Translating Single-Player GDL into PDDL

Thorsten Rauber, Peter Kissmann, and Jörg Hoffmann

Saarland University
Saarbrücken, Germany
s9thraub@stud.uni-saarland.de, {kissmann, hoffmann}@cs.uni-saarland.de

Abstract. In the single-agent case general game playing and action planning are two related topics, so that one might hope to use the established planners to improve the handling of general single-player games. However, both come with their own description language, GDL and PDDL, respectively. In this paper we propose a way to translate single-player games described in GDL to PDDL planning tasks and provide an evaluation on a wide range of single-player games, comparing the efficiency of grounding and solving the games in the translated and in the original format.

1 Introduction

The current form of general game playing (GGP) was developed around 2005, when the first international GGP competition was held [3]. It aims at developing game playing agents that can handle and efficiently play any game that is describable using the game description language GDL [9]. In its basic form a wide range of games can be modeled, though there are severe restrictions, namely to deterministic games of full information, which immediately rule out most dice and card games. In recent years the most successful GGP agents such as CadiaPlayer [1] or Ary [11] made use of the UCT algorithm [8], which often delivers good results in multi-player games but lacks in single-player settings.

That is where action planning comes in. Similar to GGP, action planning aims at developing agents that can handle any planning task that is describable by the input language, which, since 1998, is the planning domain definition language PDDL [10]. There is a close connection between general single-player games and planning. In both cases the agent will be confronted with problems for which it has to find a solution (i.e., a sequence of moves or actions) leading from a specified initial state to a state satisfying a goal condition. While in planning the solution length is often relevant, in GGP each goal state is assigned some reward and the aim is to maximize the achieved reward. Finding an optimal solution reduces to finding a solution that results in a state satisfying a goal achieving the highest possible reward. As the programmer does not know in advance the problems that the agent will be confronted with, in both settings domain-independent approaches are required.

Even though the setting is similar, there are severe differences in the input languages. If we want to make use of the existing efficient planners (e.g., Fast-Forward [6], FastDownward [5], or LAMA [13]) to improve the handling of single-player games we need to provide a translation from GDL to PDDL, which is the topic of this paper.

The remainder of this paper is structured as follows. We begin with a description of the two input languages PDDL and GDL (Section 2), then turn to the translation from GDL to PDDL (Section 3) and show some empirical results illustrating that the translation works well and that planners can handle the generated input, sometimes even outperforming the GGP agent we took as reference (Section 4). Finally, we discuss related work and draw some conclusions (Section 5).

2 Background

2.1 Action Planning

A planning task is a tuple $\Pi = \langle V, A, I, G \rangle$, where V are the state variables (we do not handle finite domains here, so we assume all variables to be binary), A is a finite set of actions a each of which is a pair $\langle \mathsf{pre}_a, \mathsf{eff}_a \rangle$ of partial assignments to V with pre_a being the precondition and eff_a the effect of action a, the initial state I is a complete assignment to V, and the goal is a partial assignment to V.

In our translation we also make use of conditional effects, which are nested effects with their own precondition cpre and effect ceff. The effect of a conditional effect is only evaluated if its precondition holds in the current state. Note that nesting conditional effects is not possible.

Derived predicates as introduced in [2] are another important feature that we will use. The idea is to infer their truth from the truth of the state variables. So derived predicates are computed from scratch in every state. A derived predicate is of the form `(:derived (h ?v1 ... ?vn) b1 ... bm)` and can be inferred if and only if its body `b1 ... bm` is true in the current state. In contrast to state variables derived predicates can only be used in preconditions and effect conditions as well as in goal descriptions. Hence they cannot be set in effects of actions [16].

We can define the semantics of a planning task in the following way by assuming that effect literals are not conflicting.

