ldots, S_m)$:*

$$(S_0, \ldots, S_m) \models \varphi \quad \textit{iff} \quad (S_0, \ldots, S_m, \ldots, S_{\hat{m}}) \models \varphi.$$

Proof.

1. By induction on $\hat{n}$. The base case $n = \hat{n}$ is immediate. For the induction step, assume that $(S_0, \ldots, S_m)$ can be extended to an $\hat{n}$-max sequence $(S_0, \ldots, S_m, \ldots, S_{\hat{m}})$. If $S_{\hat{m}}$ is terminal, then $(S_0, \ldots, S_{\hat{m}})$ is also $\hat{n}+1$-max. Otherwise, since $S_{\hat{m}}$ is reachable and G playable, there are $A_{\hat{m}}$ and $S_{\hat{m}+1}$ such that $S_{\hat{m}} \xrightarrow{A_{\hat{m}}} S_{\hat{m}+1}$. Then $(S_0, \ldots, S_m, \ldots, S_{\hat{m}}, S_{\hat{m}+1})$ is $\hat{n}+1$-max.
2. By induction on the structure of φ. For the base case, consider $\varphi = p$ for some ground atom $p \in \mathcal{P}$. Entailment for a ground atom only involves the first state of a sequence, which implies the claim. For the induction step, consider $\varphi = \bigcirc\psi$ and let $(S_0, \ldots, S_m, \ldots, S_{\hat{m}})$ be an arbitrary $\hat{n}$-max sequence extended from $(S_0, \ldots, S_m)$. If S_0 is terminal, then $m = \hat{m} = 0$, hence the two sequences are identic. Otherwise, S_1 exists and we have $(S_0, \ldots, S_m) \models \bigcirc\psi$ iff $(S_1, \ldots, S_m) \models \psi$ iff (by the induction hypothesis) $(S_1, \ldots, S_m, \ldots, S_{\hat{m}}) \models \psi$ iff $(S_0, \ldots, S_m, \ldots, S_{\hat{m}}) \models \bigcirc\psi$. The remaining cases can be argued similarly. □

Since a playable GDL specification provides legal moves for every role only in states that are both nonterminal and reachable, the first item of Proposition 9 requires the assumption of S_0 being reachable. As an example, reconsider the GDL specification of Quarto depicted in Fig. 1 (page 5). Although this game is playable, there are (unreachable) states S which are nonterminal and nonplayable, e.g. if $pctrl(r1) \in S$ and $selected(p) \notin S$ for all pieces p. Then player *r1* has no legal move in S, and the 0-max sequence (S) cannot be extended to a 1-max sequence. This has the following consequence: even if an n-max formula φ is known to be true with respect to all $\hat{n}$-max sequences starting at S for some $\hat{n} \geqslant n$, φ is not necessarily true with respect to all n-max sequences starting at S, unless S is a *reachable* state. This explains the restriction to identical initial subsequences in the equivalence result in the second item of Proposition 9.

4. Verification of sequence invariants

While in theory state sequence invariants can be verified by a complete search through the set of reachable states (provided the game is finite, of course), as investigated in [32], our interest lies in finding a practical proof method that can be applied to games with far too large a state space to permit complete search. In the following, we will present such a method in three steps. First, we define the so-called temporal extension of a set of GDL clauses that allows us to compute a fixed number of state transitions within a single program (Section 4.1). Thereafter we show how this program can be extended by clauses that encode a given state sequence invariant (Section 4.2). Finally, we demonstrate how the combined program can be used to verify the encoded invariant against the game description (Section 4.3).

4.1. Temporal GDL extension

The game description language GDL is based on an elementary time structure that consists of only two time points, "before" (encoded by **true**) and "after" (encoded by **next**). Without further additions, a game description can thus be used only for reasoning about a single state transition: given a complete, finite state and a joint move, standard entailment allows to determine a successor state according to Definition 5 (page 9). This suffices to verify sequence invariants with degree 0, but invariants of higher degree require multiple successive state transitions and hence necessitate the introduction of additional time points in the rules.

Definition 10. For a valid GDL specification G, we call $G_{\leqslant n} \stackrel{\text{def}}{=} \bigcup_{0 \leqslant i \leqslant n} G_i$ the *temporal extension of G of degree n*, where each G_i is constructed by

- omitting all clauses from G with head **init**;
- replacing each occurrence of
  - **next**(f) by **true**$(f, i+1)$, and
  - $p(t_1, \ldots, t_n)$ by $p(t_1, \ldots, t_n, i)$, for each predicate $p \neq$ **next**.

Furthermore, the *timed variants* of the sets of unit clauses $S^{\texttt{true}}$ and $A^{\texttt{does}}$, defined as (6) on page 9, are

$$S^{\texttt{true}}(i) \stackrel{\text{def}}{=} \{\textbf{true}(f_1, i)\,.\,, \ldots, \textbf{true}(f_n, i)\,.\,\},$$
$$A^{\texttt{does}}(i) \stackrel{\text{def}}{=} \{\textbf{does}(r_1, A(r_1), i)\,.\,, \ldots, \textbf{does}(r_k, A(r_k), i)\,.\,\}$$

for any $S = \{f_1, \ldots, f_n\} \subseteq \Sigma$; $A : \{r_1, \ldots, r_k\} \mapsto \Sigma$; and $i \geqslant 0$.

As an example, let G be the game depicted in Fig. 1 (page 5) and consider the clause in lines 8–9. The temporal extension $G_{\leqslant n}$ contains the following clause for each $0 \leqslant i \leqslant n$:

```
true(pool(P),i+1) :- true(pool(P),i),
                     not does(r1,select(P),i),
                     not does(r2,select(P),i).
```

It is easy to see that the resulting program can be made more efficient by omitting the time argument in any predicate that is neither a GDL keyword nor depends on **true** or **does** in the original game description. Note also that, strictly speaking, $G_{\leqslant n}$ may not be stratified even if G is so; as a simple example consider the stratified clause `next(f) :- not true(g).`, whose temporal extension is `true(f,i+1) :- not true(g,i).` However, the temporally extended program could be easily rewritten into an equivalent but stratified program: instead of simply adding a time argument to predicates p, time could be encoded into their names, obtaining different predicate names p_i for each time step. Definition 10 is more readable, and we will nonetheless tacitly assume that $G_{\leqslant n}$ is always stratified.

Before we proceed, we recall a Splitting Theorem for answer sets, which can be found, e.g., in [8] and will be applied repeatedly in the remainder of the paper.

Theorem 11. *Let P and Q be two programs such that atoms in P are not unifiable with heads from clauses in Q. Then $\mathcal{A}_{P\cup Q}$ is an answer set for $P \cup Q$ iff the set $\mathcal{A}_P$, obtained from $\mathcal{A}_{P\cup Q}$ by restricting to atoms from P, is an answer set for P and $\mathcal{A}_{P\cup Q}$ is an answer set for $(\bigcup_{p\in\mathcal{A}_P}\{p\,.\,\}) \cup Q$.*

The following result shows that a temporally extended GDL specification can now be used to reason about sequences of state transitions with a fixed horizon. The proof as well as the proofs for subsequent results can be found in Appendix A.

Theorem 12. *Let G be a valid GDL specification and $S_0 \xrightarrow{A_0} S_1 \ldots \xrightarrow{A_{m-1}} S_m$ a sequence. Consider the program $P = S_0^{\texttt{true}}(0) \cup G_{\leqslant m} \cup \bigcup_{i=0}^{m-1} A_i^{\texttt{does}}(i)$, then for all $0 \leqslant i \leqslant m$ and predicate symbols $p \notin \{$**init**, **next**$\}$ that do not depend on* **does***, we have*

$$G \cup S_i^{\texttt{true}} \vdash p(\vec{t}) \quad \textit{iff} \quad P \vdash p(\vec{t}, i).$$

The temporal GDL extension of degree $m-1$ already incorporates clauses with head **true**(f, m) and is hence sufficient for reasoning about atoms $p(\vec{t})$ of the form $\mathit{true}(f)$ up to depth m. However, different atoms $p(\vec{t})$ require an extension up to degree m to include all relevant temporalised GDL rules with head $p(\vec{t}, m)$, which motivates the occurrence of both m and $m-1$ in program P of the theorem. Since predicate **legal** never depends on **does** in a valid GDL specification and all moves in a sequence are legal, our theorem implies $P \vdash$ **legal**$(r, A_i(r), i)$ for all $r \in R$ and $0 \leqslant i \leqslant m-1$, for any given sequence $S_0 \xrightarrow{A_0} S_1 \ldots \xrightarrow{A_{m-1}} S_m$. Similarly, $P \vdash$ **terminal**(m) holds if and only if S_m is a terminal state. The last step in the proof for this theorem (cf. Appendix A) requires **does**-independent predicates p, which was the rationale behind the corresponding restriction to sequence invariants made in Definition 3 (page 7).

4.2. Encoding sequence invariants

Next we show how game-specific knowledge in form of sequence invariants can be encoded as logic program clauses which, together with the temporal extension $G_{\leqslant n}$ of a valid GDL specification G, allows their formal verification against arbitrary n-max sequences. We first define the requirements for a suitable encoding and then provide an instance which satisfies these requirements.

To encode a formula to an answer set program, we need a previously unused atom with arity 0. Since encodings for several formulas will occur in the same answer set program at a later point, these atoms are required to be unequal for syntactically different formulas. To express this, we assume a unary injective function η. For example, syntactically different formulas φ and ψ can be encoded such that $\eta(\varphi) = \texttt{phi}$ and $\eta(\psi) = \texttt{psi}$ (assuming that `phi` and `psi` do not occur

elsewhere). However, in our example encoding, even syntactically identical formulas at different time levels will require different names, e.g. the two occurrences of sub-formula φ in formula $\varphi \supset \bigcirc\varphi$. Hence, with slight abuse of notation, we also use η to denote a binary injective function with an additional time level argument.[5] For example, sub-formulas φ at different time levels can be encoded using atoms $\eta(\varphi, 0) = \texttt{phi0}$ and $\eta(\varphi, 1) = \texttt{phi1}$. Similarly to η, we use two versions of an injective function *Enc* to denote the encoding of a formula φ (by $Enc(\varphi)$), possibly with respect to a time level i (by $Enc(\varphi, i)$).

The following definition gives a formal classification of a formula encoding. It is based on single sequences, and requires that a formula φ is true with respect to a sequence if and only if the temporal GDL extension, together with an encoding of that sequence and an encoding of φ, yields a unique answer set which entails the unique atom $\eta(\varphi)$ corresponding to φ. Since additional encodings of formulas with possibly higher degree may occur in the same answer set program, the correspondence needs to respect a possibly higher degree of the temporalised GDL clauses and sequences.

Definition 13. Let $\eta(\varphi)$ be a 0-ary atom which represents a unique name for sequence invariant φ with degree n. An *encoding of* φ, denoted $Enc(\varphi)$, is a set of clauses whose heads do not occur elsewhere and such that, for each $\hat{n} \geqslant n$ and $\hat{n}$-max sequence $S_0 \xrightarrow{A_0} S_1 \ldots \xrightarrow{A_{\hat{m}-1}} S_{\hat{m}}$ $(\hat{m} \leqslant \hat{n})$ of a valid GDL specification G, the program $P = S_0^{\texttt{true}}(0) \cup G_{\leqslant \hat{n}} \cup \bigcup_{i=0}^{\hat{m}-1} A_i^{\texttt{does}}(i) \cup Enc(\varphi)$ fulfils the following:

- P has exactly one answer set;
- $(S_0, \ldots, S_{\hat{m}}) \vDash \varphi$ iff $P \vdash \eta(\varphi)$.

Note that Theorem 12 is also applicable to program P from Definition 13, as P only adds the clauses $\bigcup_{m<i\leqslant\hat{n}} G_i \cup \bigcup_{i=m}^{\hat{m}-1} A_i^{\texttt{does}}(i) \cup Enc(\varphi)$, the heads of which do not occur in the logic program used in the theorem.

The encoding we are about to present makes use of a common addition to answer set programs known as *weight atoms* [27] of the form

$$l\{p_1, \ldots, p_k\}u$$

for $k, l, u \in \mathbb{N}$ and $0 \leqslant l \leqslant u$. A set $\mathcal{A}$ satisfies a weight atom iff at least l and at most u different members of $\{p_1, \ldots, p_k\}$ are in $\mathcal{A}$. Both l and u can be omitted, in which case there is no lower (respective, upper) bound.[6] We furthermore introduce *constraints*

$$:\!-b_1, \ldots, b_n\,.$$

as abbreviation for $1\{\}:\!-b_1, \ldots, b_n\,.$ Note that the answer sets of $G \cup \{:\!-b_1, \ldots, b_n\,.\}$ are exactly the answer sets of G *except* for those that do satisfy all of $b_1, \ldots, b_n$. If weight atom $l\{p_1, \ldots, p_k\}u$ occurs in the head of a clause, each $p_i \in \{p_1, \ldots, p_k\}$ is considered a clause head. Theorem 11 is known to also hold in this extended setting [8].

