Model Checking for Reasoning about Incomplete Information Games

Xiaowei Huang[1], Ji Ruan[2], and Michael Thielscher[1]

[1] University of New South Wales, Australia
{xiaoweih,mit}@cse.unsw.edu.au
[2] Auckland University of Technology, New Zealand
jiruan@aut.ac.nz

Abstract. GDL-II is a logic-based knowledge representation formalism used in general game playing to describe the rules of arbitrary games, in particular those with incomplete information. In this paper, we use *model checking* to automatically verify that games specified in GDL-II satisfy desirable temporal and knowledge conditions. We present a systematic translation of GDL-II to a model checking language, prove the translation to be correct, and demonstrate the feasibility of applying model checking tools for GDL-II games by four case studies.

1 Introduction

The general game description language GDL, as the input language for general game-playing systems [7], 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 and non-trivial games. Genesereth *et al.* [7] list a few properties that are necessary for well-formed GDL games, including guaranteed termination and the requirement that all players have at least one legal move in non-terminal states. The introduction of incomplete information 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, and more broadly computer systems [13,3]. The programs are in certain states at each instant, and the correctness of the programs can be expressed as temporal specifications. A good example is the temporal logic formula "$AG\,\neg deadlock$" meaning *the program can never enter a deadlock state*. Epistemic logics, on the other hand, are formalisms for reasoning about knowledge and beliefs. Their 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 (i.e., an execution sequence) of the protocol. Fagin *et al.* [4] give a comprehensive study on epistemic logic for multi-agent interactions.

Ruan and Thielscher [16] have shown that the situation at any stage of a game in GDL-II can be characterized by a multi-agent epistemic (i.e., S5-) model. Yet, this result only provides a static characterization of what players know (and don't know) at a certain stage. Our paper extends this recent analysis with a temporal dimension, and also provides a practical method for verifying temporal and epistemic properties using a model checker MCK [5]. We present a systematic translation from GDL-II into equivalent specifications in the model specification language of MCK. Verifying a property φ for a game description G is then equivalent to checking whether φ holds for the translation $\mathtt{trs}(G)$. The latter can be automatically checked in MCK.

The paper is organized as follows. Section 2 introduces GDL-II and MCK. Section 3 presents the translation along with possible optimizations and a proof of its correctness. Experimental results for four case studies are given in Section 4. The paper concludes with a discussion of related work and directions for further research.

2 Background

Game Description Language 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 players' knowledge thereof, 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. 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. Meanwhile, GDL and its successor GDL-II have a precise semantics and are fully machine-processable.

The GDL-II rules in Fig. 1 formalize 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. Highlighted are the pre-defined *keywords* of GDL-II. The intuition behind the rules is as follows. Line 1 introduces the players' names (the game host is modelled by the pre-defined role called `random`). Line 2 defines 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 hide the car (line 3) and, simultaneously, the candidate chooses a door (line 7); in step 2, `random` opens a door that is not the one that holds the car nor the chosen one (lines 4–5); finally, the candidate can either stick to their earlier choice (noop) or switch to the other, yet unopened door (line 9 and 10, respectively). The candidate's only percept throughout the game is to see the door opened by the host (line 14) and where the car is after step 3 (line 15). 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`).

GDL-II is suitable for describing synchronous n-player games with randomness and imperfect information. Valid game descriptions must satisfy certain syntactic restrictions, which ensure that all necessary inferences "$\vdash$" in Definition 1 below are finite and decidable; see [12] for details. In the following, we assume the reader to be familiar with basic notions and notations of logic programming, as can be found in e.g. [11].