Definition 1 (PDDL Semantics). *The semantics of a planning task $\Pi = \langle V, A, I, G \rangle$ is given by the following state transition system (S_0, L, u, g), with $S \subseteq V$ being a state:*

- $S_0 = \{p \mid p \in I\}$ *(the initial state)*
- $L = \{(a, S) \mid a \in A \wedge \mathsf{pre}_a \subseteq S\}$ *(a set of pairs (a, S) where action a can be performed in state S)*
- $u(a, S) = S \setminus \{p \mid (not\ p) \in \mathsf{eff}_a\} \cup \{p \mid p \in \mathsf{eff}_a\}$
  $\setminus \{p \mid (when\ (\mathsf{cpre})(\mathsf{ceff})) \in \mathsf{eff}_a \wedge \mathsf{cpre} \subseteq S \wedge (not\ p) \in \mathsf{ceff}\}$
  $\cup \{p \mid (when\ (\mathsf{cpre})(\mathsf{ceff})) \in \mathsf{eff}_a \wedge \mathsf{cpre} \subseteq S \wedge p \in \mathsf{ceff}\}$
  (update function giving the successor state for a chosen action a in state S)
- $g = \{S \mid G \subseteq S\}$ *(the set of goal states).*

Example Figure 1 shows a PDDL example of blocksworld. It contains two actions. The stack action (lines 5–12) takes some block ?x, which is currently on the table and clear (i.e., no other block is stacked on top of it), and places it on top of another block

?y, which must be clear as well. The unstack action (lines 13–17) removes a block from another block if they are currently stacked and the upper one is clear. In addition there are two derived predicates. The first one denotes that a block is clear if no block is placed on top of it (lines 18–19), the second one denotes that a block is on the table if it is not currently stacked on some block (lines 20–21).

```
1  (define (domain blocksworld)
2    (:predicates (clear ?x)
3                 (table ?x)
4                 (on ?x ?y) )
5    (:action stack
6      :parameters (?x ?y)
7      :precondition (and
8        (clear ?x)
9        (clear ?y)
10       (table ?x)
11       (not (= ?x ?y)) )
12     :effect (on ?x ?y) )
13   (:action unstack
14     :parameters (?x ?y)
15     :precondition (and (clear ?x)
16                        (on ?x ?y) )
17     :effect (not (on ?x ?y)) )
18   (:derived (clear ?x)
19     (forall (?y) (not (on ?y ?x))) )
20   (:derived (table ?x)
21     (forall (?y) (not (on ?x ?y))) ) )
22
23 (define (problem blocksworld3)
24   (:domain blocksworld)
25   (:objects A B C)
26   (:init
27       (on C A) )
28   (:goal
29       (and
30         (on A B)
31         (on B C) ) ) )
```

Fig. 1. PDDL example for Blocksworld.

In this instance we have three blocks, A, B, and C (line 25), where initially block C is on top of block A (lines 26–27). Following the derived predicates, blocks A and B thus reside on the table and blocks B and C are clear. The goal is to reach a state where block A is on top of block B and block B is on top of block C (lines 28–31).

2.2 General Game Playing

General games are modeled by use of the game description language GDL. This language is similar to logic programs, so that a game is modeled by a set of rules. In order to derive any meaning, some fixed keywords are used:

role(p) p is a player of the game.
init(f) f holds in the initial state.
true(f) f holds in the current state.
next(f) f holds in the successor state.
legal(p,m) Player p may perform move m.
does(p,m) Player p chooses to perform move m.
terminal Whenever terminal becomes true the game ends.
goal(p,rw) Player p achieves reward rw (an integer value in [0,100]).
distinct(x,y) x and y are semantically different.

All rules are of the form $(<= \; h \; b_1 \ldots b_n)$ with the meaning that head h will hold if all literals $b_1, \ldots, b_n$ of the body hold. Note that in GDL all operands are given in

prefix form, and that the conjunction of the bodies' literals is implicit. Rules with a head different from the keywords are called axioms, which in principle are very similar to derived predicates in PDDL: Their truth value must be derived based on the current state and with help of the other axioms.

The semantics of a general game modeled in GDL can be found in [4]. We give only the basic definition:

Definition 2 (GDL Semantics [4]). *The semantics of a valid GDL specification G of a general game is given by the following state transition system $\langle R, S_0, T, L, u, g\rangle$:*

- $R = \{r \mid G \vdash role(r)\}$ *(the players in the game)*
- $S_0 = \{p \mid G \vdash init(p)\}$ *(the initial state)*
- $T = \{S \mid G \cup S^{true} \vdash terminal\}$ *(the set of terminal states)*
- $L = \{(role, a, S) \mid G \cup S^{true} \vdash legal(role, a)\}$ *(relation specifying legal moves)*
- $u(A, S) = \{p \mid G \cup A^{does} \cup S^{true} \vdash next(p)\}$ *(state update function)*
- $g = \{(role, rw, S) \mid G \cup S^{true} \vdash goal(role, rw)\}$ *(relation specifying rewards)*

where A^{does} describes the moves that the players have chosen and S^{true} describes the state variables that are currently true in state S.

