Simulation of Action Theories and an Application to General Game-Playing Robots

Michael Thielscher

School of Computer Science and Engineering
The University of New South Wales
Sydney, NSW 2052, Australia
mit@cse.unsw.edu.au

Abstract. We consider the problem of verifying whether one action theory can *simulate* a second one. Action theories provide modular descriptions of state machines, and simulation means that all possible sequences of actions in one transition system can be matched by the other. We show how Answer Set Programming can be used to automatically prove simulation by induction from an axiomatisation of two action theories and a projection function between them. Our interest in simulation of action theories comes from general game-playing robots as systems that can understand the rules of new games and learn to play them effectively in a physical environment. A crucial property of such games is their *playability*, that is, each legal play sequence in the abstract game must be executable in the real environment.

1 Introduction

Simulation, and bisimulation, of state transition systems is an important and well researched concept in theoretical computer science and formal logic [30, 3] but has not been applied in the context of action languages that provide logic-based, compact descriptions of state machines [5, 14, 29, 32]. We consider the problem of automatically proving whether the transition system represented by one action theory can simulate the system described by another theory.

Our interest in simulation of action theories comes from an open problem in *general game-playing robotics*, which is concerned with the design of autonomous systems that can understand descriptions of new games and learn to play them in a physical game environment [28]. This is an attempt to create a new generation of AI systems that can understand the rules of new games and then learn to play these games without human intervention [16]. Unlike specialised game-playing systems such as the chess program Deep Blue [19], a general game player cannot rely on algorithms that have been designed in advance for specific games. Rather, it requires a form of general intelligence that enables the player to autonomously adapt to new and possibly radically different problems. General game playing programs therefore are a quintessential example of a new generation of systems that end users can customise for their own specific tasks and special needs [15],

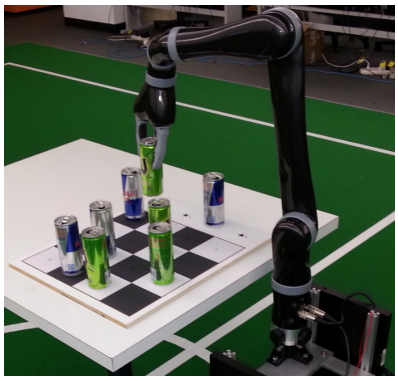

Fig. 1. A physical game environment.

and general game-playing robots extend this capability to AI systems that play games in the real world [28].

In general game playing, games are represented using a special-purpose action description language [16]. These game descriptions must satisfy a few basic requirements to ensure that a game is effectively playable; for example, there should always be at least one legal move in every nonterminal position [17]. In bringing gameplay from mere virtual into physical environments, general game-playing robots require an additional property that concern the manifestation of the game rules in the real world. Notably, a suitable game description requires all moves deemed legal by the rules of the abstract game to be executable in the real world [28].

As an example, consider the robotic environment shown in Fig. 1. It features a 4×4 chess-like board with an additional row of 4 marked positions on the right. Tin cans are the only type of objects and can be moved between the marked position (but cannot be stacked). This game environment can be interpreted in countless ways as physical manifestations of a game, including all kinds of mini chess-like games but also, say, single-player games like the 15-puzzle, where individual cans represent numbered tiles that need to be brought in the right order [28]. In fact, any abstract game is *playable* in this environment provided that all legal play sequences can be executed by the robot.

In order to prove the playability of a game, we consider its rules and those that govern the robotic environment as formal descriptions of state transition systems. This allows us to reduce the problem of verifying that a game is playable to the problem of proving that an action theory describing the environment can simulate the action theory that encodes the game. As a general technique, we will show how Answer Set Programming can be used to automatically prove the simulation of action theories based on their axiomatisation along with a projection function between the states of the two systems.

The remainder of the paper is organised as follows. Section 2 introduces a basic action description language that we will use for our analysis and which derives from the general game description language GDL [16]. In Section 3, we formally define the concept of simulation for action theories. The use of Answer Set Programming to automatically prove this property by induction are given in Section 4, and in Section 5 we show how the result can be applied to proving the playability of abstract games in physical environments. We conclude in Section 6.

2 Action Theories

A variety of knowledge representation languages exist for describing actions and change, including first-order formalisms such as the classical Situation Calculus and its variants [26, 20, 33], special-purpose action description languages [14, 29], planning formalisms [8, 11] or the general game description language [16]. While they are all subtly different, action languages invariably share the following standard elements:

- *fluents*, which describe atomic, variable properties of states;
- *actions*, whose execution triggers state transitions;
- *action preconditions*, defining conditions on states for an action to be executable;
- *effect specifications*, defining the result of actions;
- *initial state* description.