A state transition system can be obtained from a valid GDL-II game description by using the notion of the *stable models* of logic programs with negation [6]. The syntactic restrictions in GDL-II ensure that all logic programs we consider have a *unique* and *finite* stable model [12,20]. Hence, the state transition system for GDL-II has a finite set of players, finite states, and finitely many legal moves in each state. By $G \vdash p$ we denote that ground atom p is contained in the unique stable model, denoted as $\mathtt{SM}(G)$, for a stratified set of clauses G. In the following definition of the game semantics for GDL-II, *states* are identified with the set of ground atoms that are true in them.

```
role(candidate). role(random).
init(closed(1)). init(closed(2)). init(closed(3)). init(step(1)).
legal(random,hide_car(?d))  <= true(step(1)), true(closed(?d)).
legal(random,open_door(?d)) <= true(step(2)), true(closed(?d)),
                               not true(car(?d)), not true(chosen(?d)).
legal(random,noop)          <= true(step(3)).
legal(candidate,choose(?d)) <= true(step(1)), true(closed(?d)).
legal(candidate,noop)       <= true(step(2)).
legal(candidate,noop)       <= true(step(3)).
legal(candidate,switch)     <= true(step(3)).
next(car(?d))  <= does(random,hide_car(?d)).
...
next(step(4))  <= true(step(3)).
sees(candidate,?d) <= does(random,open_door(?d)).
sees(candidate,?d) <= true(step(3)), true(car(?d)).
terminal   <= true(step(4)).
goal(candidate,100) <= true(chosen(?d)), true(car(?d)).
goal(candidate,  0) <= true(chosen(?d)), not true(car(?d)).
```

Fig. 1. G_{MH} - a GDL-II description of the Monty Hall game adapted from [21]

Definition 1. *[20] Let G be a valid GDL-II description. The state transition system* $(R, s_0, \tau, l, u, \mathcal{I}, \Omega)$ *of G is given by*

- *roles* $R = \{i \mid \texttt{role}(i) \in \text{SM}(G)\}$*;*
- *initial position* $s_0 = \text{SM}(G \cup \{\texttt{true}(f) \mid \texttt{init}(f) \in \text{SM}(G)\})$*;*
- *terminal positions* $\tau = \{s \mid \texttt{terminal} \in s\}$*;*
- *legal moves* $l = \{(i, a, s) \mid \texttt{legal}(i, a) \in s\}$*;*
- *state update function* $u(M, s) = \text{SM}(G \cup \{\texttt{true}(f) \mid \texttt{next}(f) \in \text{SM}(G \cup s \cup M)\})$*, for all* joint *legal moves M (i.e., where each role in R takes one legal move);*
- *information relation* $\mathcal{I} = \{(i, M, s, p) \mid \texttt{sees}(i, p) \in \text{SM}(G \cup s \cup M)\}$*;*
- *goal relation* $\Omega = \{(i, n, s) \mid \texttt{goal}(i, n) \in s\}$.

Note that a state s contains all ground atoms that are true in the state, which includes the "fluent atoms" $\texttt{true}(f)$ in, respectively, $\{\texttt{true}(f) \mid \texttt{init}(f) \in \text{SM}(G)\}$ (for the initial state) and $\{\texttt{true}(f) \mid \texttt{next}(f) \in \text{SM}(G \cup s \cup M)\}$ (for the successor state of s and M), and all other atoms that can be derived from G *and* these fluent atoms.

Different runs of a game can be described by *developments*, which are sequences of states and moves by each player up to a certain round. A player *cannot distinguish* two developments if the player has made the same moves and perceptions in both of them.

Definition 2. *[20] Let* $(R, s_0, \tau, l, u, \mathcal{I}, \Omega)$ *be the state transition system of a GDL-II description G, then a* development *δ is a finite sequence*

$$\langle s_0, M_1, s_1, \ldots, s_{d-1}, M_d, s_d \rangle$$

such that for all $k \in \{1, \ldots, d\}$ *(*$d \geq 0$*),* M_k *is a joint move and* $s_k = u(M_k, s_{k-1})$*.*