An important observation is that negation-as-failure is assumed, i.e., anything that cannot be derived is assumed to be false. That means especially that the successor state is fully specified by evaluating the `next` rules: Everything that can be derived by rules with head `next` will be true in the successor state, everything else will be false. In other words, in GDL the frame is modeled explicitly, while in PDDL only the state changes are modeled and the rest is assumed to remain unchanged.

Example Figure 2 shows a GDL example for the same blocksworld instance as the PDDL example. The initial state (lines 3–8) is extended by a step counter, which is needed as in GGP games are supposed to terminate after a finite number of steps [9]. In order to achieve the highest possible reward of 100, the robot must find a way to reach a state where block a is on top of block b and block b is on top of block c (lines 56–58). The moves for stacking (lines 10–14) and unstacking (lines 16–18) have the same preconditions as in the PDDL example. To determine the successor state the `next` rules must be evaluated. While those in lines 20–28 are responsible for the actual state update, those in lines 30–50 model the frame. The constant axioms (lines 52–54) are needed to model the step counter. The terminal rules (lines 64–68) indicate that the game ends after three steps or if block a is on top of block b and block b is on top of block c.

3 Translation

The basic idea of our translation is to use derived predicates to handle most of the GDL rules, and to perform the state update in three steps in order to translate GDL's explicit modeling of the frame to PDDL. First, we set the player's action. Then we evaluate the `next` rules in order to get a full description of the successor state. In this step we remove the current state variables and store the successor state in temporary variables. Finally we set the successor state to the temporarily stored one.

```
 1 (role robot)
 2
 3 (init (clear b))
 4 (init (clear c))
 5 (init (on c a))
 6 (init (table a))
 7 (init (table b))
 8 (init (step 0))
 9
10 (<= (legal robot (stack ?x ?y))
11     (true (clear ?x))
12     (true (table ?x))
13     (true (clear ?y))
14     (distinct ?x ?y))
15
16 (<= (legal robot (unstack ?x ?y))
17     (true (clear ?x))
18     (true (on ?x ?y)))
19
20 (<= (next (on ?x ?y))
21     (does robot (stack ?x ?y)))
22 (<= (next (table ?x))
23     (does robot (unstack ?x ?y)))
24 (<= (next (clear ?y))
25     (does robot (unstack ?x ?y)))
26 (<= (next (step ?y))
27     (true (step ?x))
28     (succ ?x ?y))
29
30 (<= (next (clear ?x))
31     (does robot (unstack ?u ?v))
32     (true (clear ?x)))
33 (<= (next (on ?x ?y))
34     (does robot (stack ?u ?v))
35     (true (on ?x ?y)))
36 (<= (next (clear ?y))
37     (does robot (stack ?u ?v))
38     (true (clear ?y))
39     (distinct ?v ?y))
40 (<= (next (on ?x ?y))
41     (does robot (unstack ?u ?v))
42     (true (on ?x ?y))
43     (distinct ?u ?x))
44 (<= (next (table ?x))
45     (does robot (stack ?u ?v))
46     (true (table ?x))
47     (distinct ?u ?x))
48 (<= (next (table ?x))
49     (does robot (unstack ?u ?v))
50     (true (table ?x)))
51
52 (succ 0 1)
53 (succ 1 2)
54 (succ 2 3)
55
56 (<= (goal robot 100)
57     (true (on a b))
58     (true (on b c)))
59 (<= (goal robot 0)
60     (not (true (on a b))))
61 (<= (goal robot 0)
62     (not (true (on b c))))
63
64 (<= terminal
65     (true (step 3)))
66 (<= terminal
67     (true (on a b))
68     (true (on b c)))
```

Fig. 2. GDL example for blocksworld.

3.1 Basic Translation

Apart from `init` and `goal` (see Section 3.3) all GDL rules are translated as derived predicates. Each rule of the form `(<= (h ?v1 ... ?vn) b1 ... bm)` is translated into a derived predicate of the form

```
(:derived (h ?v1 ... ?vn)
  (exists (?f1 ... ?fk)
    (and b1 ... bm ) ) )
```

where `?f1 ... ?fk` are the free variables appearing in the body.