For the purpose of this paper, we will use a simple and generic specification language for basic action theories that uses Answer Set Programming (ASP) syntax to describe all of these basic elements. Many of the aforementioned action formalisms have straightforward translations into ASP, e.g. [21, 6, 2, 12, 34]. Hence, while our language borrows its five pre-defined predicates from the game description language GDL [16] in view of our motivating application, our definitions and results can be easily adapted to similar action representation formalisms.

Example. Before providing the formal language definition, let us consider the example of a 4×4 sliding puzzle, which is formally described by the action theory given in Fig. 2. The rules use the fluent $cell(x, y, z)$ to indicate the current state of position (x, y) as either occupied by tile z or being empty, where $x, y \in \{1, \ldots, 4\}$ and $z \in \{1, \ldots, 15, empty\}$. A second fluent $step(x)$ counts the number of moves, which has been limited to $x \in \{1, \ldots, 80\}$. The only action in this domain is $move(u, v, x, y)$, denoting the move of sliding the tile in (u, v) into position (x, y), where $u, v, x, y \in \{1, \ldots, 4\}$. Intuitively, the description can be understood as follows:

- Facts 1–17 completely describe the *initial state* as depicted.
- The *precondition axioms* 19–22 say that a tile can be slid into the adjacent empty cell.
- The *result* of sliding the tile in (u, v) into position (x, y) is that

```
1  init(cell(1,1, 9)).
2  init(cell(2,1, 2)).
3  init(cell(3,1, 8)).
4  init(cell(4,1,12)).
5  init(cell(1,2,11)).
6  init(cell(2,2, 3)).
7  init(cell(3,2,15)).
8  init(cell(4,2,10)).
9  init(cell(1,3, 6)).
10 init(cell(2,3,empty)).
11 init(cell(3,3,13)).
12 init(cell(4,3, 5)).
13 init(cell(1,4,14)).
14 init(cell(2,4, 4)).
15 init(cell(3,4, 1)).
16 init(cell(4,4, 7)).
17 init(step(1)).
18
19 legal(move(U,Y,X,Y)) :- true(cell(X,Y,empty)), succ(U,X), true(cell(U,Y,Z)).
20 legal(move(U,Y,X,Y)) :- true(cell(X,Y,empty)), succ(X,U), true(cell(U,Y,Z)).
21 legal(move(X,V,X,Y)) :- true(cell(X,Y,empty)), succ(V,Y), true(cell(X,V,Z)).
22 legal(move(X,V,X,Y)) :- true(cell(X,Y,empty)), succ(Y,V), true(cell(X,V,Z)).
23
24 next(cell(U,V,empty)) :- does(move(U,V,X,Y)).
25 next(cell(X,Y,Z))     :- does(move(U,V,X,Y)), true(cell(U,V,Z)).
26
27 next(cell(R,S,Z)) :- true(cell(R,S,Z)), does(move(U,V,X,Y)), R != U, R != X.
28 next(cell(R,S,Z)) :- true(cell(R,S,Z)), does(move(U,V,X,Y)), R != U, S != Y.
29 next(cell(R,S,Z)) :- true(cell(R,S,Z)), does(move(U,V,X,Y)), S != V, R != X.
30 next(cell(R,S,Z)) :- true(cell(R,S,Z)), does(move(U,V,X,Y)), S != V, S != Y.
31
32 next(step(Y)) :- true(step(X)), succ(X,Y).
33
34 succ(1,2).   succ(2,3).  ...  succ(79,80).
```

	1	2	3	4	
	9	2	8	12	1
	11	3	15	10	2
	6		13	5	3
	14	4	1	7	4

Fig. 2. The 15-puzzle described by an action theory.

- cell (u, v) becomes empty while the tile is now in (x, y) (clauses 24, 25);
- all other cells retain their tiles (clauses 27–30);
- the step counter is incremented (clause 25).

As can be seen from this example, our action theory uses the following unary predicates as pre-defined keywords:

- `init`(f), to define fluent f to be true initially;
- `true`(f), denoting the condition that f is true in a state;
- `does`(a), denoting the condition that a is performed in a state;
- `legal`(a), meaning that action a is possible;
- `next`(f), to define the fluents that are true after an action is performed.

For the formal definition of the syntax of the action specification language, we assume that the reader is familiar with basic concepts of logic programs [23] and Answer Set Programming [4]. Our action theories are normal logic programs that have to satisfy a few syntactic restrictions borrowed from GDL [17] in order to ensure that they admit a unique and finite interpretation.