A terminal development is a development such that the last state is a terminal state, i.e., $s_d \in \tau$*. The* length *of a development δ, denoted as len(δ), is the number of states in δ. By* $M(i)$ *we denote agent i's move in the joint move M. Let* $\delta|_k$ *be the prefix of δ up to length* $k \leq$ *len(δ).*

A player $i \in R \backslash \{\texttt{random}\}$ cannot distinguish *two developments* $\delta = \langle s_0, M_1, s_1, \ldots \rangle$ *and* $\delta' = \langle s_0, M_1', s_1' \ldots \rangle$ *(written as* $\delta \sim_i \delta'$*) iff len(δ) = len(δ′) and for any* $1 \leq k \leq$ *len(δ)−1:* $M_k(i) = M_k'(i)$*, and* $\{p \mid (i, M_k, s_{k-1}, p) \in \mathcal{I}\} = \{p \mid (i, M_k', s_{k-1}', p) \in \mathcal{I}\}$.

Model Checker MCK. In this paper, we will use MCK (for: "Model Checking Knowledge"), which is a model checker for temporal and knowledge specifications [5]. The overall setup of MCK supposes a number of agents acting in an environment. This is modelled by an *interpreted system*, formally defined below, where agents perform actions according to protocols. Actions and the environment may only be 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* and is used in this paper. In the temporal dimension, the specification formulas may describe the evolution of the system along a single computation, i.e., use linear time temporal logic; or they may describe the branching structure of all possible computations, i.e., use branching time or computation tree logic. We give the basic syntax of Computation Tree Logic of Knowledge (CTL*K$_n$).

Definition 3. *The language of CTL*K$_n$ (with respect to a set of atomic propositions Φ), is given by the following grammar:*

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \psi \mid A\varphi \mid X\varphi \mid \varphi \mathcal{U} \psi \mid K_i\varphi.$$

The other logic constants and connectives $\top, \bot, \vee, \rightarrow$ are defined as usual. In addition, $F\varphi$ (read: finally, φ) is defined as $\top \mathcal{U} \varphi$, and $G\varphi$ (read: globally, φ) as $\neg F \neg\varphi$.

The semantics of the logic can be given using *interpreted systems* [4]. Let S be a set, which we call the set of environment states, and Φ be the set of atomic propositions. A *run* over environment states S is a function $r : \mathbf{N} \rightarrow S \times L_1 \times \ldots \times L_n$, where each L_i is called the set of *local states of agent* i. These local states are used to concretely represent the information on the basis of which agent i computes its knowledge. Given run r, agent i, and time m, we write $r_i(m)$ for the $(i+1)$-th component (in L_i) of $r(m)$, and $r_e(m)$ for the first component (in S). An *interpreted system* over environment states S is a tuple $\mathcal{IS} = (\mathcal{R}, \pi)$, where $\mathcal{R}$ is a set of runs over environment states S, and $\pi : \mathcal{R} \times \mathbf{N} \rightarrow \mathcal{P}(\Phi)$ is an interpretation function. A *point* of $\mathcal{IS}$ is a pair (r, m) where $r \in \mathcal{R}$ and $m \in \mathbf{N}$.

Definition 4. *Let $\mathcal{IS}$ be an interpreted system, (r, m) be a point of $\mathcal{IS}$, and φ be a CTL*K$_n$ formula.* Semantic entailment $\models$ *is defined inductively as follows:*

- $\mathcal{IS}, (r, m) \models p$ *iff* $p \in \pi(r, m)$;
- *the propositional connectives* $\neg, \wedge$ *are defined as usual;*
- $\mathcal{IS}, (r, m) \models A\varphi$ *iff* $\forall r' \in \mathcal{R}$ *with* $r'(k) = r(k)$ *and* $\forall k \in [0..m]$, *we have* $\mathcal{IS}, (r', m) \models \varphi$;
- $\mathcal{IS}, (r, m) \models X\varphi$ *iff* $\mathcal{IS}, (r, m+1) \models \varphi$;
- $\mathcal{IS}, (r, m) \models \varphi \mathcal{U} \psi$ *iff* $\exists m' \geq m$ *s. t.* $\mathcal{IS}, (r, m') \models \psi$ *and* $\mathcal{IS}, (r, k) \models \varphi$ *for all* $k \in [m..m')$;
- $\mathcal{IS}, (r, m) \models K_i\varphi$ *iff* $\forall (r', m')$ *with* $r_i(m) = r'_i(m')$, *we have* $\mathcal{IS}, (r', m') \models \varphi$.

