Model Checking Games in GDL-II

Ji Ruan and Michael Thielscher

The University of New South Wales, Australia
Email: {jiruan, mit}@cse.unsw.edu.au

Abstract. The game description language GDL has been developed as a logic-based formalism for representing the rules of arbitrary games in general game playing. A recent language extension called GDL-II allows the description of nondeterministic games with any number of players who may have incomplete, asymmetric information. In this paper, we apply *model checking* to address the problem of verifying that games specified in GDL-II satisfy appropriate temporal and knowledge conditions. We present a systematic translation of a GDL-II description to a model checking tool, and show the feasibility by two case studies.

1 Introduction

The general game description language GDL, which has been established as input language for general game-playing systems [7, 10], has recently been extended to GDL-II to incorporate games with nondeterministic actions and where players have incomplete/imperfect information [20]. However, not all GDL-II descriptions correspond to games, let alone meaningful, non-trivial games. [7, 10] list a few properties that are necessary for well-formed GDL games, e.g., it terminates after finite steps and all players have at least one legal move in non-terminal states. The introduction of incomplete information to GDL-II also raises new questions, e.g., can players *always know* their legal moves in non-terminal states or *know* their goal values in terminal states?

Temporal Logics have been applied to the verification of computer programs, or more broadly computer systems, initially by A. Pnueli and Z. Manna et al. [14, 11], and by E. Clarke and E. A. Emerson et al. [4]. The programs are in certain states at each time instance, and the correctness of the programs can be expressed as temporal specifications, such as "$AG\neg deadlock$" meaning *the program can never enter a deadlock state*. Epistemic logics, on the other hand, are the formalisms of knowledge and beliefs. Its application in verification was originally motivated by the need to reason about communication protocols. One is typically interested in what knowledge different parties to a protocol have before, during and after a run (an execution sequence) of the protocol. [5] gives a comprehensive study on epistemic logic for multiple interacting agents.

We have previously analysed the epistemic logic behind GDL-II and in particular shown that the situation at any stage of a game can be characterised by a multi-agent epistemic (i.e., S5-) model [16]. Yet, this result only provides a static characterisation of what players know (and don't know) at a certain stage. This paper extends such analysis with a temporal dimension, and also provides a practical method for verifying temporal and epistemic properties using a model checker named MCK [6]. The main idea is to translate a GDL-II description into the model specification language of MCK in a

systematic and equivalent way. Checking whether a property φ holds for description G is then equivalent to checking whether φ holds for the translation $\pi(G)$. The latter can be automatically checked in MCK.

The paper is organised as follows. Section 2 introduces GDL-II and MCK. Section 3 gives the main translation and some optimisations that can be applied to the translation. Experimental results are given for two cases in Section 4. The paper concludes with a discussion of related work and directions for further research.

2 GDL-II and MCK

GDL-II A complete game description consists of the names of (one or more) players, a specification of the initial position, the legal moves and how they affect the position, and the terminating and winning criteria. The emphasis of game description languages is on high-level, declarative game rules that are easy to understand and maintain. At the same time, GDL and its successor GDL-II have a precise semantics and are fully machine-processable. Moreover, background knowledge is not required—a set of rules is all a player needs to know to be able to play a hitherto unknown game. The description language GDL-II uses these *keywords*:

`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)`	`?r` can do 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,?v)`	goal value for role `?r` is `?v`
`sees(?r,?p)`	`?r` perceives `?p` in the next position
`random`	the random player

GDL (without `sees` and `random`) is suitable for describing finite, synchronous, and deterministic n-player games with complete information about the game state [10]. The extended game description language GDL-II allows the specification of games with randomness and imperfect/incomplete information [20]. Valid game descriptions must satisfy certain syntactic restrictions; for details we have to refer to [10] for space reasons.