Currently, we cannot handle functions in general. However, in order to store the current/next state as well as the legal/chosen moves, we translate a predicate of the form `(next (p ?v1 ... ?vn))` as `(next-p ?v1 ... ?vn)`, `(true (p ?v1 ... ?vn))` as `(current-p ?v1 ... ?vn)`, `(legal ?p (move ?v1 ... ?vn))` as `(legal-move ?p ?v1 ... ?vn)`, and `(does ?p (move ?v1 ... ?vn))` as `(does-move ?p ?v1 ... ?vn)`. Furthermore, `(distinct ?v1 ?v2)` is translated as `(not (= ?v1 ?v2))`. Any constants are translated with prefix `obj-`, as GDL allows constants starting with numbers while PDDL does not.

In case some variable `?v` appears more than once in the head we must replace all but one instance of it by new variables `?vi` and add `(= ?vi ?v)` to the body's conjunction for all of the replaced variables. If there is any constant `c` in the head we must replace it by a new variable `?cvar` and add `(= ?cvar c)` to the conjunction.

Constant axioms, i.e., axiom rules with empty body, can either be placed unchanged into the initial state description or translated into derived predicates as the other rules. If for some axiom constant and non-constant rules are present we must opt for the latter.

3.2 State Update

For any move the player may perform we create an action as depicted in Figure 3. This takes the same n parameters as the original GDL move. The precondition states that the game is not yet ended, the `play` predicate is true, which expresses that the player can choose a move in the current state, and that the player is allowed to actually perform the move in the current state, i.e., that the corresponding derived `legal` predicate can be evaluated to true. When applying the action we set the corresponding move and the `eval` predicate to true, which brings us to perform the first of the two additional actions that simulate the state update.

```
(:action move
  :parameters (?v1 ... ?vn)
  :precondition (and
    (not (terminal))
    (play)
    (legal-move player ?v1 ... ?vn) )
  :effect (and
    (does-move player ?v1 ... ?vn)
    (not (play))
    (eval) ) )
```

Fig. 3. Move action in the translation from GDL to PDDL.

The basic idea of the `eval` action (cf. Figure 4, top; we need only one that takes care of all predicates and moves) is to determine the successor state based on the current state and the chosen move and to store it in temporary variables. It evaluates all derived `next` predicates and sets the temporary variables corresponding to the satisfied ones. In addition it removes all `current` state variables as well as the chosen move and changes from `eval` to `switch`, which brings us to the second additional action.

The `resume-play` action is depicted in the bottom of Figure 4. It sets the new state's `current` state variables and removes the temporary ones. Then it changes back to the `play` predicate so that the player's next move can be chosen. Similar to the `eval` action, we use only one `resume-play` action that handles all predicates.

A state update with only a single additional action would be possible as well by creating two sets of conditional effects: One as before, setting a state variable to true

```
(:action eval
  :precondition (eval)
  :effect (and
    (forall (?vp1 ... ?vpn)
      (when (next-p ?vp1 ... ?vpn) (temp-p ?vp1 ... ?vpn) )
    (forall (?vp1 ... ?vpn)
      (not (current-p ?vp1 ... ?vpn)) )
    (forall (?vm1 ... ?vmn)
      (not (does-move player ?vm1 ... ?vmn)) )
    (not (eval))
    (switch) ) )

(:action resume-play
  :precondition (switch)
  :effect (and
    (forall (?vp1 ... ?vpn)
      (when
        (temp-p ?vp1 ... ?vpn)
        (and
          (current-p ?vp1 ... ?vpn)
          (not (temp-p ?vp1 ... ?vpn)) ) ) )
    (not (switch))
    (play) ) )
```

Fig. 4. The `eval` and `resume-play` actions.

if at least one of the corresponding `next` predicates can be evaluated to true, the other setting it to false if all corresponding `next` predicates are evaluated as false. However, this often results in formulas in CNF, which most planners cannot handle efficiently.

3.3 Initial State, Rewards, and Termination

The initial state of the PDDL translation captures all state variables that are initially true. Thus, each rule `(init (f v1 ... vn))` in the GDL description is translated as `(current-f v1 ... vn)` and added to the initial state. In addition, we must add the predicate `play`.