Syntax of MCK Input Language. An MCK description consists of an environment and one or more agents. An environment model represents how states of the environment are affected by the actions of the agents. A protocol describes how an agent selects an action under a certain environment.

Formally, an *environment model* is a tuple $\mathcal{M}_e = (Agt, Acts, Var_e, Init_e, Prog_e)$ where Agt is a set of agents, $Acts$ is a set of actions available to the agents, Var_e is a set of environment variables, $Init_e$ is an initial condition, in the form of a boolean formula over Var_e, and $Prog_e$ is a standard program for the environment e to be defined below.

Let $ActVar(\mathcal{M}_e) = \{i.a \mid i \in Agt,\ a \in Acts\}$ be a set of *action variables* generated for each model $\mathcal{M}_e$. An atomic statement in $Prog_e$ is of the form $x := expr$, where $x \in Var_e$ and $expr$ is an expression over $Var_e \cup ActVar(\mathcal{M}_e)$.

A *protocol for agent* i in $\mathcal{M}_e$ is a tuple $Prot_i = (PVar_i, OVar_i, Acts_i, Prog_i)$, where $PVar_i \subseteq Var_e$ is a set of *parameter variables*, $OVar_i \subseteq PVar_i$ is a set of *observable variables*, $Acts_i \subseteq Acts$, and $Prog_i$ is a standard program. An atomic statement in $Prog_i$ is either of the form $x := expr$, or of the form $\ll a \gg$ with $a \in Acts_i$.

A *standard program* over a set Var of variables and a set A of atomic statements is either the terminated program ϵ or a sequence P of the form $stat_1 ; \ldots ; stat_m$, where each $stat_k$ is a simple statement and ';' denotes sequential composition.

Simple statement $stat_k$ can be *atomic statements* in A; or *nondeterministic branching statements* of the form: if $g_1 \rightarrow a_1$ [] $\ldots$ [] $g_m \rightarrow a_m$ fi; or *nondeterministic iteration statements* of the form: do $g_1 \rightarrow a_1$ [] $\ldots$ [] $g_m \rightarrow a_m$ od, where each a_k is an atomic statement in A and each *guard* g_k is a boolean expressions over Var.

Each atomic statement a_k can be executed only if its corresponding guard g_k holds in the current state. If several guards hold simultaneously, one of the corresponding actions is selected nondeterministically. The last guard g_m can be "$otherwise$", which is shorthand for $\neg g_1 \wedge \cdots \wedge \neg g_{m-1}$. An *if*-statement executes once but a *do*-statement can be repeatedly executed.

Semantics of MCK Input Language. Based on a set of agents running protocols in the context of a given environment, we can define an interpreted system as follows.

Definition 5. *A* system model $\mathcal{S}$ *is a pair* $(\mathcal{M}_e, Prot)$ *where* $\mathcal{M}_e = (Agt, Acts, Var_e, Init_e, Prog_e)$ *and Prot a joint protocol with* $Prot_i = (PVar_i, OVar_i, Acts_i, Prog_i)$ *for all* $i \in Agt$.