The GDL-II rules in Fig. 1 formalise a simple but famous game called *Monty Hall* where a car prize is hidden behind one of three doors and where a candidate is given two chances to pick a door. The intuition behind the rules is as follows. Line 1 introduces the players' names (the game host is modelled by `random`). Lines 3–4 define the four features that comprise the initial game state. The possible moves are specified by the rules for legal: in step 1, the `random` player must decide where to place the car (line 6) and, simultaneously, the candidate chooses a door (line 10); in step 2, `random` opens a door that is not the one that holds the car nor the chosen one (lines 7–8); finally, the candidate can either stick to their earlier choice (noop) or switch to the other, yet unopened door (line 12 and 13, respectively). The candidate's only percept throughout the game is to see the door opened by the host (line 15) and where the car is after step 3

```
1  role(candidate).  role(random).
2
3  init(closed(1)).  init(closed(2)).  init(closed(3)).
4  init(step(1)).
5
6  legal(random,hide_car(?d))  <= true(step(1)), true(closed(?d)).
7  legal(random,open_door(?d)) <= true(step(2)), true(closed(?d)),
8                                 not true(car(?d)), not true(chosen(?d)).
9  legal(random,noop)          <= true(step(3)).
10 legal(candidate,choose(?d)) <= true(step(1)), true(closed(?d)).
11 legal(candidate,noop)       <= true(step(2)).
12 legal(candidate,noop)       <= true(step(3)).
13 legal(candidate,switch)     <= true(step(3)).
14
15 sees(candidate,?d) <= does(random,open_door(?d)).
16 sees(candidate,?d) <= true(step(3)), true(car(?d)).
17
18 next(car(?d))    <= does(random,hide_car(?d)).
19 next(car(?d))    <= true(car(?d)).
20 next(closed(?d)) <= true(closed(?d)), not does(random,open_door(?d)).
21 next(chosen(?d)) <= does(candidate,choose(?d)).
22 next(chosen(?d)) <= true(chosen(?d)), not does(candidate,switch).
23 next(chosen(?d)) <= does(candidate,switch),
24                    true(closed(?d)), not true(chosen(?d)).
25 next(step(2)) <= true(step(1)).
26 next(step(3)) <= true(step(2)).
27 next(step(4)) <= true(step(3)).
28
29 terminal <= true(step(4)).
30
31 goal(candidate,100) <= true(chosen(?d)), true(car(?d)).
32 goal(candidate,  0) <= true(chosen(?d)), not true(car(?d)).
33 goal(random,0).
```

Fig. 1. A GDL-II description of the Monty Hall game adapted from [21].

(line 16). The remaining rules specify the state update (rules for next), the conditions for the game to end (rule for terminal), and the payoff for the player depending on whether they got the door right in the end (rules for goal).

We refer the formal semantics of GDL-II to [16] due to limited spaces. The semantics enables us to derive a game model from a given game description.

MCK In this paper, we will use MCK, for 'Model Checking Knowledge', which is a model checker for temporal and knowledge specifications [6, 12]. The overall setup of MCK supposes a number of agents acting in an environment. This is modelled by an interpreted system where agents perform actions according to protocols. Actions and the environment may be only partially observable at each instant in time. In MCK, different approaches to the temporal and epistemic interaction and development are implemented. Knowledge may be based on current observations only, on current observations and clock value, or on the history of all observations and clock value. The last corresponds to *synchronous perfect recall*. In the temporal dimension, the specification formulas may describe the evolution of the system along a single computation, i.e., using linear time temporal logic, or they may describe the branching structure of all possible computations, i.e, using branching time or computation tree logic. We give the basic syntax of Computation Tree Logic of Knowledge (CTLK).

Definition 1. *The language of CTLK (with respect to a set of atomic propositions Φ), is given by the following grammar:*

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \psi \mid AX\varphi \mid AF\varphi \mid AG\varphi \mid A\varphi \mathcal{U} \psi \mid E\varphi \mathcal{U} \psi \mid K_i\varphi$$

where $p \in \Phi$. Other logic constants and connectives $\top, \bot, \vee, \rightarrow$ are defined as usual.

We only explain the semantics informally here (cf. [17] for more details). The formulas of CTLK can be interpreted on states of game models. A game model consists of a set of agents, a set of possible states (esp., one initial state and a subset of terminal states), and a transition function for states. Intuitively $AX\varphi$ means that for all the next states φ must hold; $AF\varphi$ means that for all the paths of the game φ will eventually hold in the future; $AG\varphi$ means that for all the paths of the game φ always hold in the future; $A\varphi \mathcal{U} \psi$ means that for all the paths of the game, φ holds until ψ holds; $E\varphi \mathcal{U} \psi$ means that there exists a path of the game, φ holds until ψ holds; and $K_i\varphi$ means that φ holds in all the states that agent i can not distinguish from. An agent with synchronous perfect recall, can not distinguish two states if it made the same moves and had the same perceptions along two histories from the initial state.