Definition 1. *Consider an alphabet that includes the unary predicates* `init`, `legal`, `next`, `true` *and* `does`. *An* action theory *is a normal logic program* P *such that*

1. P *is* stratified, *that is, its dependency graph has no cycles with a negative edge [1];*
2. P *is* allowed, *that is, each variable in a clause occurs in a positive atom of the body of that clause [24];*
3. P *satisfies the following restrictions on the pre-defined predicates:*
   (a) `init` *occurs only in the head of clauses and does not depend on any of the other special keywords;*
   (b) `legal` *occurs only in the head of clauses and does not depend on* `does`*;*
   (c) `next` *occurs only in the head of clauses;*
   (d) `true` *and* `does` *occur only in the body of clauses.*
4. P *obeys the following* recursion restriction *to ensure finite groundings: If predicates* p *and* q *occur in a cycle in the dependency graph of* P, *or if* $p = \texttt{true}$ *and* $q = \texttt{next}$, *and* P *contains a clause*

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

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* $\boldsymbol{t}_j$ $(1 \leq j \leq n)$ *such that* b_j *does not occur in a cycle with* p *in the dependency graph of* P.

It is straightforward to verify that the action theory in Fig. 2 satisfies this definition of a proper action theory.

3 Simulation of Action Theories

The concept of simulation for action theories needs to be defined on the state transition systems that they describe, where generally states are identified by the fluents that hold and state transitions are triggered by actions [14, 29, 32]. In case of the action description language of Definition 1, this interpretation is obtained with the help of the stable models [13] of Answer Set Programs. Below, $\texttt{SM}[P]$ denotes the unique stable model of a stratified, finitely groundable program P.

Definition 2. *Let* P *be an action theory in the language of Definition 1 with ground fluents* $\mathcal{F}$ *and ground actions* $\mathcal{A}$. P *determines a* finite state machine $(\mathcal{A}, S, s_0, \delta)$ *as follows:*

1. $S = 2^{\mathcal{F}}$ *are the* states*;*
2. $s_0 = \{f \in \mathcal{F} : \texttt{init}(f) \in \texttt{SM}[P]\}$ *is the* initial state*;*
3. $\delta(a, s) = \{f \in \mathcal{F} : \texttt{next}(f) \in \texttt{SM}[P \cup \texttt{does}(a) \cup \texttt{true}|_s]\}$ *is the* transition function*, where*
   – $a \in \mathcal{A}$

- $s \in S$
- $\texttt{true}|_s = \{\texttt{true}(f) : f \in s\}$
- $\texttt{legal}(a) \in \texttt{SM}[P \cup \texttt{true}|_s]$ *(that is,* a *is* possible *in* s*).*

A state $s \in S$ *is called* reachable *if there is a finite sequence of actions* $a_1, \ldots, a_k$ *such that* $s = \delta(a_k, \ldots, \delta(a_1, s_0) \ldots)$.

Put in words,

- states are sets of ground fluents;
- the initial state is given by all derivable instances of $\texttt{init}(f)$;
- to determine if an action is legal in a state s, this state s has to be encoded using facts $\texttt{true}(f)$, and then a is possible if $\texttt{legal}(a)$ can be derived;
- likewise, to determine the effects of an action a in a state s, the action and the state have to be encoded using facts $\texttt{does}(a)$ and $\texttt{true}(f)$, respectively, and then the resulting state is given by all derivable instances of $\texttt{next}(f)$.

Example. Recall the action theory in Fig. 2 describing the 15-puzzle. It is easy to see that the initial state is

$$s_0 = \{cell(1,1,9), \ldots, cell(1,3,6), cell(2,3,empty), \ldots, step(1)\} \qquad (1)$$

It is straightforward to verify that the action $move(1,3,2,3)$ is possible in this state: After adding each of the facts in $\texttt{true}|_{s_0}$, the unique stable model of the resulting program includes $\texttt{true}(cell(2,3,empty))$, $\texttt{true}(cell(1,3,6))$ and $succ(1,2)$, hence also $\texttt{legal}(move(1,3,2,3))$ according to clause 19. From Definition 2 and the clauses 24–32 it follows that

$$\delta(move(1,3,2,3), s_0) = \{cell(1,1,9), \ldots, cell(1,3,empty), cell(2,3,6), \ldots, step(2)\}$$

Given two state transition systems, the standard definition of a simulation requires that one matches all actions in the other. In case of two action theories P_1 and P_2, this requires that the actions and states of the simulated domain, P_1, can be projected onto actions and states in the simulating domain, P_2, such that

- the initial state of P_1 projects onto the initial state of P_2;
- if an action is possible in P_2, then the corresponding action is possible in the corresponding state in P_2 and the resulting states correspond, too.