In our current setting we are only interested in finding an optimal solution, i.e., one that leads us to a goal state that achieves a reward of 100. Due to the GDL specification of a well-formed game [9] the possibility to reach a goal state with reward 100 must exist. Presuming that we have a set of goal rules:

```
(<= (goal ?p 100)
  b11 ... b1n)
...
(<= (goal ?p 100)
  bm1 ... bmn)
```

we build up the following goal description in PDDL:

```
(:goal
  (and
    (terminal)
    (or
      (exists (?v1_1 ... ?v1_n) (and b11 ... b1n))
      ...
      (exists (?vm_1 ... ?vm_n) (and bm1 ... bmn)) ) ) )
```

where `?vi_1 ... ?vi_n` are the free variables in the body of the i-th goal rule. The `terminal` predicate is necessary because we want to reach a terminal state with reward 100; the reward of the non-terminal states is irrelevant for our purposes.

3.4 Restrictions of the Translation

As pointed out earlier, currently we can only translate functions in a very limited form, namely the functions representing the state variables, as well as those representing moves. Any other form of functions is not yet supported. Replacement of any of the functions by a variable is unsupported as well. It remains future work to find efficient ways to handle these cases.

The current form of our translation cannot handle GDL descriptions that use the `goal` predicate in the body of other rules. A straight-forward fix is to translate all `goal` rules into derived predicates as well.

Concerning the rewards, we are currently only translating the maximum reward 100. By making use of plan cost minimization it is possible to also handle different rewards. We can define a new action `reach-goal` that has `terminal` and `(goal role rw)` in the precondition (where `rw` corresponds to a natural number) and sets a unique variable `goal-reached` to true. This new action has a cost of $100-$`rw`, while all other actions have a cost of 0. Then the goal description reduces to the variable `goal-reached` being true. Overall, this means that if we reach a terminal state that has reward 100 in GDL we can use the action `reach-goal` with cost 0 to reach the goal in PDDL. As planners try to minimize the total cost they will thus try to reach a terminal state that has the highest possible reward.

3.5 Correctness of the Translation

In order to prove the correctness of our translation we must define the semantical equivalence of a PDDL and GDL state w.r.t. our translation.

Definition 3 (Semantical equivalence of a PDDL state and a GDL state). *A PDDL state S^{PDDL} of a state transition system $(S_0^{PDDL}, L^{PDDL}, u^{PDDL}, g^{PDDL})$ and a GDL state S^{GDL} of a state transition system $(R, S_0^{GDL}, T, L^{GDL}, u^{GDL}, g^{GDL})$ are semantically equivalent if all of the following holds:*

1. $\forall x : x \in S^{true,GDL} \Leftrightarrow (current\text{-}x) \in S^{PDDL}$
2. $\forall x : x \in R \Leftrightarrow (role\ obj\text{-}x) \in S^{PDDL}$

3. $\forall a : (role, a, S^{GDL}) \in L^{GDL} \Leftrightarrow \exists\, a \in A : (a, S^{PDDL}) \in L^{PDDL}$
4. $S^{GDL} \in T \Leftrightarrow \exists\, terminal \in derived\_predicates : S^{PDDL} \vdash terminal$
5. *the axiom rule ax can be fired in* $S^{GDL} \Leftrightarrow$ *the derived predicate of ax can be derived in* S^{PDDL}

where A *is the set of actions and* $derived\_predicates$ *is the set of derived predicates that occur in the PDDL task.*

We are allowed to use Definition 1 (PDDL Semantics) in the definition of semantical equivalence above because the assumption of Definition 1 is fulfilled: The translation from GDL to PDDL does not create conflicting literals in effects per definition.

Due to space restrictions we can only outline the proof ideas; a longer version can be found in [12].

The basic idea is to prove the correctness by means of induction. First we can show that the initial states in both settings are semantically equivalent, which is apparent due to the translation of the initial state and the other rules in form of derived predicates.

Next we can prove that, for semantically equivalent states, the application of a move in GDL and the three steps for the state update in PDDL result in semantically equivalent states again. This follows immediately from the actions in our PDDL translation: The move action sets the player's chosen move, the `eval` action evaluates the `next` predicates based on the chosen move and the current state and stores only the positive `next` predicates in temporary variables while removing all instances of the `current` state variables. Finally, the `resume-play` action sets the `current` state variables according to the temporary ones.