3 Translation from GDL-II to MCK

Given a GDL-II description G, our program generates a translation $\pi(G)$ as the input for MCK. The result of the translation, $\pi(G)$, is equivalent to G in the sense that, the game model derived from G using GDL-II semantics satisfies same formulas as the model that is derived from $\pi(G)$ using MCK operational semantics.

Proposition 1. *Given a GDL-II description G, let $\pi(G)$ be the translation from GDL-II to MCK and φ a temporal epistemic property, then:*

$$G \models_{GDL} \varphi \text{ iff } \pi(G) \models_{MCK} \varphi$$

This enables us to check temporal epistemic properties against G by checking them against $\pi(G)$, which can be done by MCK automatically. For detailed proof, see [17].

We use the GDL-II description of Monty Hall game in Fig. 1, denoted as G_{MH}, to illustrate the whole process. The translation π can be divided into the following steps.

Computing Domains

The first step is to compute the domains, or rather supersets of the domains, of all predicates and functions of the game description. This is done by generating a dependency graph from the rules of the game description, following [19]. The nodes of the graph are the arguments of functions and predicates in game description, and there is an edge between two nodes whenever there is a variable in a rule of the game description that occurs in both arguments. Connected components in the graph share a (super-)domain.

Take, for example, the Monty Hall rules, line 3 and 6 give us the domain graph in Fig. 2, from which it can be seen that the argument of both $closed$ and $hide\_car$ ranges over the domain $\{1, 2, 3\}$.

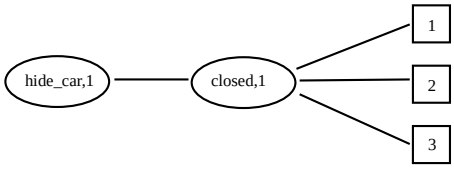


Fig. 2. A domain graph for calculating domains of functions and predicates.

Once we have computed the domains, we instantiate all the variables in G. E.g., the rule in line 6-7 in the above example are turned into three rules

```
legal(random,hide_car(1))  <= true(step(1)), true(closed(1)).
legal(random,hide_car(2))  <= true(step(1)), true(closed(2)).
legal(random,hide_car(3))  <= true(step(1)), true(closed(3)).
```

Deriving MCK Variables

The second step is to derive all the variables for $\pi(G)$. For this, we distinguish predicates that occur as arguments of `init` or `true`, and those that do not. The former are translated to boolean variables. For example, $\texttt{step}(1)$ and $\texttt{closed}(1)$ appear in $\pi(G_{MH})$ as

```
step_1: Bool
closed_1: Bool
```

Predicates of the second type typically depend on the first type of expressions. E.g., the following rule shows that the $legal$ predicate depends on two $true$ predicates:

```
legal(random,hide_car(1))  <= true(step(1)), true(closed(1)).
```

There are two ways to deal with these cases: (1) translate them into booleans like above and then use valuation statements or (2) directly *define* them in terms of the booleans translated from the first type of predicates. For example, we can translate $\texttt{legal}(\texttt{random}, \texttt{hide\_car}(1))$ to a boolean (and assign a proper value later):

```
legal_random_hide_car_1: Bool
legal_random_hide_car_1:= step_1 /\ closed_1
```

or as a definition,

```
define legal_random_hide_car_1 = step_1 /\ closed_1
```

where `step_1` and `closed_1` are both booleans and $/\backslash$ is the symbol for conjunction.

The advantage of using definitions is that no new variables are introduced to the state representation. This reduces the number of variables in the overall translation and therefore potentially saves model checking time. The predicates `terminal` and `goal` can also be treated this way.

In GDL-II, `sees` predicates specify the perceptions of agents. Such predicates depend on the first type of predicates as well but they cannot be given as definitions in the translation because they have to be observable for the relevant agents in agent protocols (given below). Therefore we translate such predicates into separate boolean variables.