This is formally captured by the following definition.

Definition 3. *Let* P_1 *and* P_2 *be two action theories, which describe finite state machines* $(\mathcal{A}, S, s_0, \delta)$ *and* $(\mathcal{B}, T, t_0, \varepsilon)$*, respectively. A* projection *of* P_1 *onto* P_2 *is a function* π *such that*

- $\pi(a) \in \mathcal{B}$ *for all* $a \in \mathcal{A}$
- $\pi(s) \in T$ *for all* $s \in S$

```
1  init(piece(a,1)).
2  init(piece(b,1)).
3  init(piece(c,1)).
4  init(piece(d,1)).
5  init(piece(a,2)).
6  init(piece(c,2)).
7  init(piece(d,2)).
8  init(piece(a,3)).
9  init(piece(b,3)).
10 init(piece(c,3)).
11 init(piece(d,3)).
12 init(piece(a,4)).
13 init(piece(b,4)).
14 init(piece(c,4)).
15 init(piece(d,4)).
16
17 legal(put(U,V,X,Y)) :- true(piece(U,V)), coord(X,Y), not true(piece(X,Y)).
18
19 next(piece(X,Y)) :- does(put(U,V,X,Y)).
20 next(piece(X,Y)) :- true(piece(X,Y)), not moved(X,Y).
21
22 moved(X,Y) :- does(put(X,Y,U,V)).
23 coord(a,1).  coord(a,2).  coord(a,3).  coord(a,4).
24 ...
25 coord(x,1).  coord(x,2).  coord(x,3).  coord(x,4).
```

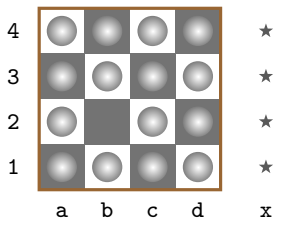

Fig. 3. An action theory describing the physical environment of the robot in Fig. 1.

A projection π *is a* simulation *of* P_1 *by* P_2 *if*

1. $\pi(s_0) = t_0$ *and*
2. *for all* $a \in \mathcal{A}$ *and all reachable* $s \in S$,
   (a) *if* a *is possible in* s *then* $\pi(a)$ *is possible in* $\pi(s)$
   (b) $\pi(\delta(a, s)) = \varepsilon(\pi(a), \pi(s))$

Example. The action theory in Fig. 3 describes the physical environment of the robot in Fig. 1 with the help of a single fluent, $piece(i, j)$, indicating whether a can has been placed at (i, j) where $i \in \{a, b, c, x\}$ and $j \in \{1, 2, 3\}$; and the action $put(i, j, k, l)$ of lifting the object at location (i, j) and putting it down at location (k, l).

The following projection function maps every action and state in the 15-puzzle to an action and state in the robotic game environment:

1. $\pi(move(u, v, x, y)) = put(\overline{u}, 5 - v, \overline{x}, 5 - y)$,
   where $\overline{1} = a, \ldots, \overline{4} = d$ (to account for the different coordinate systems);
2. $\pi(s) = \{piece(\overline{x}, 5 - y) : cell(x, y, z) \in s,\ z \neq empty\}$.

It is easy to see that under this function, initial state (1) of the 15-puzzle projects onto the initial state of the action theory for the robotic environment. Indeed, the projection provides a simulation of the 15-puzzle in the physical robot domain: According to the rules in Fig. 2, the possible actions in the 15-puzzle are to move from a cell (u, v) to an adjacent and empty cell (x, y). This implies that there is no piece in the corresponding location $(\overline{x}, 5 - y)$ on the physical board and also

that there is a piece at $(\overline{u}, 5 - v)$ since there can be no more than one empty cell in any reachable state of the game. Hence, the corresponding *put* action in the robotic environment is possible in the projected state according to clause 17 in Fig. 3. Moreover, the result of sliding a tile is that the tile and the empty location swap places, which corresponds to the effect of moving the respective tin can.

It is worth noting that the reverse does not hold: The robot can of course move any of the pieces into a non-adjacent, empty location, including the 4 marked positions to the right of the board. None of these actions can be matched by a legal move in the 15-puzzle.