4.2.1. A sample encoding

Table 2 provides a recursive definition of how a sequence invariant can be encoded as a logic program: First, every predicate $p(\vec{t})$ at level i in the invariant is translated to a clause which entails $\eta(p(\vec{t}), i)$ (a unique, 0-ary naming atom for $p(\vec{t})$ at time point i) in case $p(\vec{t}, i)$ holds (case 1). Formulas with connectives different from "$\bigcirc$" recursively resolve to their correspondent sub-formulas (cases 2–6). Finally, $Enc(\bigcirc\psi, i)$ is constructed so as to entail $\eta(\bigcirc\psi, i)$ in case level i is terminal or sub-formula ψ is true at level $i+1$ (case 7).

As an example, recall from page 8 the sequence invariant (5) for Quarto. Rewriting "$\supset$" as a disjunction and applying standard transformations, we obtain the following equivalent formula:

$$\big(\exists R : \{r1, r2\}\big)\, true\big(pctrl(R)\big) \vee \bigcirc\big(\exists R : \{r1, r2\}\big)\, true\big(pctrl(R)\big).$$

Applying the recursive definition in Table 2 yields the following encoding, where atom `phi0` is the unique name $\eta(\varphi, 0)$ for the overall formula.

[5] With the same intent of denoting unique names, we will further use η with different arguments at a later point.

[6] The semantics of this addition can be given by extending Definition 4 on page 8 with these subsequent reductions:

- Delete all clauses with a weight atom in the body such that answer set candidate $\mathcal{A}$ does not satisfy its upper bound.
- Delete all upper bounds from body weight atoms in the remaining clauses.
- Replace each remaining clause of the form $l\{p_1, \ldots, p_k\}u :\!-b_1, \ldots, b_n\,.$, by a set of clauses $p :\!-b_1, \ldots, b_n\,.$, for each $p \in \{p_1, \ldots, p_k\} \cap \mathcal{A}$.

The reduced set of clauses $G^{\mathcal{A}}$ admits a unique minimal model, and $\mathcal{A}$ is an answer set for G if and only if it coincides with this unique minimal model and, additionally, satisfies G.

Table 2
Encoding an arbitrary sequence invariant as a logic program.

1. $Enc(p(\vec{t}), i)$	$= \{\eta(p(\vec{t}), i) \texttt{ :- } p(\vec{t}, i)\texttt{ .}\}$
2. $Enc(\neg\psi, i)$	$= \{\eta(\neg\psi, i) \texttt{ :- } \textbf{not}\,\eta(\psi, i)\texttt{ .}\} \cup Enc(\psi, i)$
3. $Enc(\psi_1 \wedge \psi_2, i)$	$= \{\eta(\psi_1 \wedge \psi_2, i) \texttt{ :- } \eta(\psi_1, i), \eta(\psi_2, i)\texttt{ .}\} \cup Enc(\psi_1, i) \cup Enc(\psi_2, i)$
4. $Enc(\psi_1 \vee \psi_2, i)$	$= \{\eta(\psi_1 \vee \psi_2, i) \texttt{ :- } \eta(\psi_1, i)\texttt{ .}, \eta(\psi_1 \vee \psi_2, i) \texttt{ :- } \eta(\psi_2, i)\texttt{ .}\} \cup Enc(\psi_1, i) \cup Enc(\psi_2, i)$
5. $Enc((\exists \vec{X} : D_{\vec{X}})\psi[\vec{X}], i)$	$= \bigcup_{\vec{a}\in D_{\vec{X}}} \{\eta((\exists \vec{X} : D_{\vec{X}})\psi[\vec{X}], i) \texttt{ :- } \eta(\psi[\vec{X}/\vec{a}], i)\texttt{ .}\} \cup \bigcup_{\vec{a}\in D_{\vec{X}}} Enc(\psi[\vec{X}/\vec{a}], i)$
6. $Enc((\exists_{l..u} \vec{X} : D_{\vec{X}})\psi[\vec{X}], i)$	$= \{\eta((\exists_{l..u} \vec{X} : D_{\vec{X}})\psi[\vec{X}], i) \texttt{ :- } l\{\eta(\psi[\vec{X}/\vec{a}], i) : \vec{a} \in D_{\vec{X}}\}u\texttt{ .}\} \cup \bigcup_{\vec{a}\in D_{\vec{X}}} Enc(\psi[\vec{X}/\vec{a}], i)$
7. $Enc(\bigcirc\psi, i)$	$= \{\eta(\bigcirc\psi, i) \texttt{ :- } \textbf{terminal}(i)\texttt{ .}, \eta(\bigcirc\psi, i) \texttt{ :- } \eta(\psi, i+1)\texttt{ .}\} \cup Enc(\psi, i+1)$

```
phi0 :- ex0.              ex0 :- a0.   a0 :- true(pctrl(r1),0).
phi0 :- nxt_ex1.          ex0 :- b0.   b0 :- true(pctrl(r2),0).

nxt_ex1 :- terminal(0).   ex1 :- a1.   a1 :- true(pctrl(r1),1).
nxt_ex1 :- ex1.           ex1 :- b1.   b1 :- true(pctrl(r2),1).
```
(7)

It is easy to verify that the size of the encoding of a given formula is always linear in the size of the original formula. Together with the underlying temporally extended GDL specification the given encoding is correct w.r.t. the definition of formula entailment, as the following result shows.

Theorem 14. *Let G be a valid GDL specification and φ be a sequence invariant. Then $Enc(\varphi) := Enc(\varphi, 0)$ with the unique name atom $\eta(\varphi) := \eta(\varphi, 0)$ for φ (cf. Table 2) is an encoding of φ.*

In order to keep our framework general, in the following we abstract from our specific encoding and consider any $Enc(\varphi)$ that satisfies the requirements of Definition 13.

4.3. Proving sequence invariants

We proceed by showing how a temporally extended GDL description along with an encoding of a formula can be used to automate an induction proof for the validity of the formula. To prove that a state sequence invariant φ holds in each reachable state S (i.e., $S \vDash \varphi$), we will construct two answer set programs dependent on φ: a base case to show that φ is entailed in the initial state, and an induction step to show that, provided a state entails φ, each legal successor state will also entail φ. Together this implies that φ holds in all reachable states.

As we have seen in Section 3.2, fluents (i.e., state features) can grow indefinitely, hence the set which contains all ground fluents that occur in a reachable state (henceforth denoted by *FDom*) may be infinite. Consequently, also the set of all actions of a player r (henceforth denoted by $ADom(r)$) is potentially infinite, e.g. due to a clause like

```
legal(r,a(X)) :- true(X).
```

which defines a legal action for every fluent. In order to develop a decidable proof method for sequence invariants, we have to restrict our attention to GDL specifications that are *finite* in the sense that the associated set *FDom* is finite. By the recursion restriction in Definition 2 (page 6) this suffices to guarantee that $ADom(r)$ is finite as well.[7]

4.3.1. Base case

Action generator. Based on the sets $ADom(r)$ of possible actions for player r we can define a logic program to encode the fundamental requirement that each player has to perform a legal move in each nonterminal state up to time step n. Let $P^{legal}_{\leqslant n}$ consist of the following clauses P^{legal}_{i} for each $0 \leqslant i \leqslant n$ and $r \in R$.[8]

[7] A detailed discussion on how to reliably compute both *FDom* and *ADom* will follow in Section 6.1.

[8] We tacitly assume that predicate `terminated` does not occur elsewhere.

$$
\begin{array}{ll}
(c_1) & \texttt{terminated}(i) \,\texttt{:-}\, \textbf{terminal}(i)\,. \\
(c_2) & \texttt{terminated}(i) \,\texttt{:-}\, \texttt{terminated}(i-1)\,. \quad (\text{for } i > 0 \text{ only}) \\
(c_3) & 1\{\textbf{does}(r, a, i) : a \in \mathit{ADom}(r)\}1 \,\texttt{:-}\, \textbf{not}\ \texttt{terminated}(i)\,. \\
(c_4) & \texttt{:-}\, \textbf{does}(r, \texttt{A}, i), \textbf{not legal}(r, \texttt{A}, i)\,.
\end{array} \tag{8}
$$

Subsequently, $P^{\mathit{legal}}_{\leqslant n}$ will also be called an *action generator*.

Base case program. For a game description G and a formula φ over G with degree n, the answer set program for the *base case* is defined as follows:

$$P^{bc}_{\varphi}(G) = S^{\texttt{true}}_{\mathit{init}}(0) \cup G_{\leqslant n} \cup P^{\mathit{legal}}_{\leqslant n-1} \cup \mathit{Enc}(\varphi) \cup \big\{ \texttt{:-}\, \eta(\varphi)\,. \big\}.$$

Put in words, $P^{bc}_{\varphi}(G)$ consists of an encoding for the initial state, $S^{\texttt{true}}_{\mathit{init}}(0)$; a temporal GDL specification up to time step n, $G_{\leqslant n}$; the necessary requirements concerning legal moves, $P^{\mathit{legal}}_{\leqslant n-1}$; an encoding for the formula φ, $\mathit{Enc}(\varphi)$; and the statement that φ should not be entailed in any model of $P^{bc}_{\varphi}(G)$, $\{\texttt{:-}\, \eta(\varphi)\,.\}$. In case $P^{bc}_{\varphi}(G)$ has no answer set, the last clause implies that there is no state sequence starting at S_{init} that makes φ false—which means that φ is entailed by S_{init}.

4.3.2. Induction step

State generator. For the induction step answer set program, the state encoding $S^{\texttt{true}}_{\mathit{init}}(0)$ needs to be substituted by a general "state generator" program, whose answer sets produce the reachable states of a GDL game. In general, the computation of the reachable states requires a full game tree traversal which is not feasible in interesting games (e.g., the game tree of chess is estimated with about 10^{45} states). This motivates the use of an easy approximation that may comprise unreachable states as well. The simplest such approximation is the program

$$0 \quad \big\{\textbf{true}(f, 0) : f \in \mathit{FDom}\big\}.$$

which generates *all* combinations of fluents, whether reachable or not. In general, a state generator is defined as follows.

Definition 15. A *state generator* for a valid GDL specification G is an answer set program P^{gen} such that

- The only atoms in P^{gen} are of the form $\textbf{true}(f, 0)$, where $f \in \Sigma$, or auxiliary atoms that do not occur elsewhere; and
- for every reachable state S of G, P^{gen} has an answer set $\mathcal{A}$ such that for all $f \in \Sigma$: $\textbf{true}(f, 0) \in \mathcal{A}$ iff $f \in S$.

The practical necessity for using a superset of the reachable states in the induction step has interesting consequences, which are best seen with an example. Suppose we want to prove the Quarto sequence invariant (4) from page 8, that is, $\varphi = (\forall X, Y : I)(\exists_{0..1} P : D_P)\ \mathit{true}(\mathit{cell}(X, Y, P))$. The (unreachable!) state

$$S = \big\{\mathit{cell}(1, 1, b), \mathit{selected}(p0000), \mathit{selected}(p0001), \mathit{pctrl}(r1), \mathit{pctrl}(r2)\big\}$$

satisfies φ. In S, players *r1* and *r2* both have the legal move of placing a selected piece at cell $(1, 1)$. Consider, then, the case where they choose to place different pieces. This results in the successor state

$$\big\{\mathit{cell}(1, 1, p0000), \mathit{cell}(1, 1, p0001), \mathit{sctrl}(r1), \mathit{sctrl}(r2)\big\}$$

which violates φ. Hence, there is an undesired counterexample for the induction step as long as S is considered potentially reachable. However, knowing that sequence invariant

$$\big(\exists_{1..1} C : \big\{\mathit{sctrl}(r1), \mathit{sctrl}(r2), \mathit{pctrl}(r1), \mathit{pctrl}(r2)\big\}\big)\ \mathit{true}(C)$$

holds in all reachable states of Quarto allows to reject S and all similar states that contain more than one "control" fluent. As a consequence, none of the direct successors of the remaining φ-satisfying states violates φ—which establishes a successful proof of the induction step. This shows that the addition of all previously proved sequence invariants to a state generator can positively influence the outcome of a subsequent proof attempt. The following construction of the answer set program for the induction step of a proof accounts for this issue by the inclusion of formulas which are already known to be valid.

Induction step program. For a game description G, an arbitrary state generator P^{gen} over G, a set Ψ of valid sequence invariants that have at most degree n_Ψ, a formula φ with degree n_φ, and $\hat{n} = \max(n_\Psi, n_\varphi + 1)$, the *induction step* answer set program is

$$\begin{aligned} P^{\mathit{is}}_{\varphi,\Psi}(G) \;=\; & P^{\mathit{gen}} \cup G_{\leqslant \hat{n}} \cup P^{\mathit{legal}}_{\leqslant \hat{n}-1} \cup \mathit{Enc}(\varphi \supset \bigcirc\varphi) \cup \big\{ \texttt{:-}\, \eta(\varphi \supset \bigcirc\varphi)\,. \big\} \cup \\ & \bigcup_{\psi \in \Psi} \big(\mathit{Enc}(\psi) \cup \big\{ \texttt{:-}\, \textbf{not}\ \eta(\psi)\,. \big\}\big). \end{aligned}$$

Put in words, $P^{is}_{\varphi,\Psi}(G)$ differs from $P^{bc}_{\varphi}(G)$ in the following way. First, an arbitrary state generator P^{gen} is used instead of the initial-state encoding. Second, the time level has increased from n to $\hat{n}$. Third, formulas are now encoded thus: $\bigcup_{\psi\in\Psi}(Enc(\psi) \cup \{\texttt{:-}\,\textbf{not}\ \eta(\psi)\,\texttt{.}\})$, which ensures that $P^{is}_{\varphi,\Psi}(G)$ has only answer sets that, for all formulas $\psi \in \Psi$, contain $\eta(\psi)$ and hence represent $\hat{n}$-max sequences which satisfy ψ. These sequences still include all reachable $\hat{n}$-max sequences and are, by the clauses $Enc(\varphi \supset \bigcirc\varphi) \cup \{\texttt{:-}\,\eta(\varphi \supset \bigcirc\varphi)\,\texttt{.}\}$, further restricted to those which satisfy $\neg(\varphi \supset \bigcirc\varphi)$. In case $P^{is}_{\varphi,\Psi}(G)$ is inconsistent, there is no reachable $\hat{n}$-max sequence which satisfies $\neg(\varphi \supset \bigcirc\varphi)$—which implies that φ is satisfied in all direct successors of reachable states that themselves satisfy φ.