Since agents can recall their past moves, we make moves as part of the history, along with the perceptions of agents. While MCK's algorithms for CTLK with perfect

recall semantics do not include moves as part of the history, we need to embed such information as part of a state. Therefore we introduce an extra boolean for each `legal` instance, and replace `legal` with `did`. E.g., for $\texttt{legal}(\texttt{random}, \texttt{hide\_car}(1))$, we add

```
did_random_hide_car_1: Bool.
```

The above procedure can already generate all the variables needed. We can do further optimisation on two kinds of predicates: these appearing in the rules with empty bodies and those never appearing in the head of rules. Under GDL-II semantics, the first kind is always true and the second kind is always false. Therefore we can replace them universally with their corresponding truth values. E.g., consider the following program:

```
1 succ(1,2)
2 succ(2,3)
3 next(step(?y)) <= true (step(?x)), succ(?x, ?y).
```

We can first translate the program to the following by using the dependency graph:

```
4 succ(1,2)
5 succ(2,3)
6 next(step(2)) <= true(step(1)), succ(1,2).
7 next(step(2)) <= true(step(2)), succ(2,2).
8 next(step(3)) <= true(step(2)), succ(2,3).
9 next(step(3)) <= true(step(3)), succ(3,3).
```

Because both $\texttt{succ}(2,2), \texttt{succ}(3,3)$ are always false, and $\texttt{succ}(1,2), \texttt{succ}(2,3)$ are always true, we replace them using their truth values. Then we can further simplify this program by removing the rules with a "False" conjunct, and by removing the "True" conjuncts universally:

```
10 next(step(2)) <= true(step(1)).
11 next(step(3)) <= true(step(2)).
```

It is easy to check that lines 10–11 are equivalent to lines 1–3 (and also lines 4–9) in terms of changes over `step` predicates. This will effectively reduce the number of variables in the translation.

Initial Conditions

This step specifies the initial condition of $\pi(G)$. All the booleans translated from the predicates included within `init` are made true and all other predicates are made false. Taking G_{MH} (lines 3-4) for example, we have

```
init_cond =
closed_1 == True /\ closed_2 == True /\ closed_3 == True /\
step_1 == True /\  step_2 == False /\ step_3 == False /\ step_4 == False /\
car_1 == False /\ car_2 == False /\ car_3 == False /\ ...
```

Agent Protocols

This step specifies the agents and their protocols during the game play. The agent binding operation binds distinct agent names to the protocols they run, and instantiate each protocol's parameters. In GDL-II, the names of the agents are read off from the rules for the `role` predicate, and moves are read off from the `legal` predicates. Each agent has its own protocol. In MCK, protocol parameters are typed and some have $observable$ before the type to indicate that agents can *see* these parameters, which are then used for agents' accessibility relations. Taking the role `candidate` in G_{MH} for example, the following protocol is constructed in $\pi(G_{MH})$,

```
protocol "candidate" (
step_1: Bool, step_2: Bool, step_3: Bool, ... ,
sees_Candidate_1: observable Bool, sees_Candidate_2: observable Bool, ...,
did_Candidate_Choose_1: observable Bool, ... )
begin do
  legal_Candidate_Choose_1 -> <<Choose_1>>
  [] legal_Candidate_Choose_2 -> <<Choose_2>>
  [] legal_Candidate_Choose_3 -> <<Choose_3>>
  [] legal_Candidate_Noop -> <<Noop>>
  [] legal_Candidate_Switch -> <<Switch>>
  od
end
```

Here the parameters prefixed with `sees_` or `did_` are observable to the `candidate`. The variables prefixed with `legal_` are booleans or definitions (explained above) and they represent the preconditions of moves, e.g., `legal_Candidate_Choose_1` is the precondition for agent to chose move $<<$ `Choose_1` $>>$. "[]" means non-deterministic choice, so in each step, one of these statements within `do`...`od` will be non-determinstically executed whenever their guards are true.

We bind agent `Candidate` to the above protocol `candidate` as this:

```
agent  Candidate "candidate" ( parameter variables )
```

A protocol can be bound to multiple names, so this gives potential to code reuse when several agents share the same protocol.

State Transition

This step specifies the statements that update the variables after agents have decided which moves to make. The first part is update the variables prefixed with `did_`. E.g., the following lines in $\pi(G_{MH})$ indicates that agent `Candidate` made move `Choose_1` on the previous state.

```
if Candidate.Choose_1 -> did_Candidate_Choose_1 := True
[] otherwise -> did_Candidate_Choose_1 := False
fi;
```