4 Automating Simulation Proofs

An automated proof that one action theory indeed simulates a second one through a given projection, in general needs to inspect all action sequences possible in the simulated transition system. A viable and sound but incomplete alternative is to use induction proofs—a technique that has been successfully applied to automatically proving state constraints in action theories [18, 22]. Indeed, the required properties of a simulation according to Definition 3 can be considered as state constraints over the combined action theories. In the following, we adapt the existing ASP-based proof technique for state constraints in general games [18] to solve the problem of automatically proving simulation of two action theories by induction.

Consider two action theories P_1 and P_2. We combine these into a single answer set program $P_1 \cup P_2$, which is then augmented as follows:

1. An encoding of a given projection function π from P_1 to P_2 by:
   (a) Clauses

   ```
   1 isimulation_error :- ¬Π[init].
   2 tsimulation_error :- ¬Π[true].
   3 nsimulation_error :- ¬Π[next].
   ```

   Here, Π stands for an ASP encoding of the conditions (on the fluents in the two action theories) under which a state from P_1 projects onto a state from P_2 according to π. The expression Π[`init`] etc. means to replace every occurrence of a fluent f in Π by `init`(f) etc.[1]
   (b) Clauses

   ```
   4 does(π(a)) :- does(a).
   ```

   for actions a from P_1.
2. An encoding of the induction hypothesis as

[1] For example, if the projection function requires that there be no empty cell (x, y) in the abstract game that houses a piece in the physical environment, then $\neg\Pi$ could be $(\exists x, y)\, cell(x, y, empty) \wedge piece(x, y)$, in which case $\neg\Pi$[`init`] is $(\exists x, y)$ `init`$(cell(x, y, empty)) \wedge$ `init`$(piece(x, y))$.

```
5  { true(F) : fluent(F) }.
6  1 { does(A) : action(A) } 1.
7  :- action(A), does(A), not legal(A).
8  :- tsimulation_error.
```

where the auxiliary predicate `fluent` ranges over all fluents in either P_1 or P_2 while `action` ranges over the actions of P_1 only.

3. The negation of the base case and of the induction step as

```
9  counterexample :- isimulation_error.
10 counterexample :- does(A), not legal(A).
11 counterexample :- nsimulation_error.
12 :- not counterexample.
```

If the resulting ASP admits *no* stable models, then this proves the projection function to be a simulation of P_1 by P_2: Clause 12 excludes solutions without a counter-example, which according to clauses 9–11 is obtained when

1. the initial state does not project, which corresponds to condition 1 in Definition 3;
2. an action exists (clause 6) that is legal (clause 7) but whose projection is not possible, which corresponds to condition 2(a) in Definition 3;
3. a state, i.e. a set of fluents, exists (clause 5) so that the result of a state transition does not project, which corresponds to condition 2(b) in Definition 3.

We have thus obtained at a technique for automating simulation proofs that is correct and also very viable in practice as it avoids inspecting all possible action sequences of the simulated state transition system, as a variety of systematic experiments with similar inductive proof techniques have demonstrated in the past [18].

While sound, these induction proofs are in general incomplete as can be shown with our two example action theories for the 15-puzzle and the robotic environment as given in Fig. 2 and 3, respectively.

Example. Using the same schema as the generic clauses 1–4 above, the projection function defined for our example in Section 3 can be encoded thus (where for the sake of clarity we assume that the two coordinate systems were identical):

```
does(put(U,V,X,Y)) :- does(move(U,V,X,Y)).

isimulation_error :- init(piece(X,Y)), not icell_tile(X,Y).
isimulation_error :- not init(piece(X,Y)), icell_tile(X,Y).
tsimulation_error :- true(piece(X,Y)), not tcell_tile(X,Y).
tsimulation_error :- not true(piece(X,Y)), tcell_tile(X,Y).
nsimulation_error :- next(piece(X,Y)), not ncell_tile(X,Y).
nsimulation_error :- not next(piece(X,Y)), ncell_tile(X,Y).

icell_tile(X,Y) :- init(cell(X,Y,Z)), Z != empty.
tcell_tile(X,Y) :- true(cell(X,Y,Z)), Z != empty.
ncell_tile(X,Y) :- next(cell(X,Y,Z)), Z != empty.
```

Put in words, a projected state requires a tin can at location (x, y) if, and only if, the corresponding cell in the 15-puzzle exists and is not empty. Combined with the action theories of Fig. 2 and 3 and augmented by the general clauses 5–12 from above, the resulting ASP *does* admit stable models. For instance, one model of the ASP includes

$$\texttt{true}(\mathit{cell}(1,1,\mathit{empty})),\ \texttt{true}(\mathit{cell}(1,2,\mathit{empty})),\ \texttt{legal}(\mathit{move}(1,1,1,2))$$