The last thing to prove is that PDDL's goal states secure the highest possible reward of 100, which again follows immediately from the construction – we translate the conditions of the `goal` rules that award 100 points as the goal condition.

4 Empirical Evaluation

In the following we present empirical results of the proposed translation and a comparison in terms of coverage and runtime between two planners (Fast-Forward (FF) [6] and FastDownward (FD) [5]) and a GGP agent (the symbolic solver and explicit UCT player in Gamer [7]) taking the original input. All tests were performed on a machine with an Intel Core i7-2600k CPU with 3.4 GHz and 8 GB RAM. We tried to translate 55 single-player games from the ggpserver[1] with our implementation of the translation. Nine games cannot be translated by our approach due to the restrictions we posted earlier in Section 3.4. For the remaining 46 games the second column of Table 1 shows the required time of the translation. These times range between 0.15s and 0.35s, mainly dependent on the size of the game description.

The first numbers of the third and fourth column show the preprocessing (esp. grounding) and solving time of FF, the first numbers of columns 5 and 6 indicate the necessary time of the translate and search process of FD (among other things the translate process is responsible for grounding the input), and the last three columns detail

[1] `http://ggpserver.general-game-playing.de`

Table 1. Empirical results of 46 GDL games. Games where we could remove the step-counter are denoted with an asterisk (*). For FF and FD the numbers in parantheses are the times for the games with removed step-counter. All times in seconds.

Game	Trans	FF pre	FF	FD trans	FD search	G. inst	G. solv	G. play
8puzzle(*)	0.17	– (–)		9.06 (2.33)	535.54 (0.38)	0.63	22.67	err
asteroids(*)	0.2	0.7 (0.06)	5.82 (1.76)	0.51 (0.38)	0.34 (0.53)	0.16	1.69	2.85
asteroidsparallel(*)	0.21	10.47 (1.23)	– (–)	4.36 (1)	16.2 (27.85)	0.03	–	err
asteroidsserial(*)	0.23	2.41 (0.31)	– (–)	10.11 (5.8)	– (472.92)	0.41	–	–
blocks(*)	0.16	0.01 (0.01)	0.01 (0.01)	0.07 (0.08)	0 (0.02)	0.06	0.33	0.05
blocksworldparallel	0.18	–		17.57	0.24	0.31	0.77	0.06
blocksworldserial	0.18	0.04	–	0.24	0	0.1	0.57	0.06
brain_teaser_extended(*)	0.19	5.56 (0.1)	– (–)	NAI (NAI)		0.17	1.56	8.22
buttons(*)	0.17	0 (0)	0 (0.01)	0.05 (0.06)	0 (0.02)	0.05	0.3	0.05
circlesolitaire	0.16	0.02	–	NAI		0.06	0.5	0.06
cube_2x2x2(*)	0.2	– (–)		798.83 (78.24)	0.1 (0.09)	0.16	–	0.8
duplicatestatesmall	0.19	0.01	–	0.96	0.2	0.06	0.46	0.05
duplicatestatemedium	0.26	0.17	–	–		0.18	1.17	0.23
duplicatestatelarge	0.35	1.55	–	–		0.64	6.16	1.54
firefighter	0.18	–		0.34	0	0.21	–	err
frogs_and_toads(*)	0.24	– (–)		– (–)		32.94	–	–
god(*)	0.19	– (–)		67.73 (38.16)	0.38 (0.23)	–		9.67
hanoi	0.16	0.73	88.26	0.46	0.24	0.14	0.67	2.83
hanoi_6_disks(*)	0.17	232.22 (0.09)	– (–)	NAI (NAI)		0.67	1.93	13.74
hitori	0.21	–		–		–		0.38
incredible(*)	0.18	– (0.64)	– (–)	1.28 (1.25)	0.08 (0.11)	0.33	4.55	57.73
kitten_escapes_from_fire	0.18	234.36	–	NAI		0.5	0.8	0.09
knightmove	0.19	–		NAI		1.4	–	250.93
knightstour	0.18	–		–		0.24	316.2	3.07
lightsout(*)	0.18	– (–)		NAI (NAI)		0.1	183.8	err
lightsout2(*)	0.16	– (0.14)	– (–)	NAI (NAI)		0.1	185.12	err
max_knights	0.2	–		NAI		69.46	–	err
maze(*)	0.15	0.01 (0)	– (0)	0.07 (0.07)	0 (0.02)	0.05	0.35	0.06
mimikry(*)	0.2	– (–)		NAI (NAI)		11.3	4.23	12.44
oisters_farm(*)	0.18	0.06 (0.01)	0.06 (0.01)	0.32 (0.15)	0 (0.03)	0.08	0.63	0.61
pancakes	0.16	–		–		1.0	1.26	12.19
pancakes6	0.16	–		–		0.99	1.23	32.87
pancakes88	0.17	–		–		–		err
peg_bugfixed	0.18	–		–		1.63	–	err
queens	0.19	–		NAI		–		–
ruledepthlinear	0.27	–		9.05	0	0.25	0.61	0.08
ruledepthquadratic	0.34	–		–		–		–
statespacesmall	0.15	0.02	–	0.04	0	0.06	0.48	0.08
statespacemedium	0.22	0.03	0.03	0.09	0	0.15	1.46	err
statespacelarge	0.32	0.07	0.17	0.19	0.02	0.39	0.29	err
tpeg	0.18	–		–		1.77	–	err
troublemaker01	0.15	0.03	–	0.02	0	0.05	0.29	0.04
troublemaker02	0.15	0.01	–	0.02	0	0.05	0.29	0.05
twisty-passages	0.18	–		46.95	0.84	50.31	3.84	13.9
walkingman(*)	0.2	– (–)		28.58 (30.23)	0.18 (0.13)	878.13	3.91	0.06
wargame01(*)	0.23	– (–)		NAI (–)		–		0.26
Coverage	46	21/46 (11/19)	7/46 (5/19)	24/46 (12/19)	23/46 (12/19)	40/46	31/46	31/46