Essentially, the effects of the agents' joint actions will be computed so this section is connected to the `does` and `next` predicates in the description G. Here we use Clark Completion to update these variables. This can be illustrated by the variable `chosen_1` from our running example. First take all the rules that have `chosen`(1) in the head from the original description G_{MH} and instantiate ?`d` with 1:

```
1 next(chosen(1)) <= does(candidate,choose(1)).
2 next(chosen(1)) <= true(chosen(1)),
3                    not does(candidate,switch).
4 next(chosen(1)) <= does(candidate,switch),
5                    true(closed(1)),
6                    not true(chosen(1)).
```

Then translate the bodies of these three rules and take their disjunction to be the guard of the resulting 'if' statement as follows: (note that `chosen`(1) is translated to `chosen_1` and so are the other ground atoms)

```
if (did_Candidate_Choose_1) \/ (chosen_1 /\ neg did_Candidate_Switch) \/
(did_Candidate_Switch /\ closed_1 /\ neg chosen_1) -> chosen_1 := True
[] otherwise -> chosen_1 := False
fi;
```

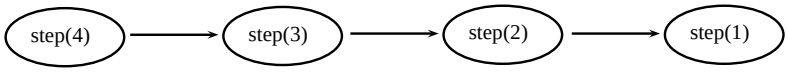
In addition, we may need to arrange the order of MCK statements carefully, because MCK's input language is imperative, which means that the statements are executed in a given order. In contrast, GDL-II is a declarative language and the order of the rules does not change the meaning (or semantics) of the whole description. Take the following example:

```
1 next(step(2)) <= true(step(1)).
2 next(step(3)) <= true(step(2)).
3 next(step(4)) <= true(step(3)).
```

The first rule means that if `step(1)` is true in the current state, then `step(2)` will be true in the next state. Note that this rule is the only rule with head `next(step(2))`, so we can apply the Clark Completion and get a statement: `step_2` := `step_1`. This is fine by itself, but we have a problem when the three rules are translated together in the original order:

```
step_2 := step_1;
step_3 := step_2;
step_4 := step_3;
```

The problem is this: if `step_1` is true originally, then after executing this three statements we have all the variables to be true, whereas in GDL-II `step(3)` and `step(4)` would still be false in the next state. In fact, when we update the value of `step_3` in MCK, we need to make sure to use the guard value `step_2` from the previous state. When we follow the exact order of above, then `step_2` is updated before `step_3` is getting updated. One solution to this problem is to use a dependency graph:



This graph indictaes the order in which the variables need to be updated: `step_4`, ..., `step_1`. Thus the correct program in MCK is as follows:

```
step_4 := step_3;
step_3 := step_2;
step_2 := step_1;
```

But what if the dependency graph has loops? Consider this example:

```
1 next(holds(x,a)) <= true(holds(x,a)).
2 next(control(x)) <= true(control(o)).
3 next(control(o)) <= true(control(x)).
```

The first loop is a self loop and does not create any problem, but the second one does pose a problem. The following program does not capture the meaning of lines 2–3 in the above description:

```
control_x := control_o;
control_o := control_x;
```

where the second statement uses a boolean updated by the first.

Our solution is to break the loop by cutting one dependency and then creating a new variable to record the last variable in the new dependency graph. Back to the above example, we can cut the dependency from `control(o)` to `control(x)`, and then use a new variable to give the following correct translation:

```
control_x_old := control_x;
control_x := control_o;
control_o := control_x_old;
```

Put in words, `control_x_old` is used to remember the old value of `control_x`.

There is another more general way to solve this problem: give all ground atoms a new variable (appended with `old`) and use them to record the values of their corresponding part in the beginning of the translation. But this can be computationally expensive for MCK in practice, because the number of variables will be doubled. It will not increase the number of reachable states of the game, but the total state space of the MCK program will be increased exponentially (i.e., 2^n for n new variables). So our above solution by using dependency graphs will be much more efficient in practice.

Specifications

The last step is to encode the temporal and epistemic properties to be verified, using:

```
<specification type> = ... temporal and epistemic formula ...
```

The specification types we will use are "`spec_spr_nested`", "`spec_obs_ctl`" and "`spec_spr_bmc n`", where spr indicates that the model checking algorithm will use synchronous perfect recall semantics, and obs indicates observational semantics, bmc means bounded model checking. The first two algorithm use Ordered Binary Decision Diagram (OBDD) encoding and the second uses SAT encoding. We will explain the difference when we present the experimental results in the next session. Our temporal and epistemic formulas are given in CTLK syntax.