Let a state with respect to $\mathcal{S}$ *be an assignment s over the set of variables* Var_e. *A* transition model *over* $\mathcal{S}$ *is* $\mathcal{M}(\mathcal{S}) = (S, I, \{O_i\}_{i \in Agt}, \rightarrow, V)$, *where S is the set of states of* $\mathcal{S}$; *I is the set of initial states s such that* $s \models Init_e$; $O_i(s) = s \upharpoonright OVar_i$ *is the partial assignment given on the observable variables of agent i,* $\rightarrow$ *is a transition relation on* $S \times S$;[1] *and a valuation function V is given by: for any boolean variable x,* $x \in V(s)$ *iff* $s(x) = true$. [2]

An infinite sequence of states $s_0 s_1 \ldots$ *is an* initialized computation of $\mathcal{M}(\mathcal{S})$ *if* $s_0 \in I$, $s_k \in S$ *and* $s_k \rightarrow s_{k+1}$ *for all* $k \geq 0$. *An* **interpreted system over** $\mathcal{S}$ *is* $\mathcal{IS}(\mathcal{S}) = (\mathcal{R}, \pi)$, *where* $\mathcal{R}$ *is the set of runs such that each run r corresponds to an initialized computation* $s_0 s_1 \ldots$ *with* $r_e(m) = s_m$, *and* $r_i(m) = O_i(s_0) O_i(s_1) \ldots O_i(s_m)$; *and* $\pi(r, m) = V(s_m)$.

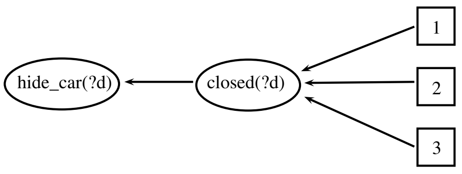
[1] More precisely, $s \rightarrow s'$ if s' is obtained by executing the parallel program $Prog_e \,||_{i \in Agt}\, Prog_i$ on s; see [14] for details.

[2] For simplicity, we assume x to be boolean; this can be easily extended to enumerated type variables: Suppose x is a variable with type $\{e_1, \ldots, e_m\}$, then use m booleans $x.e_1, \ldots, x.e_m$ such that $x.e_k \in V(s)$ iff $s(x) = e_k$.

3 Translation from GDL-II to MCK

Our main contribution in this paper is a systematic translation from a GDL-II description G into an MCK description $\mathtt{trs}(G)$. The translation is provably correct in that the game model derived from G using the semantics of GDL-II satisfies the exact same formulas as the model that is derived from $\mathtt{trs}(G)$ using the semantics of MCK. This will be formally proved later in this section. We use the GDL-II description of the Monty Hall game from Fig. 1, denoted as G_{MH}, to illustrate the whole process. The translation $\mathtt{trs}$ can be divided into the following steps.

Preprocessing. The first step is to obtain a variable-free (i.e., ground) version of the game description G. We can compute the domains, or rather supersets thereof, for all predicates and functions of G by generating a domain 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. E.g., lines 2–3 in G_{MH} give us the domain graph as above, from which it can be seen that the arguments of both $\mathtt{closed}()$ and $\mathtt{hide\_car}()$ range over $\{1, 2, 3\}$.



Once we have computed the domains, we instantiate all the variables in G to obtain all ground atoms, e.g., $\mathtt{true(closed(1))}$, $\mathtt{legal(random, hide\_car(1))}$, etc. Our following translation operates on an equivalent variable-free version of G, which for convenience we still refer to as G.

Deriving Environment Variables. This step derives all the environment variables Var_e. Let $\mathtt{AT}$ be the set of ground atoms in G. Define the following subsets of $\mathtt{AT}$ according to the keywords: $\mathtt{AT_t} = \{\mathtt{h} \in \mathtt{AT} \mid \mathtt{h} = \mathtt{true(p)}\}$, $\mathtt{AT_n} = \{\mathtt{h} \in \mathtt{AT} \mid \mathtt{h} = \mathtt{next(p)}\}$, $\mathtt{AT_d} = \{\mathtt{h} \in \mathtt{AT} \mid \mathtt{h} = \mathtt{does(i, a)}\}$, $\mathtt{AT_i} = \{\mathtt{h} \in \mathtt{AT} \mid \mathtt{h} = \mathtt{init(p)}\}$, $\mathtt{AT_s} = \{\mathtt{h} \in \mathtt{AT} \mid \mathtt{h} = \mathtt{sees(r, p)}\}$, and $\mathtt{AT_l} = \{\mathtt{h} \in \mathtt{AT} \mid \mathtt{h} = \mathtt{legal(r, p)}\}$. Let p be obtained by replacing '(' and ',' with '_' and by removing ')' in a ground atom $\mathtt{p}$. Define t as follows:

- $t(\mathtt{init(p)}) = p$, $t(\mathtt{true(p)}) = p\_old$ and $t(\mathtt{next(p)}) = p$;
- $t(\mathtt{does(i, a)}) = did\_i$;
- $t(\mathtt{p}) = p$ for all $\mathtt{p} \in \mathtt{AT} \setminus (\mathtt{AT_i} \cup \mathtt{AT_t} \cup \mathtt{AT_n} \cup \mathtt{AT_d})$.

Note that the ground atoms with keywords $\mathtt{legal}$, $\mathtt{terminal}$, $\mathtt{goal}$ are all in $\mathtt{AT} \setminus (\mathtt{AT_i} \cup \mathtt{AT_t} \cup \mathtt{AT_n} \cup \mathtt{AT_d})$. As an example, $t(\mathtt{sees(i, a)}) = sees\_i\_a$ and $t(\mathtt{legal(i, a)}) = legal\_i\_a$. The set of environment variable Var_e is then $\{t(\mathtt{p}) \mid \mathtt{p} \in \mathtt{AT}\}$. For convenience, we denote $t(A)$ as $\{t(x) \mid x \in A\}$.

The type of each variable $did\_i \in t(\mathtt{AT_d})$ is the set of legal moves of agent i plus two additional moves, $\mathtt{INIT}$ and $\mathtt{STOP}$, that do not appear in G, i.e., $\{\mathtt{a} \mid \mathtt{legal(i, a)} \in \mathtt{AT}\} \cup \{\mathtt{INIT}, \mathtt{STOP}\}$. The type of variables in $Var_e \setminus t(\mathtt{AT_d})$ is $\mathtt{Bool}$.

Initial Condition. This step specifies the environment initial condition $Init_e$, which is an assignment over Var_e. By using the semantics of G and $\mathtt{AT_i}$, we first compute the initial state s_0 (see Definition 1). Then for any $\mathtt{p} \in \mathtt{AT_i}$, we add boolean expression

"$t(\mathtt{p}) == \mathit{true}$" to Init_e as a conjunct; and for all $\mathit{did\_i} \in t(\mathtt{AT_d})$, we add "$\mathit{did\_i} ==$ INIT". For the rest, add "$t(\mathtt{p}) == \mathit{true}$" if $\mathtt{p} \in s_0$, and "$t(\mathtt{p}) == \mathit{false}$" if $\mathtt{p} \notin s_0$.

Agent Protocols. This step specifies the agents and their protocols. The names of the agents are read off the $\mathtt{role}()$ facts. Let $\mathit{Prot}_i = (\mathit{PVar}_i, \mathit{OVar}_i, \mathit{Acts}_i, \mathit{Prog}_i)$ be the protocol of agent i, such that $\mathit{PVar}_i = \mathit{Var}_e$, $\mathit{OVar}_i = \{\mathit{sees\_i\_p} \mid \mathit{sees\_i\_p} \in t(\mathtt{AT_s})\} \cup \{\mathit{did\_i}\}$ includes all the variables representing i's percept and i's move, and $\mathit{Acts}_i = \{\mathtt{a} \mid \mathtt{legal(i,a)} \in G\}$ includes all the legal moves of agent i. Note that Acts_i does not include the two special moves in the protocol. The last component Prog_i is a standard program of the following format:

```
begin do neg terminal ->
    if legal_i_a1 -> <<a1>> [] legal_i_a2 -> <<a2>> [] ...
    fi od end
```

This program intuitively means that if the current state is not terminal, then a legal move is selected non-determinstically by i. The statements between $\mathtt{do} \cdots \mathtt{od}$ are executed repeatedly. The variables inside $<<>>$ represent moves.