Indeed, the action theory in Fig. 2 sanctions the move from one empty cell to a neighbouring empty cell while this is not possible in the robot domain, where only pieces can be moved. Another model of the ASP includes

$$\texttt{true}(\mathit{cell}(1,1,1)),\ \texttt{true}(\mathit{cell}(1,2,\mathit{empty})),\ \texttt{legal}(\mathit{move}(1,1,1,2))$$
$$\texttt{true}(\mathit{cell}(1,2,2)),\ \texttt{true}(\mathit{piece}(1,2))$$

Indeed, the action theory in Fig. 2 sanctions the move into a cell with a numbered tile, here $(1, 2)$, if the fluent is also true that says that this cell is empty. Again this is not possible in the robot domain, where a tin can cannot be put down at a location already occupied by an object.

Clearly, both these generated counter-examples refer to unreachable states in the 15-puzzle, hence their existence does not disprove our projection to provide a simulation of this game by the robot. In fact, we can enhance the capability of any ASP for proving simulation by adding state constraints of the simulated action theory that help to exclude unreachable states from being considered as counter-examples. Specifically, the 15-puzzle satisfies these state constraints:

```
inconsistent :-
   true(cell(U,V,empty)), true(cell(X,Y,empty)), U != X.
inconsistent :-
   true(cell(U,V,empty)), true(cell(X,Y,empty)), V != Y.
inconsistent :-
   true(cell(X,Y,empty)), true(cell(X,Y,Z)), Z != empty.
:- inconsistent.
```

Put in words, no consistent state contains two different cells that are both empty, or a cell that is both empty and occupied by a numbered tile. These constraints themselves can be automatically proved from the underlying action theory of Fig. 2 using existing methods [18, 22]. Once they are added, the ASP for proving that the robotic domain can simulate the 15-puzzle admits no stable model, which establishes the intended result.

5 General Game-Playing Robots and the Playability of Games

The annual AAAI GGP Competition [16] defines a general game player as a system that understands the formal Game Description Language (GDL) [25] and is able to play effectively any game described therein. Since the first contest in

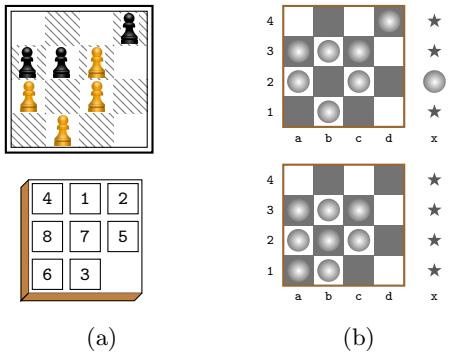

Fig. 4. (a) Different games and (b) their projections onto the game environment of Fig. 1.

2005, General Game Playing has evolved into a thriving AI research area. Established methods include Monte Carlo tree search [9], the automatic generation of heuristic evaluation functions [7, 31], and learning [10].

General game-playing robots extend this capability to AI systems that play games in the real world [28]. In bringing gameplay from mere virtual into physical environments, this adds a new requirement for suitable game descriptions, which concerns the manifestation of the game rules in the real world: An abstract game described in GDL can be played in a real robotic environment only if all moves deemed legal by the rules are actually possible in the physical world.

When we use a physical environment to play a game, the real objects become representatives of entities in the abstract game. A pawn in chess, for instance, is typically manifested by an actual wooden piece of a certain shape and colour. But any other physical object, including a tin can, can serve the same purpose. Conversely, any game environment like the $4 \times 4(+4)$ board with cans depicted in Fig. 1 can be interpreted in countless ways as physical manifestation of a game. For example, Fig. 4(a) shows two positions from two different games, a mini chess-like game and the 8-puzzle as a smaller variant of the standard sliding tile game. We can view the states depicted to the left of each position (Fig. 4(b)) as their projection onto our example physical game environment, in which the extra row can be used to park captured pieces (in chess-like games) or where gameplay is confined to a subregion of the board (for the 8-puzzle). Note that this manifestation abstracts away possible differences in the type of cans such as their colour or shape (or contents for that matter). Hence, it is only through a projection function that the robotic player knows whether a can stands for a white or a black pawn, say. The same holds for the sliding puzzles, where the goal position (with all tiles in ascending order) actually projects onto the very same abstract environment state as the starting position—the distinction lies entirely in the meaning attached to individual cans in regard to which number they represent. It is noteworthy that a similar feature is found in many games

humans play, where also the physical manifestation of a game position is often an incomplete representation; for example, the pieces on a chessboard alone are not telling us whose move it is or which side still has castling rights [27].