4.3.3. Example

Recall the example encoding (7) (page 14) for the Quarto state invariant

$$\varphi = \neg\big(\exists R : \{r1, r2\}\big)\ true\big(pctrl(R)\big) \supset \bigcirc\big(\exists R : \{r1, r2\}\big)\ true\big(pctrl(R)\big).$$

This formula can now be proved to hold in all reachable states:

Base case. Let G be the clauses in Fig. 1 (page 5). Since the initial state contains $sctrl(r1)$, the temporal extension of clause 15 in Fig. 1 implies that the atom **true**(`pctrl(r2),1`) is contained in each answer set of $P^{bc}_{\varphi}(G)$. Consequently, the example encoding for φ implies that also $\eta(\varphi, 0) = \texttt{phi0}$ is contained. This however contradicts the constraint `:-phi0.` in $P^{bc}_{\varphi}(G)$, so $P^{bc}_{\varphi}(G)$ has no answer set and, thus, φ holds in the initial state of the game.

Induction step. The induction step program $P^{is}_{\varphi,\Psi}(G)$ contains $Enc(\varphi \supset \bigcirc\varphi, 0) \cup \{\texttt{:-}\,\eta(\varphi \supset \bigcirc\varphi)\,\texttt{.}\}$, where $Enc(\varphi \supset \bigcirc\varphi, 0)$ can be specified such as to contain

- the encoding $Enc(\varphi, 0)$ for φ as given in (7), where $\eta(\varphi, 0) = \texttt{phi0}$;
- an additional set $Enc(\varphi, 1)$ that differs from $Enc(\varphi, 0)$ only in the used name atoms and time points increased by 1, where we specify $\eta(\varphi, 1) = \texttt{phi1}$; and
- the following additional clauses, where $\eta(\varphi \supset \bigcirc\varphi) = \texttt{phi\_imp\_nxt\_phi}$:

```
phi_imp_nxt_phi :- neg_phi.   neg_phi :- not phi0.
phi_imp_nxt_phi :- nxt_phi.

nxt_phi :- terminal(0).
nxt_phi :- phi1.
```

The specified encoding together with the constraint `:- phi_imp_nxt_phi.` implies that each of the answer sets $\mathcal{A}$ of $P^{is}_{\varphi,\Psi}(G)$ satisfies the following three conditions:

1. `phi0` $\in \mathcal{A}$,
2. **terminal**$(0) \notin \mathcal{A}$, and
3. `phi1` $\notin \mathcal{A}$.

Now let r range over $\{r1, r2\}$. The body of one clause with head `phi0` of $Enc(\varphi, 0) \subseteq Enc(\varphi \supset \bigcirc\varphi, 0)$ must be true in $\mathcal{A}$ due to the first condition and cannot be satisfied by **terminal**(0) due to the second condition. Hence, for some r either **true**$(\texttt{pctrl}(r), 0) \in \mathcal{A}$ or **true**$(\texttt{pctrl}(r), 1) \in \mathcal{A}$. Furthermore, since $Enc(\varphi, 1) \subseteq Enc(\varphi \supset \bigcirc\varphi, 0)$, the third condition implies that **true**$(\texttt{pctrl}(r), 1) \notin \mathcal{A}$ and **true**$(\texttt{pctrl}(r), 2) \notin \mathcal{A}$ for each r (and that **terminal**$(1) \notin \mathcal{A}$).

Hence, there must be some r such that **true**$(\texttt{pctrl}(r), 0) \in \mathcal{A}$. By the temporal extension of clauses 14 and 15 in Fig. 1, this results in existence of an r such that **true**$(\texttt{pctrl}(r), 2) \in \mathcal{A}$, in contradiction to the previously mentioned implications of the third condition. Thus, $P^{is}_{\varphi,\Psi}(G)$ has no answer set, which implies that φ is satisfied in all direct successors of reachable states that themselves satisfy φ.

4.4. Soundness of the verification method

The following result is a prerequisite for the soundness proof of the verification method introduced in the previous section. It provides a one-to-one relation between answer set programs encoding a particular state sequence and those including an action generator.

Theorem 16. *Let G be a valid GDL specification and $\mathcal{A}$ be a subset of the ground atoms over G together with* $\{\texttt{terminated}(i) : i \in \mathbb{N}\}$. *The following two statements are equivalent*:

(1) *$\mathcal{A}$ is an answer set for*

$$P = S_0^{\texttt{true}}(0) \cup G_{\leqslant n} \cup P^{legal}_{\leqslant n-1}.$$

(2) *There is an n-max sequence* $Seq = (S_0 \xrightarrow{A_0} S_1 \dots \xrightarrow{A_{m-1}} S_m)$ *such that* $\mathcal{A}$ *is the unique answer set for*

$$P^{Seq} = S_0^{\texttt{true}}(0) \cup G_{\leqslant n} \cup P^{c_1,c_2}_{\leqslant n-1} \cup \bigcup_{i=0}^{m-1} A_i^{\texttt{does}}(i),$$

where $P^{c_1,c_2}_{\leqslant n-1} = \bigcup_{i=0}^{n-1} P_i^{c_1,c_2}$ *and* $P_i^{c_1,c_2}$ *denotes all clauses of the shape* (c_1) *and* (c_2) *in the part* P_i^{legal} *of the action generator, defined as* (8) *on page* 15.

The following theorem states our main result, the soundness of the verification method.

Theorem 17. *Let G be a playable and valid GDL specification whose initial state is* S_{init}*. Let* Ψ *be a set of sequence invariants over G which are satisfied in all reachable states, and let* φ *be a sequence invariant. If* $P^{bc}_{\varphi}(G)$ *and* $P^{is}_{\varphi,\Psi}(G)$ *are inconsistent, then for all finite sequences* $(S_{init}, S_1, \dots, S_k)$ *we have* $S_k \vDash \varphi$*.*

Note that the playability assumption of the GDL specification in Theorem 17 can be omitted in case $n_\Psi \leqslant \deg(\varphi) + 1$ for the maximal degree n_Ψ of formulas in Ψ, since then the induction step proof does not require an extension of sequence Seq_{n+1} according to Proposition 9 (page 11).

5. Improvements

5.1. Proving multiple properties at once

Requiring a general game player to evoke an ASP system individually for each formula in a large set of candidate properties is not feasible for the practice of general game playing with a limited amount of time to analyse the rules of a hitherto unknown game. In the following we therefore develop a crucial extension of our method that enables a general game player to evoke the ASP system only once in order to determine precisely which of a whole set Φ of formulas is valid w.r.t. a given game description. We will show that for this purpose it suffices to construct only two answer set programs for Φ, one to establish all base case proofs and one for all induction steps. For any $\varphi \in \Phi$, then, if all answer sets for the base case program satisfy φ we know that φ is entailed in the initial state. If additionally all answer sets of the induction step program satisfy $\varphi \supset \bigcirc\varphi$, we can conclude that φ is entailed in all reachable states. In practice, this results in a significantly more efficient proof method, especially when grouping structurally similar formulas which, for example, have the same degree or incorporate different instances of the same atoms. In Section 6.3 we will further motivate this intuition by an experiment setup that allows to prove various properties efficiently.

For a game description G and a finite set of state sequence invariants Φ with maximal degree $\hat{n}_{bc}$, the generalised base case answer set program is defined as follows:

$$P^{bc}_{\Phi}(G) = S^{\texttt{true}}_{init}(0) \cup G_{\hat{n}_{bc}} \cup P^{legal}_{\hat{n}_{bc}-1} \cup \bigcup_{\varphi\in\Phi} Enc(\varphi).$$

Compared to $P^{bc}_{\varphi}(G)$, the constraint $\{:\!-\eta(\varphi)\,.\}$ is no longer used, which results in a unique answer set for *each* of the $\hat{n}_{bc}$-max sequences starting in S_{init} (as opposed to distinct answer sets for sequences that violate φ in $P^{bc}_{\varphi}(G)$ only). This is necessary to keep all relevant answer sets for formulas from Φ different from φ which do not satisfy $:\!-\eta(\varphi)$. Moreover, encodings are added for all the formulas in Φ, consequently raising the overall degree of the generated answer set program to the maximal formula degree $\hat{n}_{bc}$.

Now let $\hat{n}_{is}$ be the maximal degree of formulas in $\Phi \cup \Psi \cup \{\varphi \supset \bigcirc\varphi\}$. Applying similar changes to $P^{is}_{\varphi,\Psi}(G)$, we define the generalised induction step answer set program as follows:

$$\begin{aligned} P^{is}_{\Phi,\Psi}(G) \;=\; & P^{gen} \cup G_{\hat{n}_{is}} \cup P^{legal}_{\hat{n}_{is}-1} \cup \bigcup_{\varphi\in\Phi} Enc(\varphi \supset \bigcirc\varphi) \cup \\ & \bigcup_{\psi\in\Psi} \big(Enc(\psi) \cup \{:\!-\,\textbf{not}\,\eta(\psi)\,.\}\big). \end{aligned}$$

The generalised method can be proved sound, too.

Theorem 18. *Let G be a playable and valid GDL specification whose initial state is* S_{init}*. Moreover, let* Φ *and* Ψ *be sets of sequence invariants over G such that each* $\psi \in \Psi$ *is satisfied in all reachable states, and let* $\varphi \in \Phi$*. If every answer set for* $P^{bc}_{\Phi}(G)$ *contains* $\eta(\varphi)$ *and every answer set for* $P^{is}_{\Phi,\Psi}(G)$ *contains* $\eta(\varphi \supset \bigcirc\varphi)$*, then for all finite sequences* $(S_{init}, S_1, \dots, S_k)$ *we have* $S_k \vDash \varphi$*.*

The following theorem shows that the generalisation succeeds in proving at least all the state sequence invariants that can be proved with the original method.

Theorem 19. *Consider the same assumptions and naming conventions as in Theorem* 18.

(1) *If* $P^{bc}_{\varphi}(G)$ *is inconsistent then* $\eta(\varphi)$ *is in all answer sets of* $P^{bc}_{\Phi}(G)$.
(2) *If* $P^{is}_{\varphi,\Psi}(G)$ *is inconsistent then* $\eta(\varphi \supset \bigcirc\varphi)$ *is in all answer sets of* $P^{is}_{\Phi,\Psi}(G)$.

It should be stressed, however, that the converse of (2) in Theorem 19 does not hold: An answer set for $P^{is}_{\varphi,\Psi}(G)$ represents an established $\hat{n}$-max sequence *Seq* (cf. Theorem 16, page 16) that violates $\varphi \supset \bigcirc\varphi$. *Seq* however might not be extendable to an $\hat{n}_{is}$-max sequence (cf. the remark following Proposition 9 on page 11) that could serve as counterexample for $\varphi \supset \bigcirc\varphi$ in $P^{is}_{\Phi,\Psi}(G)$. Hence our efficiency improvement even strengthens the result, depending on the maximal degree $\hat{n}$ of the given formula set Φ. For the same reason, adding proved formulas as evidence can strengthen the results of both the original method and its generalisation.

5.2. Nonplayable sequences

Sequence invariant entailment $\vDash$ (cf. Definition 8, page 10), defined over n-max sequences for the degree n of the formula to be verified, does not account for sequences that are of length smaller than n and end in a nonterminal state that does not permit a move for one of the players (also called nonplayable sequences). This has an interesting effect, as the following simple, nonplayable game shows:

```
role(r).
init(f).
legal(r,a) :- true(f).
```

Consider the sequence invariant that axiomatises playability, that is,

$$\varphi = \neg terminal \supset (\forall R : D_R)\ (\exists M : D_M)\ legal(R, M)$$

with the domains $D_R = \{r\}$ and $D_M = \{a\}$ for the example game. Additionally, consider an arbitrary formula ψ with degree 1 that is known to be satisfied in the initial state, $S_{init} = \{f\}$. Then $\varphi \wedge \psi$ is satisfied in S_{init} since the only 1-max sequence $\{f\} \xrightarrow{\{(r,a)\}} \{\}$ satisfies $\varphi \wedge \psi$. Formula $\varphi \wedge \psi$ is also satisfied in state $\{\}$, as no 1-max sequence emerges from that state. Since these are the only reachable states, $\varphi \wedge \psi$ is considered true in each reachable state, contradicting our intuition that the game is nonplayable and hence that φ should be false. The only counterexample, however, would be the sequence $(\{\})$ of length 0, which is nonplayable with respect to length 1. But as nonplayable sequences are not among the 1-max sequences, $(\{\})$ will never be considered in our setting.