MCK checks a property φ on the initial state of the translated game $\pi(G)$ with specification type x, and then when the computation is done, it returns either 'holds' (i.e., $\pi(G) \models_x \varphi$) or 'fails' (otherwise).

4 Experimental Results

We present some experimental results on two incomplete information games: Monty Hall and Krieg-Tictactoe. The machines have Intel Core i5-2500 Quad CPU 3.3 GHz and 8GB Ram running under GNU Linux OS 2.6.32. The MCK version is 1.0.0.

Monty Hall

Following the method presented in the previous section, we compare the efficiency of two different translations of the Monty Hall game from Fig. 1. The first translation $\pi_1(G_{MT})$ is the straightforward one without optimisation. It contains 43 boolean variables. The second translation $\pi_2(G_{MT})$ is the result of applying the various optimisations given above, resulting in only 28 boolean variables.

The following properties have been checked using MCK on these two translations:

- $\varphi_1 = (\bigwedge_m legal(Candidate, m) \rightarrow K_{Candidate} legal(Candidate, m))$

  This property intuitively means that *'Candidate' knows his legal moves at the current state*. Furthermore, we define $\varphi_2 = AX\varphi_1$ which intuitively means that

'Candidate' knows his legal moves at all next states; $\varphi_3 = AXAX\varphi_1$; and $\varphi_4 = AXAXAX\varphi_1$.

- $\varphi_5 = AXAXAX(terminal \wedge K_{Candidate}terminal)$. This property intuitively means that for all states after three steps, the game is terminal and the candidate knows this.
- $\varphi_6 = \neg AXAXAX(goal(Candidate, 100)) \wedge \neg AXAXAX(goal(Candidate, 0))$. This property intuitively means that it is not always the case that *Candidate* will win (i.e., $goal(Candidate, 100)$ is true) after three steps, nor that he will always lose.
- $\varphi_7 = AFterminal$. This property means the game will eventually reach a terminal state.

We check φ_1 to φ_5 using the spr_nested algorithm associated with *synchronous perfect recall* (spr) semantics because they all involve knowledge, and then check φ_6 and φ_7 using the obs_ctl algorithm associated with the *observational* (obs) semantics because these formulas do not involve knowledge (which reduces the model checking time). For comparison, we also check φ_6 under spr semantics. The following table shows the model checking time (measured in seconds) for these seven formulas. These formulas all hold in the initial state of G_{MT} and MCK returns corrected results.

Translation	φ_1	φ_2	φ_3	φ_4	φ_5	φ_6(spr)	φ_6(obs)	φ_7
$\pi_1(G_{MT})$	6.70	20.63	49.37	129.66	222.06	561.24	3.41	6.47
$\pi_2(G_{MT})$	0.53	3.10	9.01	19.59	17.25	39.32	0.50	0.49

We can see that our second translation needs notably less time than the first translation under both semantics. Also when a formula contains more temporal depth, it tends to need more time. The result on φ_6 shows that the model checking under obs-semantics may need much less time than that of spr-semantics. For φ_7, it cannot be checked under spr-semantics because operator AF is not supported.

Krieg-TicTacToe

We also studied a more complex game called Krieg-TicTacToe, an incomplete information version of TicTacToe. In this game, two players cannot see their opponent's markings, and if one player tries to mark a position that has been occupied by the opponent, then the game master will tell the player that the move is not valid and ask it to try again. The turn-taking and winning conditions remain the same. The GDL-II description of this game (call it G_{KT}) can be found on ggpserver.general-game-playing.de.

Our first translation $\pi_1(G_{KT})$ has 111 boolean variables, and the optimised translation $\pi_2(G_{KT})$ has 70 boolean variables. Both are around six times larger than the translations of the Monty Hall game.

We select a few representative properties:

- $\psi_1 = (\bigwedge_m legal(xplayer, m) \rightarrow K_{xplayer}legal(xplayer, m))$. This property intuitively means that *'xplayer' knows his legal moves at the current state*. Similarly we define $\psi_2 = AX\psi_1$, $\psi_3 = AXAX\psi_1$ and $\psi_4 = AXAXAX\psi_1$.

– $\psi_5 = AXAXcontrol(xplayer)$. This property says that *after two steps xplayer is in control in all the resulting states*. This property would be true for the original TicTacToe due to the turn taking under complete information. But under incomplete information, this is not true anymore. It is because after one step, $oplayer$ has control and she might try an invalid move, in that case, she will be given another chance to select a move for the next step.
– $\psi_6 = AG(tried(xplayer, 1, 1) \rightarrow AXtried(xplayer, 1, 1))$. This property says that it is always the case that if xplayer already tried to mark the position (1,1), then in all the next states, this is still true.