The manifestation of a game in a physical environment can be mathematically captured by *projecting* the positions from the abstract game onto actual states of the game environment, and then a game is *playable* if all actions in the abstract game can be matched by actions in the projected environment [28]. The language GDL, which is commonly used to describe games in general game playing and which supports the description of any finite n-player game ($n \geq 1$), includes elements for the specifications of different players and goals [16]. Since these are irrelevant for the question whether a game is playable and because our action language of Definition 1 is in fact a stripped-down version of GDL, the formal concept of simulation of action theories, along with our proof technique, can be employed for the purpose of automatically proving that a game is playable in a robotic environment. The only requirement is to symbolically describe the latter by an action theory in the same language.

6 Conclusion

In this paper we have defined the concept of one action theory being able to simulate a second one. We have shown how Answer Set Programming can be used to automatically prove simulation by induction from an axiomatisation of the two action theories and a projection function between them. We have motivated and applied these results in the context of systems that draw together two topical yet disparate areas of artificial intelligence research: general game playing and robotics.

Our definition of simulation in action theories follows the standard one in theoretical computer science and formal logic, in that actions always need to be matched by single actions. In practice, this requires a similar level of abstraction in both models. But our notion of projection in Definition 3 can be straightforwardly generalised to allow for different degrees of abstraction in that an action in one model corresponds to a *sequence* of actions in the other one. A single move in the abstract 15-puzzle, for example, could then be mapped onto a complex movement of the robot arm in the physical environment: move above the can, open the fingers, go down, close the fingers, move to the target location, open the fingers, raise above the can, close the fingers and return back to the home position. The automation of simulation proofs then needs to be suitably extended by incorporating sequences of state updates in one action theory [18] and aligning them with a single state transition in the simulated system.

7 Acknowledgement

This article, like so many before (and hopefully after!), may very well have never been written had I not met Gerhard Brewka when I was still an undergraduate at Wolfgang Bibel's research group in Darmstadt. I very well remember that first

research project meeting I ever attended, where I gave the first scientific talk in my life, hopelessly nervous of course, but then to my greatest surprise caught the attention of the most prominent scientist present. His words of encouragement, after a lot of sweat, tears and the journey of writing, eventually led to my first article in a journal, and a decent one at that, marking the beginning of my life as a scientist. Gerd may not remember, but over the years we have collaborated in research projects, co-authored papers and jointly supervised students, and a lifelong friendship ensued.

References

1. Apt, K., Blair, H., Walker, A.: Towards a theory of declarative knowledge. In: Minker, J. (ed.) Foundations of Deductive Databases and Logic Programming, chap. 2, pp. 89–148. Morgan Kaufmann (1987)
2. Babb, J., Lee, J.: Cplus2ASP: Computing action language $\mathcal{C}+$ in answer set programming. In: Cabalar, P., Son, T. (eds.) Proceedings of the International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR). LNCS, vol. 8148, pp. 122–134. Springer, Corunna, Spain (Sep 2013)
3. van Benthem, J.: Logic in Games. MIT Press (2014)
4. Brewka, G., Eiter, T., Truszczynski, M.: Answer set programming at a glance. Communications of the ACM 54(12), 92–103 (2011)
5. Brewka, G., Hertzberg, J.: How to do things with worlds: on formalizing actions and plans. Journal of Logic and Computation 3(5), 517–532 (1993)
6. Cerexhe, T., Gebser, M., Thielscher, M.: Online agent logic programming with oClingo. In: Proceedings of the Pacific Rim International Conference on Artificial Intelligence. LNCS, Springer, Gold Coast (Dec 2014)
7. Clune, J.: Heuristic evaluation functions for general game playing. In: Proceedings of the AAAI Conference on Artificial Intelligence. pp. 1134–1139. AAAI Press, Vancouver (Jul 2007)
8. Fikes, R., Nilsson, N.: STRIPS: A new approach to the application of theorem proving to problem solving. Artificial Intelligence 2, 189–208 (1971)
9. Finnsson, H., Björnsson, Y.: Simulation-based approach to general game playing. In: Proceedings of the AAAI Conference on Artificial Intelligence. pp. 259–264. AAAI Press, Chicago (Jul 2008)
10. Finnsson, H., Björnsson, Y.: Learning simulation control in general game-playing agents. In: Proceedings of the AAAI Conference on Artificial Intelligence. pp. 954–959. AAAI Press, Atlanta (Jul 2010)
11. Fox, M., Long, D.: PDDL2.1: an extension to PDDL for expressing temporal planning domains. Journal of Artificial Intelligence Research 20, 61–124 (2003)
12. Gebser, M., Kaminski, R., Knecht, M., Schaub, T.: plasp: A prototype for PDDL-based planning in ASP. In: Delgrande, J., Faber, W. (eds.) Proceedings of the International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR). LNCS, vol. 8148, pp. 358–363. Springer, Vancouver, Canada (May 2011)
13. Gelfond, M.: Answer sets. In: van Harmelen, F., Lifschitz, V., Porter, B. (eds.) Handbook of Knowledge Representation. pp. 285–316. Elsevier (2008)
14. Gelfond, M., Lifschitz, V.: Representing action and change by logic programs. Journal of Logic Programming 17, 301–321 (1993)
15. Genesereth, M., Björnsson, Y.: The international general game playing competition. AI Magazine 34(2), 107–111 (2013)