On the one hand, playability is a standard requirement for General Game Playing Competitions and thus can be presupposed by a general game-playing system. A GDL game designer, on the other hand, might be particularly interested in proving whether a game she has designed is indeed playable, which motivates the following considerations. To begin with, observe that playability of a game has no influence on the outcome of a proof attempt for sequence invariant φ when tried together with a set Ψ of previously proved formulas of degree less or equal to 1, since:

Base case amounts to verifying φ with respect to the only 0-max sequence starting in S_{init}, namely (S_{init}), which incorporates no state transition and hence is independent of the playability assumption; and
Induction step amounts to verifying φ with respect to every 1-max sequence which starts in a state satisfying φ, which is again independent of the playability assumption since φ represents playability itself.

Hence, the proof method can be used to reliably prove the playability formula φ, relying on previously proved formulas of degree $\leqslant 1$ only, in order to assume playability thereafter. If this proof attempt is not successful, the (indirect) playability assumption can be dropped by incorporating nonplayable sequences into the proof method as follows:

- Altering the definition of an n-max sequence (cf. Definition 8, page 10) such that in case of length smaller than n the last state of the sequence might also be nonterminal and not permit a legal move for one of the players.
- Adding the following clauses to the game description G (cf. Definition 2, page 6):

```
has_no_legal(R) :-not has_legal(R), role(R).
has_legal(R)    :-legal(R,A).
```

- Adding to $Enc(\bigcirc\psi, i)$ (cf. Section 4.2) for each $r \in R$:

$$\eta(\bigcirc\psi, i) \texttt{:-has\_no\_legal}(r, i).$$

- Adding to P^{legal}_{n-1} (cf. Section 4.3) for all $0 \leqslant i \leqslant n-1$ and $r \in R$:

$$\texttt{terminated}(i) \texttt{:-has\_no\_legal}(r, i).$$

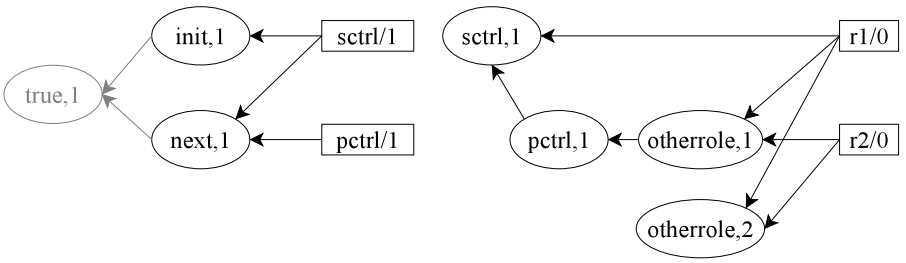

Fig. 2. An example domain graph for calculating domains of functions and predicates. Ellipses denote individual arguments of functions or predicates while squares denote constants or function symbols.

Besides increasing the complexity of the constructed answer set programs $P^{bc}_{\varphi}(G)$ and $P^{is}_{\varphi,\Psi}(G)$, this modification weakens the proof method. As an example, suppose we extend the (nonplayable) game from the beginning of this section by the following clauses:

```
next(f) :- true(f), not true(g).
next(g) :- true(g).
```

Note that the extended game is playable, as opposed to the original one. Now consider the formula *true*(*f*), which holds in each reachable state. A proof attempt, however, yields the (unreachable) 1-max sequence $Seq = (\{f, g\}, \{g\})$ as a counterexample for the induction step, since $Seq \not\models \mathit{true}(f) \supset \bigcirc\, \mathit{true}(f)$. Assuming playability and the original setting, *Seq* is rejected as soon as some formula ψ with degree 2 is known to be satisfied in each reachable state and added to $P^{is}_{\varphi,\Psi}(G)$, because *Seq* cannot be extended to a 2-max sequence. This indeed allows to prove the induction step for *true*(*f*). This is in contrast to the modified setting, where *Seq* is also considered 2-max and might satisfy ψ as well. In conclusion, the presented modification reliably copes with games which are not known to be playable, but whenever this assumption can be made, the more efficient and stronger original proof method should be used instead.

6. Implementation

We have implemented our proof method using Fluxplayer [35] to generate the answer set program (ASP), in combination with the ASP solver Clingo (version 3.0.1) from the state-of-the-art answer set solving collection Potassco [10]. When proving multiple properties at once (cf. Section 5.1) we use the option "cautious reasoning" for Clingo to compute the intersection of all answer sets.

6.1. Domain calculation

For formulating sequence invariants as well as for the encoding of the action and state generators that are used in the proofs, we need information about the domains of predicates and functions in a game description. Specifically, we need the set of potential actions *ADom*(*r*) for each role *r* of a game to encode the action generator and the set of all ground fluents *FDom* to encode the state generator (cf. Section 4.3). These sets are not directly given by the game description but have to be inferred from it. The minimal set of all ground fluents is the union of all reachable states, while the minimal set of potential actions for a role *r* is the union of the legal actions of *r* in all reachable states. In general, computing the minimal sets *FDom* and *ADom*(*r*) requires to enumerate all reachable states. Since this is infeasible for any game of practical interest, we compute supersets of the domains with an algorithm that only depends on the size of the game description but not on the actual state space of the game.

Following the approach described in [35], we compute the supersets of the domains of all relations and functions of the game description by generating a so-called *domain graph* from the rules of the game description. Fig. 2 shows the domain graph for the following subset of the Quarto rules in Fig. 1:

```
init(sctrl(r1)).
next(sctrl(R))  :- true(pctrl(R)).
next(pctrl(R1)) :- true(sctrl(R2)), otherrole(R1,R2).

otherrole(r1,r2).
otherrole(r2,r1).
```

Formally, a domain graph is defined as follows.

Definition 20. Let G be a GDL specification. Let G' be G together with the following three rules:

```
true(F)     :- init(F).
true(F)     :- next(F).
does(R,M)   :- legal(R,M).
```

A domain graph for a GDL specification G is the smallest directed graph $D = (V, E)$ with vertices V and edges E such that:

- For every n-ary predicate or function p in G', $p/n \in V$ and $(p, i) \in V$ for all $i \in \{1, \ldots, n\}$.[9]
- If a function $f(x_1, \ldots, x_n)$ occurs as i-th argument of a predicate or function p in the head of a rule in G', then $f/n \to (p, i) \in E$.
- If a variable occurs as i-th argument of a predicate or function p in the head of a rule $r \in G'$ and as j-th argument of a predicate or function q in a positive literal in the body of r, then $(q, j) \to (p, i) \in E$.

Informally speaking, there is a node in the graph for every argument position of each function symbol and predicate symbol in the game description. For example, Fig. 2 contains the nodes (*otherrole*, *1*) and (*otherrole*, *2*) referring to the arguments of the binary predicate *otherrole*. Furthermore, there is a node for each constant and function symbol. For example, the nodes *r1*/0, *r2*/0 for the constants of the same name and the nodes *sctrl*/1, *pctrl*/1 for the unary functions *sctrl* and *pctrl*. There is an edge between an argument node and function symbol node if there is a head of a rule in the game description where the function symbol appears in the respective argument of a function or predicate. For example, there is an edge between the nodes *sctrl*/1 and (*init*, *1*) because of the rule **init** `(sctrl(r1))`. Furthermore, there is an edge between two argument position nodes if there is a rule in the game in which the same variable appears in both arguments, once in the head and once in the body of the rule. For example, because of shared variable `R` in the rule **next** `(sctrl(R))` `:-` **true** `(pctrl(R))`., there is the edge (*pctrl*, *1*) → (*sctrl*, *1*). The three additional rules capture the intuition that a fluent in the game is either true initially or in some successor state and that any legal move may potentially be executed by some player.

After constructing the domain graph $D = (V, E)$ from the game rules, we compute its transitive closure $D^+ = (V, E^+)$. The domains of all predicates and functions in the game description can now be defined as follows.

Definition 21. Let $D^+ = (V, E^+)$ be the transitive closure of the domain graph for a GDL specification G. Let $dom(p/n)$ denote the set of all ground instances of the n-ary predicate or function p and $dom(p, i)$ denote the domain of the i-th argument of predicate or function p:

- $dom(p, i) = \{c : c/0 \to (p, i) \in E^+\} \cup \bigcup_{q/n \to (p,i) \in E^+, n>0} dom(q/n)$,
- $dom(p/n) = \{p(x_1, \ldots, x_n) : x_1 \in dom(p, 1), \ldots, x_n \in dom(p, n)\}$.

Note that the definition above yields finite sets for all domains if the extended GDL specification G' from Definition 20 obeys the recursion restriction (Definition 2, condition 3).[10]

With the definitions above we can compute the set of ground fluents *FDom* as the domain of the first (and only) argument of the predicate **true**, that is, $dom(\textbf{true}, 1)$. Instead of computing the domain and enumerating all ground fluents in the state generator

$$0 \quad \{\textbf{true}(f, 0) : f \in FDom\}.$$

we encode the domain as an answer set program as follows.

Definition 22. Let $D^+ = (V, E^+)$ be the transitive closure of the domain graph for the GDL specification G. Let $\eta(dom(v))$ be a predicate symbol which represents a unique name for the domain of $v \in V$. The encoding *Dom* of the domains of all predicates and functions of G consists of the following ASP rules:

- $\eta(dom(p/n))(p(X_1, \ldots, X_n)) :- \eta(dom(p, 1))(X_1), \ldots, \eta(dom(p, n))(X_n)$., for each $p/n \in V$.
- $\eta(dom(p, i))(c)$., for each edge $c/0 \to (p, i) \in E^+$.
- $\eta(dom(p, i))(X) :- \eta(dom(q/n))(X)$., for each edge $q/n \to (p, i) \in E^+$ with $n > 0$.

It is easy to see that the (unique) answer set of *Dom* contains $\eta(dom(p, i))(t)$ for some ground term t if, and only if, $t \in dom(p, i)$.

[9] We treat constants as nullary functions.

[10] Otherwise, that is, if G but not G' satisfies the recursion restriction, the set of reachable states of the game and thus the set of ground fluents might be infinite (cf. Section 3.2).

Using the above definition, we encode the state generator as the answer set program consisting of *Dom* and the following rule:

$$0\ \ \big\{\mathbf{true}(F,0) : \eta\big(dom(\mathbf{true},1)\big)(F)\big\}.$$

This rule makes use of the *condition* feature of the Potassco system [10]: The term $p(X) : q(X)$ is automatically expanded to $p(t_1), \ldots, p(t_n)$ where $t_1, \ldots, t_n$ are all ground terms such that $q(t_i)$ holds.

In a similar fashion, we can obtain a straightforward encoding of an action generator:

$$1\big\{\mathbf{does}(r,A,i) : \eta\big(dom(\mathbf{does},2)\big)(A)\big\}1 :\text{-}\ \mathbf{not}\ \texttt{terminated}(i)\,.$$

However, this definition ignores the fact that different roles might have different potential actions and thus yields too many ground instances of actions for many games in practice. In other words, while usually all fluents in $dom(\mathbf{true},1)$ may actually occur in a reachable state, many of the actions in $dom(\mathbf{does},2)$ are never legal for any player. By Definition 21, domains of functions and predicates are essentially computed as the cross product of the domains of their arguments. Consider, then, a game like Checkers and the action `move(Piece,X1,Y1,X2,Y2)` of moving a piece from cell `X1,Y1` to cell `X2,Y2`.[11] In Checkers there are 4 different pieces ("men" and "kings" of either of two colours), and it is played on an 8 by 8 board. Thus, $dom(move/5)$ alone contains $4 * 8^4 = 16{,}384$ ground instances. However, due to the restrictions of how pieces move in Checkers, only a few hundred moves are actually possible. The problem becomes even more apparent with more complicated actions, e.g., `triplejump(Piece,X1,Y1,X2,Y2,X3,Y3,X4,Y4)`.

For these reasons, we encode $ADom(r)$ differently. The idea is to compute a *static* version P^{legal}_{static} of the rules that define the legal moves of the players. Here, static means a relaxation of the rules that is independent of **true**, defined as follows.

Definition 23. Let G be a GDL specification and $\eta(static(p))$ be a predicate symbol which represents a unique name for the static version of predicate p. For each rule $p(\vec{X}) :\text{-}\ B$ such that $p = \mathbf{legal}$ or $\mathbf{legal}$ depends on p in the dependency graph of G with positive edges, P^{legal}_{static} contains the rule

$$\eta\big(static(p)\big)(\vec{X}) :\text{-}\ B_{static}.$$

where B_{static} comprises the following literals:

$$\big\{\eta\big(dom(\mathbf{true},1)\big)(\vec{Y}) : true(\vec{Y}) \in B\big\} \cup$$
$$\big\{\eta\big(static(q)\big)(\vec{Y}) : q(\vec{Y}) \in B \wedge q \neq \mathbf{true}\big\} \cup$$
$$\big\{\mathbf{not}\ q(\vec{Y}) : \mathbf{not}\ q(\vec{Y}) \in B \wedge q \neq \mathbf{true} \wedge q \text{ does not depend on } \mathbf{true}\big\}.$$

Based on this definition, the action generator $P^{legal}_{\leqslant n}$ consists of the clauses $Dom \cup P^{legal}_{static}$ and the following clauses for each $0 \leqslant i \leqslant n$ and $r \in R$; see also its original rules (8) in Section 4.3:

(c_1) $\texttt{terminated}(i) :\text{-}\ \mathbf{terminal}(i)\,.$

(c_2) $\texttt{terminated}(i) :\text{-}\ \texttt{terminated}(i-1)\,.$ (for $i > 0$ only)

(c_3) $1\{\mathbf{does}(r,\texttt{A},i) : \eta(static(\mathbf{legal}))(r,\texttt{A})\}1 :\text{-}\ \mathbf{not}\ \texttt{terminated}(i)\,.$

(c_4) $:\text{-}\ \mathbf{does}(r,\texttt{A},i), \mathbf{not}\ \mathbf{legal}(r,\texttt{A},i)\,.$

6.2. Optimising the rules

Current answer set solvers, such as the Potassco system that we used in our experiments, ground the input answer set program prior to computing solutions. Grounding an ASP increases its size exponentially, in the worst case. The grounding step can easily dominate the step that actually solves the ASP both in memory and run-time requirements. Thus, for an efficient implementation, it is essential to transform the constructed ASPs into a form whose grounding is as small as possible before handing it to the answer set solver.