We first check ψ_1 to ψ_4 using the spr_nested algorithm. The following table only shows the model checking time (in seconds) for the second translation $\pi_2(G_{KT})$:

Translation	ψ_1	ψ_2	ψ_3	ψ_4
$\pi_2(G_{KT})$	86.24	1539.24	26782.95	NA

It indicates that the time complexity increases quickly with the depth of the formula. In the case of φ_4 we could not obtain a result within 24 hours. This led us to bounded model checking (BMC) in which the specification is required to be a formula in the so-called universal fragment of a logic. The universal fragment of a logic requires that the negation operator may apply only to atomic propositions, and the modal operators can only be AX, AF, AG, A$\mathcal{U}$ and K_i. Each specification will also be given a bound number n to indicate the depth of the game tree to be checked by MCK. The following table shows the model checking time (in seconds) for both translations:

Translation	$\psi_1(b\,1)$	$\psi_2(b\,2)$	$\psi_3(b\,3)$	$\psi_4(b\,4)$	$\psi_5(b\,3)$	$\psi_6(b\,5)$	$\psi_6(b\,4)$
$\pi_1(G_{KT})$	0.27	1.70	4.40	10.34	3.69	11.87	6.87
$\pi_2(G_{KT})$	4.63	8.69	32.00	588.38	24.33	113.62	48.81

Note that each formula is given a bound when being fed to MCK; the bound n is indicated as $(b\ n)$. It is interesting to see that under BMC, the first translation has a better efficiency now. The main disadvantage of BMC is that it only check the model up to bound n. So if there is no counter example found under bound n, it usually does not mean that no counter example can be found at bound $n + 1$. E.g., formula φ_6 has no counter example under bound 4, but it has a counter example under bound 5.

We can partially answer why a seemingly more optimised translation yields a worse result in BMC. Unlike the OBDDs, SAT algorithms are more sensitive to the complexity of boolean statements which can express complicated relations between booleans, rather than to the number of booleans. In the optimised version $\pi_2(G_{KT})$, we use "*definitions*" to reduce the number of variables but that, on the other hand, increases the complexity of boolean statements.

5 Related Work and Further Research

There are a few papers on reasoning about games in GDL and its extension GDL-II. [8] uses Answer Set Programming for verifying finitely-bounded temporal invariance properties against a given game description by structural induction. [9] extends [8] to deal

with epistemic properties for GDL-II. That formalism restricts on positive-knowledge formulas while the approach in this paper does not have such restriction and can handle more expressive epistemic and temporal formulas. [15] provides a reasoning mechanism for strategic and temporal properties but it is restricted on the original GDL for complete information games. [16] exams the epistemic logic behind GDL-II and in particular shows that the situation at any stage of a game can be characterised by a multi-agent epistemic (i.e., S5-) model. [18], an extension to [15, 16], provides both semantic and syntactic characterisations of GDL-II descriptions in terms of a strategic and epistemic logic, and shows the equivalence of these two characterisations. The current paper does not handle strategies but is more applied than [18] as we can directly using a model checker.

Some other work are related to this paper more generally in terms of planning and model checking. [1] applies symbolic planning to solve parity games equivalent to μ-calculus model checking problems. [2] solves planning problems based on a high-level action language and model checking; and [3] gives automatic plan generation for non-deterministic domains using OBDD (which is also used by MCK). [13] introduces an approach to conformant planning (where the initial situation is not fully known and actions may have non-deterministic effects) by converting such problems into classical planning problems. It is similar to our approach in sprit but the actual formalisms are rather different.

We conclude by pointing out some directions for further research. Our case study on Krieg-TicTacToe suggests that the optimisation we have applied allows us to verify some formulas in a reasonable amount of time but is not yet fully functional for more complex formulas. However a hand-made version of Krieg-TicTacToe (with more abstraction) in MCK does suggest that MCK has no problem to cope with the amount of reachable states of Krieg-TicTacToe. So the question is, what other optimisation techniques can we find for the translation? On the other hand, we would like to investigate how to make MCK language more expressive by allowing n-ary predicates, fixpoints, loops in transition relations. This may result in a more direct translation.