(from left to right) the results of Gamer's instantiator, solver, and Prolog-based UCT player. The value "–" indicates that the process did not terminate within the time limit of 900 seconds. In case of FD note that it cannot handle nested negated axioms (i.e., negated axioms dependent on other negated axioms) and throws an error, here denoted by "NAI" (Negated Axioms Impossible). The settings we used for the FD search process are: --heuristic "hff=ff()" --search "lazy_greedy(hff, preferred=hff)".

For FF we can see that it can ground 21 out of the 46 translatable games and solve only 7 of those within 900 seconds. With FD we can solve 23 games within this time limit. In most cases FD's translate process takes either too long (11 cases) or has trouble with nested negated axioms (11 cases). In a single case (asteroidsserial) FD's translate process finishes but no solution can be found in time. Gamer is able to solve 31 games within 900 seconds. The comparison of the runtimes between FD and Gamer delivers a mixed result. FD is faster than Gamer in 19 games whereas Gamer is faster in 17 games, but these include 7 games where FD throws the NAI exception. The UCT player can also find solutions for 31 of the games. Unfortunately, it crashes with some Prolog-related errors in ten cases. In the cases it runs fine it is able to solve four games that the solver can not. In the other games often the player is much slower than the solver and only in few cases clearly faster.

At first it seems surprising that, on the one hand, there are games like asteroidsparallel that can be solved very fast by FD while Gamer cannot find a solution within 900 seconds. On the other hand there are games like 8puzzle where the pure solving time is clearly in favor of Gamer. One explanation for this behavior becomes apparent when considering step-counters, which 19 of the games contain to ensure termination after a finite number of steps. While Gamer requires games to be finite, the planners do not mind if some paths exist where the game does not terminate. For these games we manually removed the step-counters and ran the planners again (the number in parantheses in the table). In this case we can see that in several games the time and memory-overhead for both grounding and solving decreases. FF can ground two additional games and FD can solve one additional game. The biggest decrease in solving time comes with the 8puzzle, where FD's time reduces from 536s to 0.4s, clearly below that of Gamer. However, if the step-counter is removed and we want to use the found plan in the original game we have to somehow make sure that the plan does not exceed the allowed length.