Some of the following transformations change an answer set program in such a way that it has different models. While we are only interested in the existence of a model in the proof method presented in Section 4, the method for proving multiple properties at once (described in Section 5.1) only works if certain atoms in the model stay unchanged. In particular, the atoms $\eta(\varphi)$ and $\eta(\varphi \supset \bigcirc\varphi)$ for all candidate properties φ must be in all answer sets of the transformed ASP if, and only if, they are in all answer sets of the original ASP. Otherwise, the proof method does not yield the same results with the transformed answer set program. The atoms that need to be preserved in the answer sets are called *used atoms*.

Based on these observations, we impose the following restriction on all transformations of answer set programs.

[11] See the repository ggpserver.general-game-playing.de/public/show_games.jsp for a complete encoding of Checkers.

Definition 24. Let P be an answer set program, $t(P)$ be the transformed program according to transformation t, and U be the set of used atoms. We call t a *valid transformation w.r.t. U* iff for every answer set $\mathcal{A}$ of P there is an answer set $\mathcal{A}'$ of $t(P)$ such that $\mathcal{A} \cap U = \mathcal{A}' \cap U$ and vice versa.

Note that $t(P)$ does not necessarily have the same number of answer sets. Especially if $U = \emptyset$ (say, in case we are only interested in the existence of an answer set), it suffices for $t(P)$ to admit one answer set if P has at least one.

In the following, we will present several optimisations that reduce the size of answer set programs. All of these optimisations are valid transformations according to the above definition w.r.t. the atoms used in the proof methods presented in this paper.

No time argument for static predicates. In Section 4.1, we introduced the temporal extension of a GDL specification. Essentially, we added a time argument to every predicate of the specification. However, some of the predicates of a game description are static, that is, do not depend on the state of the game or the moves of the players. Thus, they are equivalent in every time step. For example, in Quarto (Fig. 1) all of the predicates **role**, **distinct**, `sameattr`, `nthbit`, `!=`, `otherrole`, and `index` are static.

Formally, predicate p is static, if there is no path from p to **true** or **does** in the dependency graph (cf. Definition 2) for the GDL description. We reduce the size of the generated answer set program by omitting the time argument that is added in Definition 10, if a predicate is static. Thus, rules for those predicates are only added once to the answer set program independent of the number of time steps.

Cutting indirections. The number of rules can be further reduced by removing unnecessary indirections, that is, predicates that are defined by only one rule. For example, the encoding of temporal formulas presented in Table 2 produces a predicate for every sub-formula of a formula. The predicates for atomic formulas (1), negations (2), conjunctions (3), and counting quantifiers (6) occur as the head of one rule only. Unless the encoded formula contains the same sub-formula several times, all of the generated predicates occur only once in the body of some rule.

We reduce the number of rules and the number of predicates in an answer set program in the following way. Let P be an answer set program with a rule $r = p(\vec{x}) :- Body.$ such that

- r is the only rule with predicate p in the head,
- $p(\vec{x})$ is not used in the answer set(s) of P, and
- p does neither occur negated nor in a weight atom in P.

Then the transformed program $t(P)$ is obtained from P by removing rule r and replacing every atom $p(\vec{y})$ in P with $(Body\,\sigma)$, if $p(\vec{x})$ and $p(\vec{y})$ unify with most general unifier σ. Also removed are rules containing atoms $p(\vec{y})$ that are not unifiable with $p(\vec{x})$.

Removing existential variables. The size of the grounding of an ASP is strongly influenced by the number of variables in a rule. In the worst case the number of ground clauses is exponential in the number of variables. This number can often be reduced by introducing new predicates and rules. Consider, e.g., the rule `p(X,Z) :- q(X,Y), r(Y), s(Z).`, where `Y` in the body is existentially quantified. If we replace this rule by

```
p(X,Z) :- qr(X), s(Z).
qr(X)  :- q(X,Y), r(Y).
```

where `qr` is a new predicate symbol, we obtain two rules with two variables each instead of one rule with three variables. This changes the number of ground clauses from $|dom(X)| * |dom(Y)| * |dom(Z)|$ to $|dom(X)| * |dom(Z)| + |dom(X)| * |dom(Y)|$, which amounts to a considerable reduction if the domains of the variables are large enough (> 2).

We apply the following transformation to all rules in an answer set program P until a fixed point is reached: Consider a rule $(head :- body.) \in P$ containing a variable V which does not occur in *head*. We split *body* into $body^+$ and $body^-$ in such a way that $body^+$ consists of all literals from *body* that contain V, and $body^-$ are the remaining literals from *body*. Let $\{V_1, \ldots, V_n\}$ be the set of variables in $body^+$ that also occur in $body^-$ or in the head of the rule. Furthermore, let $\{V_1^-, \ldots, V_m^-\}$ be the variables in $body^+$ that only occur in negative literals in $body^+$ and let $dom_1, \ldots, dom_m$ be predicate symbols from *Dom* (cf. Definition 22) encoding their respective domains. If $body^-$ contains a variable that is not contained in $\{V_1, \ldots, V_n\}$, then $head :- body.$ is replaced by the rules

$$head :- p(V_1, \ldots, V_n), body^-.$$
$$p(V_1, \ldots, V_n) :- body^+, dom_1(V_1^-), \ldots, dom_m(V_m^-).$$

where p is a predicate symbol that does not occur anywhere else in P. Note that the addition of the domain restrictions $dom_1(V_1^-), \ldots, dom_m(V_m^-)$ to the second rule is necessary to ensure that the resulting rules are allowed, i.e., to ensure that each variable in a rule occurs in at least one positive literal in the body. In principle, the domain of each variable can be inferred from its context by taking the intersection of all argument domains from positive literals in *body* in which the

variable occurs. However, for a correct implementation, it is sufficient to take only one such argument domain. Thus, for each $V_j^- \in \{V_1^-, \ldots, V_m^-\}$, we use the domain predicate $dom_j = \eta(dom(p_j, a_j))$ for an arbitrary function or predicate p_j in the positive literals in *body* in which V_j^- occurs as the a_j-th argument.

Removing unnecessary rules. Depending on the formula that is to be proved, some of the rules in the generated ASP might not be necessary. For example, the temporal GDL extension contains rules with head $\mathbf{true}(f, i+1)$ for all time steps i obtained from the next rules of the game. However, the remaining rules of the ASP do not contain any instance of $\mathbf{true}(f, n+1)$ for the last time step n. Hence, removing those rules does not change the number of answer sets of the program. The same applies to $\mathbf{legal}(r, a, n)$: Legality of moves is irrelevant in the last time step (n), unless of course the formula to be proved depends on the atom $\mathbf{legal}(r, a)$. Removing unnecessary rules from an ASP reduces the size of the grounded program and thus the cost for both grounding and solving the answer set program.

To capture the notion of necessary vs. unnecessary rules, we first extend the definition of dependency graphs (cf. Definition 2) to ground atoms. An extended dependency graph differs from the standard graph in that it has ground atoms as nodes instead of predicate symbols, and there is an edge $p(\vec{t}_p) \to q(\vec{t}_q)$ in the graph whenever there is a ground instance of a rule with head $p(\vec{t}_p)$ and $q(\vec{t}_q)$ in the body.

Based on this notion of an extended dependency graph, we compute the set of necessary atoms for an answer set program P in the following way. Let U be the set of used atoms for the proof (cf. Definition 24); say $U = \emptyset$, if we are only interested in the existence of an answer set. Furthermore, let U_c be the ground instances of all atoms that occur in constraints or weight atoms in P. Then, the set of necessary atoms is

$$\hat{U} = U \cup U_c \cup \left\{q : (\exists p)\ p \in (U \cup U_c) \wedge p \to^* q\right\}$$

where $p \to^* q$ denotes the existence of a path from p to q in the extended dependency graph of P. To conclude, we remove all clauses from P whose head is an atom that does not unify with any atom in $\hat{U}$.

6.3. Experimental results

We conducted experiments with our system using a wide range of games from the past AAAI General Game Playing Competitions, all of which are available on the online game repositories at Dresden[12] and Stanford.[13] The results are summarised in Fig. 3. For each game the following sets of formulas were generated:

Functionals In Quarto, each cell contains at most one piece (cf. property (2), page 7). For example, fluent $cell(1, 1, p1111)$ means that cell $(1, 1)$ houses piece $p1111$. More generally speaking, for every pair of cell coordinates (x, y) there is always at most one p such that $cell(x, y, p)$ is true. Similar properties hold in many games, and in order to detect these, we generate all formulas of the form

$$(\forall \vec{X} : D_{\vec{X}})\ (\exists_{l..1} \vec{Y} : D_{\vec{Y}})\ true\big(f(\vec{Z})\big)$$

for each fluent symbol f, each $l \in \{0, 1\}$, and each nonempty subsequence $\vec{Y}$ of the variables in $f(\vec{Z})$.[14] In addition, we identify as *control fluents* those that have one argument that ranges over the roles (e.g., *sctrl* and *pctrl* in Fig. 1, page 5). If the set F_c of these fluents contains two or more elements, we also attempt to prove that exactly one of them holds at any time via the formula $(\exists_{1..1} F : F_c)\ true(F)$.

Legals We include the state sequence invariant for Playability (cf. Section 5.2),

$$\neg terminal \supset (\forall R : D_R)\ (\exists M : D_M)\ legal(R, M),$$

where D_R is the set of roles and D_M the (finite) domain of moves. In addition, we attempt to prove the property Turn-Taking. In GDL, each turn-taking game has to be modelled by simultaneous moves, which can be achieved with the help of a pseudo action like *noop* in Quarto that has no effects. We consider a game to be turn-taking if at most one player has two or more legal moves in each reachable game state. With D_R and D_M as above, this can be expressed via

$$(\exists_{0..1} R : D_R)\ (\exists_{2..\infty} M : D_M)\ legal(R, M).$$

Note that this formula makes no reference to the actually used name of a noop action. The name is hence completely independent from the proof result, and does not even have to be the same in each game state.

[12] ggpserver.general-game-playing.de/public/show_games.jsp.

[13] games.stanford.edu/resources/resources.html.

[14] In the formula above, $\vec{X}$ is the (possibly empty) sequence of all variables in $\vec{Z}$ that are not in $\vec{Y}$.

game	Functionals		Legals (pl, tt)		Goal (z, u, m)		Persistence	
3pttc	0.34	(4/10)	0.77	(y,y)	0.30	(?,y,y)	0.73	(77/354)
bidding-tictactoe	0.17	(1/10)	0.09	(?,n)	0.17	(?,?,?)	0.21	(9/89)
breakthrough	0.81	(3/3)	1.41	(y,y)	1.45	(y,y,y)	1.06	(32/242)
capture_the_king	–	(3/8)	3.26	(?,y)	3.37	(y,y,n)	65.81	(7/1710)
catcha_mouse	0.27	(4/5)	0.15	(?,y)	0.36	(?,y,y)	1.19	(359/896)
CephalopodMicro	1.70	(5/17)	0.49	(y,y)	1.47	(y,y,y)	1.59	(18/209)
checkers	–	(3/8)	–		9.90	(?,?,?)	–	
chinesecheckers6	–	(4/6)	1.27	(y,y)	45.26	(?,?,y)	29.12	(80/634)
chomp	0.13	(3/4)	0.08	(?,y)	0.12	(y,y,y)	0.12	(58/61)
connect4	0.22	(2/3)	0.24	(?,y)	0.37	(y,y,?)	0.29	(294/492)
endgame	–	(4/6)	3.23	(?,y)	1.22	(y,y,?)	7.15	(2/511)
knightfight	0.81	(3/10)	2.25	(?,y)	0.83	(?,?,n)	1.59	(100/602)
kriegtictactoe	0.15	(4/11)	0.08	(?,y)	0.21	(y,y,?)	0.14	(27/74)
othello-comp2007	4.80	(3/5)	1.55	(y,y)	–		3.62	(8/250)
pawn_whopping	0.29	(3/5)	1.62	(y,y)	0.25	(y,y,n)	0.39	(32/234)
quarto	12.19	(6/7)	9.09	(?,y)	–		28.44	(288/582)
smallest	1.01	(4/4)	0.15	(y,n)	15.84	(?,y,y)	0.53	(12/148)
tictactoe	0.13	(4/4)	0.19	(y,y)	0.14	(y,y,n)	0.10	(27/38)
tttcc4	10.47	(4/8)	18.04	(y,y)	1.72	(?,y,y)	14.97	(311/1228)

Fig. 3. Property proof times in seconds for a variety of games from the past AAAI General Game Playing Competitions ("–" means that the prover was aborted after 100 seconds). Information in parentheses: (m/n)–m formulas proved true out of all n formulas from the respective set which are true in the initial state; (y)–formula proved true; (n)–formula invalid in the initial state; (?)–formula invalid in some (not necessarily reachable) state; (pl, tt)–pl is the result for playability and tt the result for turn-taking; (z, u, m)–z is the result for zero-sum, u the result for uniqueness of goal values, and m the result for monotonically increasing goal values. Experiments were run on an Intel Core 2 Duo CPU with 3.16 GHz.