Also there are logics that deal with strategic and epistemic reasoning, so we are interested in generalising this model checking approach to such logics (see [18] for a first theoretical result). Similar to [1–3, 13], we would like to also explore how plans (or strategies) can be generated via model checking for general game playing.

Acknowledgements We thank Xiaowei Huang, Ron van der Meyden and two anonymous reviewers for their helpful comments. This research was supported under Australian Research Council's (ARC) *Discovery Projects* funding scheme (DP 120102023). The second author is the recipient of an Australian Research Council Future Fellowship (FT 0991348) and is also affiliated with the University of Western Sydney.

References

1. Bakera, M., Edelkamp, S., Kissmann, P., Renner, C.D.: Solving μ-calculus parity games by symbolic planning. In: Peled, D., Wooldridge, M. (eds.) MoChArt. Lecture Notes in Computer Science, vol. 5348, pp. 15–33. Springer (2008)

2. Cimatti, A., Giunchiglia, E., Giunchiglia, F., Traverso, P.: Planning via model checking: A decision procedure for ar. In: Proceedings of 4th European Conference on Planning ECP'97. pp. 130–142. Springer-Verlag (1997)
3. Cimatti, A., Roveri, M., Traverso, P.: Automatic obdd-based generation of universal plans in non-deterministic domains. In: Mostow, J., Rich, C. (eds.) AAAI/IAAI. pp. 875–881. AAAI Press / The MIT Press (1998)
4. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Kozen, D. (ed.) Logics of Programs — Proceedings 1981 (LNCS Volume 131). pp. 52–71. Springer-Verlag: Berlin, Germany (1981)
5. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning About Knowledge. The MIT Press: Cambridge, MA (1995)
6. Gammie, P., van der Meyden, R.: MCK: Model checking the logic of knowledge. In: Alur, R., Peled, D. (eds.) Proceedings of the 16th International Conference on Computer Aided Verification (CAV 2004). pp. 479–483. Springer (2004)
7. Genesereth, M., Love, N., Pell, B.: General game playing: Overview of the AAAI competition. AI Magazine 26(2), 62–72 (2005)
8. Haufe, S., Schiffel, S., Thielscher, M.: Automated verification of state sequence invariants in general game playing. Artificial Intelligence Journal 187–188, 1–30 (2012)
9. Haufe, S., Thielscher, M.: Automated verification of epistemic properties for general game playing. In: Proceedings of the 13th International Conference on Principles of Knowledge Representation and Reasoning (KR 2012) (2012)
10. 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 (2006)
11. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems. Springer-Verlag: Berlin, Germany (1992)
12. van der Meyden, R., Gammie, P., Baukus, K., Lee, J., Luo, C., Huang, X.: User manual for mck 0.5.0. Tech. rep., University of New South Wales (2010)
13. Palacios, H., Geffner, H.: Compiling uncertainty away in conformant planning problems with bounded width. J. Artif. Intell. Res. (JAIR) 35, 623–675 (2009)
14. Pnueli, A.: The temporal logic of programs. In: Proceedings of the Eighteenth IEEE Symposium on the Foundations of Computer Science. pp. 46–57 (1977)
15. Ruan, J., van der Hoek, W., Wooldridge, M.: Verification of games in the game description language. Journal Logic and Computation 19(6), 1127–1156 (2009)
16. Ruan, J., Thielscher, M.: The epistemic logic behind the game description language. In: Proceedings of the Conference on the Advancement of Artificial Intelligence (AAAI). pp. 840–845. San Francisco (2011)
17. Ruan, J., Thielscher, M.: Model checking games in GDL-II: the technical report. Tech. Rep. CSE-TR-201219, University of New South Wales (2012)
18. Ruan, J., Thielscher, M.: Strategic and epistemic reasoning for the game description language GDL-II. In: Proceedings of the European Conference on Artificial Intelligence (ECAI 2012) (2012)
19. Schiffel, S., Thielscher, M.: Fluxplayer: A successful general game player. In: Proceedings of the 22nd AAAI Conference on Artificial Intelligence (AAAI-07). pp. 1191–1196. AAAI Press (2007)
20. Thielscher, M.: A general game description language for incomplete information games. In: Proceedings of AAAI. pp. 994–999 (2010)
21. Thielscher, M.: The general game playing description language is universal. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI). pp. 1107–1112. Barcelona (2011)