When only looking at the grounding times we can see that Gamer nearly never is slower than FF and FD if we consider only games where grounding takes more than 1s. The only domain that causes big problems for Gamer in contrast to FD is walking-man. However, the instantiator of Gamer handles the GDL input directly while FD and FF have to cope with the translated files, which bring some overhead. Thus, it is not surprising that Gamer is much faster in grounding the games.

5 Discussion and Conclusion

So far we are not aware of any refereed publications of a translation from GDL to PDDL. The only works we found are two students' theses providing translations as well. One [14] tries to find add- and delete effects, so that the state update is much easier, as the frame effects can be removed. Also, it is able to find types for different constants, which should speed up the grounding and reduce the memory consumption. However, this comes at quite an overhead as the axioms containing `does` terms must be rolled out so that in several cases the translator ran out of memory. Furthermore, the generated output seems to be quite troublesome for FD; of the tested games not a single one could be handled by that planner. The other thesis [15] tries to perform the state update in a single step using conditional effects. Here, several things remain unclear

in the description (e.g., the handling of axioms in general and especially that of `does` terms appearing in the bodies of axioms) and no experimental results are presented.

We have proposed a new way to translate single-player games. From the results we have seen that the translation works fine, with some restrictions in the use of functions. The resulting PDDL files can be handled by state-of-the-art planners, though especially the grounding in those planners seems to be a bottleneck, especially in games containing a step-counter. There are several cases where a planner is faster than both, the GGP solver and player we compared against, so that it might indeed make sense to run this kind of translation and solve the games by means of a planner. Nevertheless, classical GGP agents should be run in parallel as well, as we cannot predict whether the planner or the GGP player will be more efficient. This way we can get the best of both worlds.

References

1. Björnsson, Y., Finnsson, H.: Cadiaplayer: A simulation-based general game player. IEEE Transactions on Computational Intelligence and AI in Games 1(1), 4–15 (2009)
2. Edelkamp, S., Hoffmann, J.: PDDL2.2: The language for the classical part of the 4th international planning competition. Tech. Rep. 195, Albert-Ludwigs-Universität, Institut für Informatik, Freiburg, Germany (2004)
3. Genesereth, M.R., Love, N., Pell, B.: General game playing: Overview of the AAAI competition. AI Magazine 26(2), 62–72 (2005)
4. Haufe, S., Schiffel, S., Thielscher, M.: Automated verification of state sequence invariants in general game playing. Artificial Intelligence 187, 1–30 (2012)
5. Helmert, M.: The Fast Downward planning system. Journal of Artificial Intelligence Research 26, 191–246 (2006)
6. Hoffmann, J.: FF: The fast-forward planning system. The AI Magazine 22(3), 57–62 (2001)
7. Kissmann, P., Edelkamp, S.: Gamer, a general game playing agent. KI 25(1), 49–52 (2011)
8. Kocsis, L., Szepesvári, C.: Bandit based monte-carlo planning. In: Proceedings of the 17th European conference on Machine Learning (ECML'06). pp. 282–293 (2006)
9. Love, N.C., Hinrichs, T.L., Genesereth, M.R.: General game playing: Game description language specification. Tech. Rep. LG-2006-01, Stanford Logic Group (2008)
10. McDermott, D., et al.: The PDDL Planning Domain Definition Language. The AIPS-98 Planning Competition Comitee (1998)
11. Méhat, J., Cazenave, T.: A parallel general game player. KI 25(1), 43–47 (2011)
12. Rauber, T.: Translating Single-Player GDL into PDDL. Bachelor's thesis, Department of Computer Science, Faculty of Natural Sciences and Technology I, Saarland University (2013)
13. Richter, S., Westphal, M.: The LAMA planner: Guiding cost-based anytime planning with landmarks. Journal of Artificial Intelligence Research 39, 127–177 (2010)
14. Rüdiger, C.: Use of existing planners to solve single-player games. Großer Beleg, Fakultät Informatik, Technische Universität Dresden (2009)
15. Sievers, S.: Erweiterung eines Planungssystems zum Lösen von Ein-Personen-Spielen. Bachelor's thesis, Arbeitsgruppe Grundlagen der künstlichen Intelligenz, Institut für Informatik, Albert-Ludwigs Universität Freiburg (2009)
16. Thiebaux, S., Hoffmann, J., Nebel, B.: In defense of PDDL axioms. Artificial Intelligence 168(1–2), 38–69 (2005)