Goal In all games at the AAAI Competition the goal values range from 0 to 100 [25]. Hence, a game is to be considered zero-sum if the goal values of all players, in case they exist, add up to 100 in each reachable terminal state. This we formulate via the state sequence invariant

$$\mathit{terminal} \supset \bigwedge_{\substack{g_1,\ldots,g_n \in GV \\ g_1+\cdots+g_n \neq 100}} \big(\neg \mathit{goal}(r_1, g_1) \vee \cdots \vee \neg \mathit{goal}(r_n, g_n)\big),$$

where $D_R = \{r_1, \ldots, r_n\}$ is the set of roles and GV is the (finite) set of goal values that occur in the game description. Using the same identifiers, we furthermore include a formula which expresses that the goal values of all players are unique in each terminal state,

$$(\forall R : D_R)\ \big(\mathit{terminal} \supset (\exists_{1..1} V : GV)\ \mathit{goal}(R, V)\big).$$

We call a game monotonic if, for each player r and each reachable state S, r has exactly one goal value in S, and goal values never decrease in the course of the development of the game. To formulate this property, we include a third formula

$$(\forall R : D_R)\ (\varphi_1 \wedge \varphi_2), \quad \text{where}$$

- φ_1 expresses that the goal value for r is unique in each reachable state:

$$\varphi_1 = (\exists_{1..1} V : GV)\ \mathit{goal}(R, V), \quad \text{and}$$

- φ_2 formulates that each goal value for r in a state is not higher than any goal value for r in any of its direct successor states:

$$\varphi_2 = \neg \mathit{terminal} \supset \bigwedge_{\substack{v_1, v_2 \in GV \\ v_1 > v_2}} \neg\big(\mathit{goal}(R, v_1) \wedge \bigcirc \mathit{goal}(R, v_2)\big).$$

Persistence In Quarto, once a piece is placed on the board, it remains in this cell for the rest of the game. Likewise, pieces are always permanently removed from the pool. Similar properties occur in a variety of games, and in order to detect these, we generate the set of all formulas of the form

$$\mathit{true}\big(f(\vec{t})\big) \supset \bigcirc\ \mathit{true}\big(f(\vec{t})\big) \quad \text{and} \quad \neg\,\mathit{true}\big(f(\vec{t})\big) \supset \bigcirc \neg\,\mathit{true}\big(f(\vec{t})\big),$$

where $f(\vec{t})$ is any ground fluent instance in the game in question.

Proving a multitude of similar properties in one run spares the solver from repeating the same tasks over and over again, such as grounding, indexing, and several clause optimisations. This significantly lowers overall time consumption. However,

proving *all* of the aforementioned formulas together does not yield optimal results, because different kinds of formulas tend to require different rules from the game description for verification and to allow fewer clause optimisations when attempted jointly. This motivated the following setup for our experiments.

Functionals are simple-structured and provide valuable state space restrictions, hence they are the first to be tested and then included in all subsequent proofs. We perform a second run, which is likely to produce further successfully proved functionals since these are often interdependent—recall that in order to prove that each cell contains at most one piece (property (2), page 7) in Quarto we first needed to discover that always exactly one instance of a control fluent holds. Proof attempts for Legals were run separately since, unlike in the case of Functionals, their induction step requires clauses with head **legal** for time step 1. Due to their complex structure (their encoding refers to *all* legal moves of the game) their addition to the established facts tends to slow down subsequent proof attempts, which is why they are not considered in any further proofs. The property class Goal contains the only properties we considered that refer to predicate **goal** and the defining clauses; hence they are attempted in a further distinct run and the results are also not included subsequently. Persistence formulas have a higher degree and thus require a copy of the GDL clauses with an additional time step, which is why they are proved together in yet another separate run.

In Quarto, the prover successfully shows that in all reachable game states there is at most one selected piece; that exactly one player has control over either selecting or placing a piece; and that each cell has exactly one value (that is, a piece or *empty*). Moreover, it proves that a piece which is placed in a cell remains in this cell; that a piece which is removed from the pool stays removed; and that a nonempty cell never becomes empty again. Results for the other formula sets are summarised in Fig. 3, together with results for a variety of other games. Times in column "Functionals" indicate two proof attempts (each attempt including one ASP proof for the base case and one ASP proof for the induction step), the other times indicate one attempt. The times include both generation and grounding of the respective answer set programs (the latter is done by Clingo). Additional time in the range of a few seconds is needed for the initialisation of Fluxplayer (which includes the calculation of the domain graph) once for each newly considered game. In general, many instances of the four property classes can be proved within at most a few seconds, which demonstrates that our proof method is applicable even in the usual time setting of a General Game Playing Competition. Proving multiple properties at once is especially effective with Persistence properties, which usually requires to check several hundred instances per game. Also more complex games like the chess variants "endgame" and "capture_the_king" yield practicable results. Checkers, on the other hand, cannot be handled efficiently due to its inherent vast amount of legal moves comprising simple piece moves, double jumps, and triple jumps.

The results in Fig. 3 are almost exclusively for games with perfect information, owing to the fact that imperfect information games are not yet included in the AAAI Competition. The only exception in Fig. 3 is "kriegtictactoe," which has been defined in [36] inspired by the game Kriegspiel (see, e.g., [30,33]) and which deviates from standard Tic-Tac-Toe in that the players cannot see each other's moves. The results indicate that imperfect information in Krieg-Tic-Tac-Toe has no influence on the properties which can be proved except for Playability, which presupposes the earlier proof of an additional property that is not considered here.

7. Conclusion

Automated theorem proving enables general game-playing systems to infer properties of new games that follow from the rules without being explicitly given. It also enables game designers to automatically verify desired properties of their formal game descriptions. In this paper, we have first defined syntax and semantics of a formal language for describing game-specific properties as state sequence invariants in the context of General Game Playing. We have then shown how these formulas can be encoded as a logic program, and we have developed a proof theory with the help of Answer Set Programming. Finally, we have reported on systematic experiments with a variety of games that have been used by the scientific community in the past. While the main focus of this paper is theoretical, our experimental results show that both game designers and general game-playing systems can make practical use of our method to automatically verify game-specific knowledge against a previously unknown game description.

In terms of related work, the syntax and semantics of our language for state sequence invariants is inspired by work on control knowledge in planning problems [3], which relates actions (with preconditions and effects) to joint moves (with legality and update specifications). The construction of temporally extended rules (cf. Section 4.1) is a well-known technique of enriching action laws in order to reason about sequences of actions and to solve planning problems; (see, e.g., [18,15]). Our method of automatically proving temporally extended properties for general games has been preceded by the approach presented in [32], but theirs requires to systematically search the entire set of reachable positions in a game and is therefore limited to very simple games in practice.

All of the methods presented in this paper have been developed to prove factual properties of games. In this regard, our techniques apply equally to both games described in standard GDL as well as imperfect-information games given in the recently extended description language [37]. The latter, however, opens the road for additionally investigating properties about the *knowledge* of individual players [31]. A relevant and significant aspect of future work is to investigate how our framework can be generalised in order to formalise, encode, and automatically prove knowledge properties against formal descriptions of games with imperfect and asymmetric information.

Acknowledgements

We thank the anonymous reviewers for their very valuable, extensive feedback and constructive suggestions for improving the paper. This research was supported under Australian Research Council's (ARC) *Discovery Projects* funding scheme (project number DP 120102023) and by the Icelandic Centre for Research (RANNIS, grant number 210019). Michael Thielscher is the recipient of an ARC Future Fellowship (project number FT 0991348). He is also affiliated with the University of Western Sydney.

Appendix A. Proofs of theorems

Theorem 12. *Let G be a valid GDL specification and $S_0 \xrightarrow{A_0} S_1 \ldots \xrightarrow{A_{m-1}} S_m$ a sequence. Consider the program $P = S_0^{\mathtt{true}}(0) \cup G_{\leqslant m} \cup \bigcup_{i=0}^{m-1} A_i^{\mathtt{does}}(i)$, then for all $0 \leqslant i \leqslant m$ and predicate symbols $p \notin \{\mathbf{init}, \mathbf{next}\}$ that do not depend on* **does***, we have*

$$G \cup S_i^{\mathtt{true}} \vdash p(\vec{t}) \quad \textit{iff} \quad P \vdash p(\vec{t}, i).$$

Proof. Let $P_0 = S_0^{\mathtt{true}}(0)$ and $P_m = G_{m-1} \cup A_{m-1}^{\mathtt{does}}(m-1) \cup P_{m-1}$ for $m > 0$. We first prove the intermediate result

$$S_m = \{f : P_m \vdash \mathbf{true}(f, m)\} \tag{A.1}$$

by induction on m. The base case $m = 0$ is immediate. Induction step: By Definition 5 we have

$$f \in S_{m+1} \quad \text{iff} \quad f \in u(A_m, S_m) \quad \text{iff} \quad G \cup A_m^{\mathtt{does}} \cup S_m^{\mathtt{true}} \vdash \mathbf{next}(f).$$

Using the induction hypothesis, this is equivalent to

$$G \cup A_m^{\mathtt{does}} \cup \{\mathbf{true}(f') : P_m \vdash \mathbf{true}(f', m)\} \vdash \mathbf{next}(f).$$

The clauses from G which solely contribute to the initial state encoding do not influence entailment of $\mathbf{next}(f)$, since their heads do not occur in the remaining clauses. Together with the construction of the temporal GDL extension from Definition 10, this yields equivalence to

$$G_m \cup A_m^{\mathtt{does}}(m) \cup \{\mathbf{true}(f', m) : P_m \vdash \mathbf{true}(f', m)\} \vdash \mathbf{true}(f, m+1).$$

Similarly, atoms from P_m other than $\mathbf{true}(f, m)$ do not influence entailment of $\mathbf{true}(f, m+1)$, hence we get equivalence to

$$G_m \cup A_m^{\mathtt{does}}(m) \cup \{p : P_m \vdash p\} \vdash \mathbf{true}(f, m+1).$$

Since P_m does not contain heads of $G_m \cup A_m^{\mathtt{does}}(m)$, we can apply Theorem 11 (page 12) to establish equivalence to

$$G_m \cup A_m^{\mathtt{does}}(m) \cup P_m \vdash \mathbf{true}(f, m+1).$$

Since $P_{m+1} = G_m \cup A_m^{\mathtt{does}}(m) \cup P_m$, this completes the induction step and hence proves the intermediate result (A.1). For the remainder, it follows

$$G \cup S_i^{\mathtt{true}} \vdash p(\vec{t}) \quad \text{iff} \quad G \cup \{\mathbf{true}(f') : P_i \vdash \mathbf{true}(f', i)\} \vdash p(\vec{t}),$$

which in turn, by arguments similar to those for the intermediate result, is equivalent to

$$G_i \cup P_i \vdash p(\vec{t}, i).$$

Case $i = m$ yields $P = G_i \cup P_i$. Case $i < m$: the unique answer sets for $G_i \cup P_i$ and $G_i \cup P_i \cup A_i^{\mathtt{does}}$ agree on the true instances of $p(\vec{t}, i)$, as $p(\vec{t}, i)$ does not depend on **does**. Since $G_i \cup P_i \cup A_i^{\mathtt{does}}$ does not contain clause heads from $P \setminus (G_i \cup P_i \cup A_i^{\mathtt{does}})$, entailment of $p(\vec{t}, i)$ is again not affected. Hence both cases $i = m$ and $i < m$ yield equivalence to

$$P \vdash p(\vec{t}, i). \qquad \square$$

Theorem 14. *Let G be a valid GDL specification and φ be a sequence invariant. Then $Enc(\varphi) := Enc(\varphi, 0)$ with the unique name atom $\eta(\varphi) := \eta(\varphi, 0)$ for φ (cf. Table 2) is an encoding of φ.*

Proof. Let $\hat{n} \geqslant \deg(\varphi)$, $S_0 \xrightarrow{A_0} S_1 \ldots \xrightarrow{A_{\hat{m}-1}} S_{\hat{m}}$ an arbitrary $\hat{n}$-max sequence and $P_\varphi = S_0^{\mathtt{true}}(0) \cup G_{\leqslant \hat{n}} \cup \bigcup_{i=0}^{\hat{m}-1} A_i^{\mathtt{does}}(i) \cup Enc(\varphi, 0)$. P_φ clearly admits a unique answer set. We prove $(S_0, \ldots, S_{\hat{m}}) \vDash \varphi$ iff $P_\varphi \vdash \eta(\varphi, 0)$ via structural induction on φ. First note that the uniqueness of $\eta(\psi, i)$ for each formula ψ and each time step i implies $\eta(\psi, i)$ to be in the unique answer set $\mathcal{A}$ for P_φ if and only if there is a clause with head $\eta(\psi, i)$ in $Enc(\varphi, 0)$ such that its body is satisfied in $\mathcal{A}$.