16. Genesereth, M., Love, N., Pell, B.: General game playing: Overview of the AAAI competition. AI Magazine 26(2), 62–72 (2005)
17. Genesereth, M., Thielscher, M.: General Game Playing. Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan & Claypool (2014)
18. Haufe, S., Schiffel, S., Thielscher, M.: Automated verification of state sequence invariants in general game playing. Artificial Intelligence 187–188, 1–30 (2012)
19. Hsu, F.H.: Behind Deep Blue: Building the Computer that Defeated the World Chess Champion. Princeton University Press (2002)
20. Kowalski, R.: Database updates in the event calculus. Journal of Logic Programming 12, 121–146 (1992)
21. Lee, J.: Reformulating the situation calculus and the event calculus in the general theory of stable models and in answer set programming. Journal of Artificial Intelligence Research 43, 571–620 (2012)
22. Li, N., Fan, Y., Liu, Y.: Reasoning about state constraints in the situation calculus. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI). Beijing, China (Aug 2013)
23. Lloyd, J.: Foundations of Logic Programming. Series Symbolic Computation, Springer, second, extended edn. (1987)
24. Lloyd, J., Topor, R.: A basis for deductive database systems II. Journal of Logic Programming 3(1), 55–67 (1986)
25. Love, N., Hinrichs, T., Haley, D., Schkufza, E., Genesereth, M.: General Game Playing: Game Description Language Specification. Tech. Rep. LG–2006–01, Stanford Logic Group, Computer Science Department, Stanford University, 353 Serra Mall, Stanford, CA 94305 (2006), available at: `games.stanford.edu`
26. McCarthy, J.: Situations and Actions and Causal Laws. Stanford Artificial Intelligence Project, Memo 2, Stanford University, CA (1963)
27. Pritchard, D.: The Encycolpedia of Chess Variants. Godalming (1994)
28. Rajaratnam, D., Thielscher, M.: Towards general game-playing robots: Models, architecture and game controller. In: Cranefield, S., Nayak, A. (eds.) Proceedings of the Australasian Joint Conference on Artificial Intelligence. LNCS, vol. 8272, pp. 271–276. Springer, Dunedin, New Zealand (Dec 2013)
29. Sandewall, E.: Features and Fluents. The Representation of Knowledge about Dynamical Systems. Oxford University Press (1994)
30. Sangiorgi, D.: Introduction to Bisumlation and Coinduction. Cambridge University Press (2011)
31. Schiffel, S., Thielscher, M.: Fluxplayer: A successful general game player. In: Proceedings of the AAAI Conference on Artificial Intelligence. pp. 1191–1196. AAAI Press, Vancouver (Jul 2007)
32. Schiffel, S., Thielscher, M.: A multiagent semantics for the Game Description Language. In: Filipe, J., Fred, A., Sharp, B. (eds.) Agents and Artificial Intelligence. Communications in Computer and Information Science, vol. 67, pp. 44–55. Springer (2010)
33. Thielscher, M.: From situation calculus to fluent calculus: State update axioms as a solution to the inferential frame problem. Artificial Intelligence 111(1–2), 277–299 (1999)
34. Thielscher, M.: Answer set programming for single-player games in general game playing. In: Hill, P., Warren, D. (eds.) Proceedings of the International Conference on Logic Programming (ICLP). LNCS, vol. 5649, pp. 327–341. Springer, Pasadena (Jul 2009)