Base Case $\varphi = p(\vec{t})$: $(S_0, \ldots, S_{\hat{m}}) \vDash p(\vec{t})$ iff (by Definition 8) $G \cup S_0^{\mathtt{true}} \vdash p(\vec{t})$ iff (by Theorem 12, cf. the remark following Definition 13) $P_\varphi \vdash p(\vec{t}, 0)$ iff $P_\varphi \vdash \eta(p(\vec{t}), 0)$.

Induction Step: The cases different from $\varphi = \bigcirc\psi$ follow by an argumentation similar to the base case, together with the induction hypothesis. Now consider formula $\varphi = \bigcirc\psi$ with degree $n+1$.

- If S_0 is terminal: $(S_0) \vDash \bigcirc\psi$ follows by Definition 8, $P_\varphi \vdash \textbf{terminal}(0)$ follows by Theorem 12 and yields $P_\varphi \vdash \eta(\varphi, 0)$.
- If S_0 is nonterminal then $(S_1, \ldots, S_{\hat{m}}, S_{\hat{m}+1})$ exists and is $\hat{n}$-max, hence

$$(S_0, S_1, \ldots, S_{\hat{m}}, S_{\hat{m}+1}) \vDash \bigcirc\psi \quad \text{iff} \quad (S_1, \ldots, S_{\hat{m}}, S_{\hat{m}+1}) \vDash \psi.$$

Let $\cdot^{i\to i+1}$ be a renaming that replaces each time argument i by $i+1$ in timed GDL atoms and each occurrence of $\eta(\rho, i)$ by $\eta(\rho, i+1)$ for each formula ρ. Then, for program $P_\psi^{i\to i+1} = S_1^{\mathtt{true}}(1) \cup (G_{\leqslant \hat{n}+1} \setminus G_0) \cup \bigcup_{i=1}^{\hat{m}} A_i^{\mathtt{does}}(i) \cup Enc(\psi, 1)$, the induction hypothesis implies

$$(S_1, \ldots, S_{\hat{m}}, S_{\hat{m}+1}) \vDash \psi \quad \text{iff} \quad P_\psi^{i\to i+1} \vdash \eta(\psi, 1).$$

Since $S_1^{\mathtt{true}}(1) = \{\textbf{true}(f, 1) : G_0 \cup A_0^{\mathtt{does}}(0) \cup S_0^{\mathtt{true}}(0) \vdash \textbf{true}(f, 1)\}$ (by Definitions 5 and 10) and because clause heads in $P_\psi^{i\to i+1}$ do not occur in $G_0 \cup A_0^{\mathtt{does}}(0) \cup S_0^{\mathtt{true}}(0)$ and clause heads in $G_0 \cup A_0^{\mathtt{does}}(0) \cup S_0^{\mathtt{true}}(0)$ different from atoms in $S_1^{\mathtt{true}}(1)$ do not occur in $P_\psi^{i\to i+1}$, we have that

$$P_\psi^{i\to i+1} \vdash \eta(\psi, 1) \quad \text{iff} \quad S_0^{\mathtt{true}}(0) \cup G_{\leqslant \hat{n}+1} \cup \bigcup_{i=0}^{\hat{m}} A_i^{\mathtt{does}}(i) \cup Enc(\psi, 1) \vdash \eta(\psi, 1).$$

This, in turn, is equivalent to $P_\varphi \vdash \eta(\varphi, 0)$ since $P_\varphi \nvdash \textbf{terminal}$ by Theorem 12. □

Theorem 16. *Let G be a valid GDL specification and $\mathcal{A}$ be a subset of the ground atoms over G together with $\{\mathtt{terminated}(i) : i \in \mathbb{N}\}$. The following two statements are equivalent*:

(1) *$\mathcal{A}$ is an answer set for*

$$P = S_0^{\mathtt{true}}(0) \cup G_{\leqslant n} \cup P_{\leqslant n-1}^{legal}.$$

(2) *There is an n-max sequence $Seq = (S_0 \xrightarrow{A_0} S_1 \ldots \xrightarrow{A_{m-1}} S_m)$ such that $\mathcal{A}$ is the unique answer set for*

$$P^{Seq} = S_0^{\mathtt{true}}(0) \cup G_{\leqslant n} \cup P_{\leqslant n-1}^{c_1,c_2} \cup \bigcup_{i=0}^{m-1} A_i^{\mathtt{does}}(i),$$

where $P_{\leqslant n-1}^{c_1,c_2} = \bigcup_{i=0}^{n-1} P_i^{c_1,c_2}$ and $P_i^{c_1,c_2}$ denotes all clauses of the shape (c_1) and (c_2) in the part P_i^{legal} of the action generator, defined as (8) *on page* 15.

Proof. (2) ⇒ (1): First we show that $\mathcal{A}$ satisfies P. Since P differs from P^{Seq} only by containing clauses of the shape (c_3) and (c_4) (defined as part of the action generator in (8) on page 15) instead of $\bigcup_{i=0}^{m-1} A_i^{\mathtt{does}}(i)$, this follows if $\mathcal{A}$ satisfies all clauses (c_3) and (c_4) for $0 \leqslant i \leqslant n-1$.

- Time steps $0 \leqslant i < m$: for each player r there is exactly one action a such that $\textbf{does}(r, a, i) \in \mathcal{A}$, namely $a = A_i(r)$; and $\textbf{legal}(r, a, i) \in \mathcal{A}$ follows by definition of *Seq* and Theorem 12. This satisfies the clauses (c_3) and (c_4) for $0 \leqslant i < m$.
- Time steps $m \leqslant i \leqslant n-1$: If one such i exists, then $m < n$ and hence S_m is terminal, which implies $\textbf{terminal} \in \mathcal{A}$ (again by Theorem 12) and thus $\{\mathtt{terminated}(j) : m \leqslant j \leqslant n-1\} \subseteq \mathcal{A}$. This satisfies the clauses (c_3) and (c_4) for $m \leqslant i \leqslant n-1$.

Now $\mathcal{A}$ is also an answer set for the program constructed from P by omitting constraints (c_4), since its reduct coincides with the reduct of P^{Seq}. By the previous argumentation $\mathcal{A}$ satisfies all constraints (c_4) and hence is also an answer set for P.

(1) ⇒ (2): Let G_n^{dep} be the clauses from G_n whose heads depend on **does**, and let $G_n^{\overline{dep}}$ be all others. By induction on n, we prove that if

$$P_n = S_0^{\mathtt{true}}(0) \cup G_{\leqslant n-1} \cup G_n^{\overline{dep}} \cup P_{\leqslant n-1}^{legal}$$

has answer set $\mathcal{A}_n$, then there is an n-max sequence $Seq = (S_0 \xrightarrow{A_0} S_1 \ldots \xrightarrow{A_{m-1}} S_m)$ such that $\mathcal{A}_n$ is the unique answer set for

$$P_n^{Seq} = S_0^{\mathtt{true}}(0) \cup G_{\leqslant n-1} \cup G_n^{\overline{dep}} \cup P_{\leqslant n-1}^{c_1,c_2} \cup \bigcup_{i=0}^{m-1} A_i^{\mathtt{does}}(i).$$

This implies the claim for $P = P_n \cup G_n^{dep}$ and $P^{Seq} = P_n^{Seq} \cup G_n^{dep}$ by Theorem 11, since P_n and P_n^{Seq} do not contain heads of G_n^{dep}. For the Base Case $n = 0$ the two programs coincide. Induction Step: Assume that P_{n+1} has answer set $\mathcal{A}_{n+1}$. Since $P_{n+1} = P_n \cup G_n^{dep} \cup G_{n+1}^{\overline{dep}} \cup P_n^{legal}$, P_n does not contain heads of $P_{n+1} \setminus P_n$, hence (by Theorem 11) P_n has an answer set $\mathcal{A}_n$ such that $\mathcal{A}_n \subseteq \mathcal{A}_{n+1}$. By the induction hypothesis there is an n-max sequence $Seq = (S_0 \xrightarrow{A_0} S_1 \dots \xrightarrow{A_{m-1}} S_m)$ such that $\mathcal{A}_n$ is the unique answer set for P_n^{Seq}. We consider two cases:

- S_m terminal: Seq is also $(n+1)$-max. We have $\mathbf{terminal}(m) \in \mathcal{A}_n$ (by Theorem 12) and hence $\{\texttt{terminated}(i) : m \leqslant i \leqslant n\} \subseteq \mathcal{A}_{n+1}$. This implies that $\mathcal{A}_{n+1}$ does not contain any instance $\mathbf{does}(r, a, n)$, hence $\mathcal{A}_{n+1}$ is also an answer set for P_{n+1}^{Seq}.
- S_m nonterminal: Then $m = n$, hence $\mathbf{terminal}(i) \notin \mathcal{A}_n$ for all $0 \leqslant i \leqslant n$ (by Theorem 12) and hence $\texttt{terminated}(n) \notin \mathcal{A}_{n+1}$. By (c_3) and (c_4) there is a mapping A_n such that for each $r \in R$ there is exactly one a such that $\{\mathbf{legal}(r, a, n), \mathbf{does}(r, a, n)\} \subseteq \mathcal{A}_{n+1}$. All $\mathbf{legal}(r, a, n)$ must also be in $\mathcal{A}_n$ (as in $P_{n+1} \setminus P_n$ these heads do not exist) and hence (again by Theorem 12) we have $S_n \xrightarrow{A_n} S_{n+1}$ for some S_{n+1}. In this case $Seq' = (S_0, \dots, S_n, S_{n+1})$ is $(n+1)$-max. By construction, $\mathcal{A}_{n+1}$ is the unique answer set for $P_{n+1}^{Seq'}$. □

Theorem 17. *Let G be a playable and valid GDL specification whose initial state is S_{init}. Let Ψ be a set of sequence invariants over G which are satisfied in all reachable states, and let φ be a sequence invariant. If $P_\varphi^{bc}(G)$ and $P_{\varphi,\Psi}^{is}(G)$ are inconsistent, then for all finite sequences $(S_{init}, S_1, \dots, S_k)$ we have $S_k \vDash \varphi$.*

Proof. Let $\deg(\varphi) = n$. The proof is via induction on k. For the base case, we prove that $P_\varphi^{bc}(G)$ being inconsistent implies $S_{init} \vDash \varphi$. For the induction step, we prove that if there are S_k, A_k, and S_{k+1} such that $S_k \vDash \varphi$ and $S_k \xrightarrow{A_k} S_{k+1}$, then $P_{\varphi,\Psi}^{is}(G)$ being inconsistent implies $S_{k+1} \vDash \varphi$.

Base Case: We prove that if $S_{init} \nvDash \varphi$, then $P_\varphi^{bc}(G)$ admits an answer set.

$S_{init} \nvDash \varphi$ implies that there is an n-max sequence $Seq = S_{init} \xrightarrow{A_0} S_1 \dots \xrightarrow{A_{m-1}} S_m$ such that $(S_{init}, S_1, \dots, S_m) \nvDash \varphi$. Now let P^{Seq} and P be as in Theorem 16 (where $S_{init} = S_0$). $P^{Seq} \cup Enc(\varphi)$ admits a unique answer set $\mathcal{A}$. By Definition 13 we have $\eta(\varphi) \notin \mathcal{A}$, hence $\mathcal{A}$ is also the unique answer set for $P^{Seq} \cup Enc(\varphi) \cup \{:\!-\eta(\varphi)\,.\}$. P^{Seq} and $P = P_\varphi^{bc}(G) \setminus (Enc(\varphi) \cup \{:\!-\eta(\varphi)\,.\})$ do not contain heads of $Enc(\varphi) \cup \{:\!-\eta(\varphi)\,.\}$, hence by Theorem 11 and Theorem 16, $\mathcal{A}$ is also an answer set for $P_\varphi^{bc}(G)$.

Induction Step: Let $\hat{n} = \max(n_\Psi, n+1)$ for the maximal degree n_Ψ of formulas in Ψ. Assume $S_k \xrightarrow{A_k} S_{k+1}$ for some A_k and S_{k+1}. We prove that if $S_{k+1} \nvDash \varphi$, then $P_{\varphi,\Psi}^{is}(G)$ admits an answer set.

$S_{k+1} \nvDash \varphi$ implies that there is an n-max sequence $S_{k+1} \xrightarrow{A_{k+1}} S_{k+2} \dots \xrightarrow{A_{k+m}} S_{k+m+1}$ (where $0 \leqslant m \leqslant n$) such that $(S_{k+1}, \dots, S_{k+m+1}) \nvDash \varphi$. It follows that $Seq_{n+1} = S_k \xrightarrow{A_k} S_{k+1} \xrightarrow{A_{k+1}} S_{k+2} \dots \xrightarrow{A_{k+m}} S_{k+m+1}$ is $(n+1)$-max and that $Seq_{n+1} \nvDash \bigcirc\varphi$. Furthermore, by the induction hypothesis we have $S_k \vDash \varphi$ and hence also $Seq_{n+1} \vDash \varphi$ by Proposition 9. These arguments imply that $Seq_{n+1} \nvDash \varphi \supset \bigcirc\varphi$.

Since S_k is reachable and the GDL specification is playable, Seq_{n+1} can be extended to an $\hat{n}$-max sequence $Seq_{\hat{n}}$ by Proposition 9 such that $Seq_{\hat{n}} \nvDash \varphi \supset \bigcirc\varphi$, and S_k satisfying each $\psi \in \Psi$ also implies $Seq_{\hat{n}} \vDash \psi$. An argumentation similar to the base case—considering $\varphi \supset \bigcirc\varphi$ instead of φ, $\hat{n}$ instead of n, $Seq_{\hat{n}}$ instead of Seq, and the additional subprogram $\bigcup_{\psi\in\Psi}(Enc(\psi) \cup \{:\!-\mathbf{not}\,\eta(\psi)\,.\})$—implies existence of an answer set $\mathcal{A}$ for $(P_{\varphi,\Psi}^{is}(G) \setminus P^{gen}) \cup S_k^{\texttt{true}}(0)$. Now $P_{\varphi,\Psi}^{is}(G)$ is obtained by exchanging $S_k^{\texttt{true}}(0)$ with the state generator P^{gen}, which (by reachability of S_k, Definition 15, and Theorem 11) in turn implies existence of an answer set. □

Theorem 18. *Let G be a playable and valid GDL specification whose initial state is S_{init}. Moreover, let Φ and Ψ be sets of sequence invariants over G such that each $\psi \in \Psi$ is satisfied in all reachable states, and let $\varphi \in \Phi$. If every answer set for $P_\Phi^{bc}(G)$ contains $\eta(\varphi)$ and every answer set for $P_{\Phi,\Psi}^{is}(G)$ contains $\eta(\varphi \supset \bigcirc\varphi)$, then for all finite sequences $(S_{init}, S_1, \dots, S_k)$ we have $S_k \vDash \varphi$.*

Proof. The proof is similar to the proof for Theorem 17, with the following additional observations:

- Considering the base case, S_{init} is reachable, hence by Proposition 9 the n-max sequence Seq that violates φ can be extended to an $\hat{n}_{bc}$-max sequence that violates φ.
- Considering the induction step, S_k is reachable, hence by Proposition 9 the $\hat{n}$-max sequence $Seq_{\hat{n}}$ that violates $\varphi \supset \bigcirc\varphi$ can be extended to an $\hat{n}_{is}$-max sequence that violates $\varphi \supset \bigcirc\varphi$.
- The additional encodings $\bigcup_{\rho\in\Phi\setminus\{\varphi\}} Enc(\rho)$ in $P_\Phi^{bc}(G)$ ($\bigcup_{\rho\in\Phi\setminus\{\varphi\}} Enc(\rho \supset \bigcirc\rho)$ in $P_{\Phi,\Psi}^{is}(G)$, respectively) only result in additional unique name atoms in an obtained answer set, without falsifying any other atoms.
- The absence of constraints $:\!-\eta(\varphi)\,.$ in $P_\Phi^{bc}(G)$ (and of $:\!-\eta(\varphi \supset \bigcirc\varphi)\,.$ in $P_{\Phi,\Psi}^{is}(G)$, respectively) does not influence the existence of an answer set $\mathcal{A}$ which is such that $\eta(\varphi) \notin \mathcal{A}$ ($\eta(\varphi \supset \bigcirc\varphi) \notin \mathcal{A}$).

Now, $S_{init} \not\models \varphi$ ($S_k \not\models \varphi \supset \bigcirc\varphi$, respectively) results in an answer set for $P^{bc}_{\Phi}(G)$ ($P^{is}_{\Phi,\Psi}(G)$, respectively) which does not contain $\eta(\varphi)$ ($\eta(\varphi \supset \bigcirc\varphi)$, respectively), which proves the claim. □

Theorem 19. *Consider the same assumptions and naming conventions as in Theorem* 18.

(1) *If* $P^{bc}_{\varphi}(G)$ *is inconsistent then* $\eta(\varphi)$ *is in all answer sets of* $P^{bc}_{\Phi}(G)$.
(2) *If* $P^{is}_{\varphi,\Psi}(G)$ *is inconsistent then* $\eta(\varphi \supset \bigcirc\varphi)$ *is in all answer sets of* $P^{is}_{\Phi,\Psi}(G)$.

Proof.

(1) Let $P_{\hat{n}_{bc}}$ be as P in Theorem 16, replacing S_0 by S_{init} and n by $\hat{n}_{bc}$. Assume that $P^{bc}_{\Phi}(G) = P_{\hat{n}_{bc}} \cup \bigcup_{\rho\in\Phi} Enc(\rho)$ admits an answer set $\mathcal{A}$ such that $\eta(\varphi) \notin \mathcal{A}$. Then there is an $\hat{n}_{bc}$-max sequence $Seq_{\hat{n}_{bc}}$ starting at S_{init} such that $Seq_{\hat{n}_{bc}} \not\models \varphi$ by Theorem 16 and Definition 13. Then for the initial n-max fragment Seq_n of $Seq_{\hat{n}_{bc}}$ we have $Seq_n \not\models \varphi$ by Proposition 9. Thus, again by Theorem 16 and Definition 13, $P_n \cup Enc(\varphi)$ (with P_n as P in Theorem 16, replacing S_0 by S_{init}) admits an answer set $\mathcal{A}'$ such that $\eta(\varphi) \notin \mathcal{A}'$, which is also an answer set for $P^{bc}_{\varphi}(G) = P_n \cup Enc(\varphi) \cup \{:-\eta(\varphi).\}$.
(2) Assume that $P^{is}_{\Phi,\Psi}(G)$ admits an answer set $\mathcal{A}$ such that $\eta(\varphi \supset \bigcirc\varphi) \notin \mathcal{A}$ and let $S_0^{\mathtt{true}}(0) \subseteq \mathcal{A}$ be the set of all atoms of the shape $\mathbf{true}(f, 0)$ in $\mathcal{A}$. Then $(P^{is}_{\Phi,\Psi}(G) \setminus P^{gen}) \cup S_0^{\mathtt{true}}(0)$ admits an answer set $\mathcal{A}'$ such that $\eta(\varphi \supset \bigcirc\varphi) \notin \mathcal{A}'$. The claim now follows by an argumentation similar to (1), where we use S_0 instead of S_{init}, $\hat{n}_{is}$ instead of $\hat{n}_{bc}$, and $\varphi \supset \bigcirc\varphi$ instead of φ. □

References

[1] K. Apt, H.A. Blair, A. Walker, Towards a theory of declarative knowledge, in: J. Minker (Ed.), Foundations of Deductive Databases and Logic Programming. Morgan Kaufmann, 1987, pp. 89–148, Chapter 2.
[2] K. Apt, M. van Emden, Contributions to the theory of logic programming, Journal of the ACM 3 (1982) 841–862.
[3] F. Bacchus, F. Kabanza, Using temporal logic to express search control knowledge for planning, Artificial Intelligence 116 (2000) 123–191.
[4] Y. Björnsson, H. Finnsson, CADIAPLAYER: A simulation-based general game player, IEEE Transactions on Computational Intelligence and AI in Games 1 (2009) 4–15.
[5] K. Clark, Negation as failure, in: H. Gallaire, J. Minker (Eds.), Logic and Data Bases, Plenum Press, 1978, pp. 293–322.
[6] J. Clune, Heuristic evaluation functions for general game playing, in: Proceedings of the AAAI Conference on Artificial Intelligence, AAAI Press, Vancouver, 2007, pp. 1134–1139.
[7] J. Clune, Heuristic evaluation functions for general game playing, Ph.D. thesis, University of California, Los Angeles, 2008.
[8] P. Ferraris, Answer sets for propositional theories, in: Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR), Springer, 2005, pp. 119–131.
[9] H. Finnsson, Y. Björnsson, Learning simulation control in general game-playing agents, in: Proceedings of the AAAI Conference on Artificial Intelligence, AAAI Press, Atlanta, 2010, pp. 954–959.
[10] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, M. Schneider, Potassco: The Potsdam answer set solving collection, AI Communications 24 (2) (2011) 105–124.
[11] A. van Gelder, The alternating fixpoint of logic programs with negation, in: Proceedings of the 8th Symposium on Principles of Database Systems, ACM SIGACT-SIGMOD, 1989, pp. 1–10.
[12] M. Gelfond, Answer sets, in: F. van Harmelen, V. Lifschitz, B. Porter (Eds.), Handbook of Knowledge Representation, Elsevier, 2008, pp. 285–316.
[13] M. Gelfond, V. Lifschitz, The stable model semantics for logic programming, in: R. Kowalski, K. Bowen (Eds.), Proceedings of the International Joint Conference and Symposium on Logic Programming (IJCSLP), MIT Press, Seattle, 1988, pp. 1070–1080.
[14] M. Genesereth, N. Love, B. Pell, General game playing: Overview of the AAAI competition, AI Magazine 26 (2005) 62–72.
[15] E. Giunchiglia, J. Lee, V. Lifschitz, N. McCain, H. Turner, Nonmonotonic causal theories, Artificial Intelligence 153 (2004) 49–104.
[16] S. Iwata, T. Kasai, The Othello game on an n*n board is PSPACE-complete, Theoretical Computer Science 123 (1994) 329–340.
[17] D. Kaiser, The design and implementation of a successful general game playing agent, in: Proceedings of the International Florida Artificial Intelligence Research Society Conference (FLAIRS), AAAI Press, 2007, pp. 110–115.
[18] H. Kautz, B. Selman, Pushing the envelope: Planning, propositional logic, and stochastic search, in: B. Clancey, D. Weld (Eds.), Proceedings of the AAAI Conference on Artificial Intelligence, MIT Press, Portland, 1996, pp. 1194–1201.
[19] M. Kirci, N. Sturtevant, J. Schaeffer, A ggp feature learning algorithm, Künstliche Intelligenz 25 (2011) 35–42.
[20] Z. Kissel, Associative memory and the board game Quarto. Crossroads, The ACM Magazine for Students 10 (2003).
[21] F. Kröger, S. Merz, Temporal Logic and State Systems, Springer, 2008.
[22] G. Kuhlmann, K. Dresner, P. Stone, Automatic heuristic construction in a complete general game player, in: Proceedings of the AAAI Conference on Artificial Intelligence, AAAI Press, Boston, 2006, pp. 1457–1462.
[23] J. Lloyd, Foundations of Logic Programming. Series Symbolic Computation, second, extended edition, Springer, 1987.
[24] J. Lloyd, R. Topor, A basis for deductive database systems II, Journal of Logic Programming 3 (1986) 55–67.
[25] N. Love, T. Hinrichs, D. Haley, E. Schkufza, M. Genesereth, General game playing: Game description language specification, Technical Report LG-2006-01, Stanford Logic Group, Computer Science Department, Stanford University, 353 Serra Mall, Stanford, 2006, CA 94305. Available at: games.stanford.edu.
[26] J. Méhat, T. Cazenave, A parallel general game player, Künstliche Intelligenz 25 (2011) 43–47.
[27] I. Niemelä, P. Simons, T. Soininen, Stable model semantics of weight constraint rules, in: M. Gelfond, N. Leone, G. Pfeifer (Eds.), Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR), Springer, El Paso, 1999, pp. 317–331.
[28] B. Pell, Strategy generation and evaluation for meta-game playing, Ph.D. thesis, Trinity College, University of Cambridge, 1993.
[29] J. Pitrat, Realization of a general game playing program, in: A. Morrell (Ed.), Proceedings of IFIP Congress, Edinburgh, 1968, pp. 1570–1574.
[30] D. Pritchard, The Encyclopedia of Chess Variants, Godalming, 1994.
[31] J. Ruan, M. Thielscher, The epistemic logic behind the game description language, in: Proceedings of the AAAI Conference on Artificial Intelligence, AAAI Press, San Francisco, 2011, pp. 840–845.

[32] J. Ruan, W. van der Hoek, M. Wooldridge, Verification of games in the game description language, Journal of Logic and Computation 19 (2009) 1127–1156.
[33] S. Russell, J. Wolfe, Efficient belief-state AND-OR search with application to kriegspiel, in: L. Kaelbling, A. Saffiotti (Eds.), Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI, Edinburgh, 2005, pp. 278–285.
[34] S. Schiffel, Symmetry detection in general game playing, in: Proceedings of the AAAI Conference on Artificial Intelligence, AAAI Press, Atlanta, 2010, pp. 980–985.
[35] S. Schiffel, M. Thielscher, Fluxplayer: A successful general game player, in: Proceedings of the AAAI Conference on Artificial Intelligence, AAAI Press, Vancouver, 2007, pp. 1191–1196.
[36] S. Schiffel, M. Thielscher, Reasoning about general games described in GDL-II, in: Proceedings of the AAAI Conference on Artificial Intelligence, AAAI Press, San Francisco, 2011, pp. 846–851.
[37] M. Thielscher, A general game description language for incomplete information games, in: Proceedings of the AAAI Conference on Artificial Intelligence, AAAI Press, Atlanta, 2010, pp. 994–999.
[38] M. Thielscher, The general game playing description language is universal, in: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), AAAI Press, Barcelona, 2011, pp. 1107–1112.