State Transition. This step specifies the environment program Prog_e. Each environment variable is updated in correspondence with the rules in G. The main task is *to translate these rules into MCK statements in a correct order*. In GDL-II, the order of the rules does not matter as the stable model semantics [12,20] always gives the same unique model, but MCK uses the imperative programming style in which the order of the statements does matter; e.g., executing "$x := 0; x := 1;$" results in a different state than "$x := 1; x := 0;$". To take care of the order, we separate the program Prog_e into three parts.

The first part updates the variables in $t(\mathtt{AT_d})$ using the following template (for i):

```
if i.a1   -> did_i := a1 [] i.a2   -> did_i := a2 []
    ...  otherwise -> did_i := STOP
fi;
```

The second part of Prog_e updates the variables in $t(\mathtt{AT_t})$ and $t(\mathtt{AT_n} \cup \mathtt{AT_s})$. For all $\mathit{p\_old} \in t(\mathtt{AT_t})$, an atomic statement of the form $\mathit{p\_old} := p$ is added to ensure that the value of p is remembered before it is updated. For any atom $h \in t(\mathtt{AT_n} \cup \mathtt{AT_s})$, suppose $h = t(\mathtt{h})$ and $\mathit{Rules}(\mathtt{h})$ is the set of rules in G with head $\mathtt{h}$:

$$\begin{array}{l} r_1 : \mathtt{h} \Leftarrow b_{11}, \cdots, b_{1j} \\ \ldots \quad \ldots \\ r_k : \mathtt{h} \Leftarrow b_{k1}, \cdots, b_{kj} \end{array}$$

where b_{xy} is a literal over $\mathtt{AT}$. Define a translation tt as follows:

- $tt(\mathtt{does(i,a)}) = \mathit{did\_i} == a$;
- $tt(\mathtt{not\ x}) = \mathit{neg}\ tt(\mathtt{x})$; and other cases are same as t.

The translation of $\mathit{Rules}(\mathtt{h})$ has the following form:

$$h := (tt(b_{11}) \wedge \cdots \wedge tt(b_{1j})) \ \vee \ \cdots \ \vee \ (tt(b_{k1}) \wedge \cdots \wedge tt(b_{kj}))$$

This simplifies to $h := \mathit{true}$ if one of the bodies is empty. Essentially, this is a form of the standard Clark Completion [2], which captures the idea that $\mathtt{h}$ will be false in the next state unless there is a rule to make it true. The statements with $t(\mathtt{AT_t})$ should be given before those with $t(\mathtt{AT_n} \cup \mathtt{AT_s})$.

The third part deals the variables in $t(\mathtt{AT} \setminus (\mathtt{AT_t} \cup \mathtt{AT_n} \cup \mathtt{AT_s} \cup \mathtt{AT_d} \cup \mathtt{AT_i}))$. Pick such an atom h and take $Rules(\mathtt{h})$. The literals in the body of these rules are translated differently from the last case, as h refers to the current instead of the next state. Define a new translation tt' as follows:

– $tt'(\mathtt{true(p)}) = p$ and all other cases are identical to tt.

The translation of $Rules(\mathtt{h})$ is similar to the above by replacing tt by tt'. The statements in the third part are ordered according to the dependency graph. If $\mathtt{h}'$ depends on h, then the statement of $tt'(\mathtt{h})$ must appear before that of $tt'(\mathtt{h}')$. The fact that GDL rules are stratified ensures that a desirable order can always be found.

Optimizations. The above translation can be further optimized to make the model checking more efficient by reducing the number of variables.

(1) Using definitions. The variables in $t(\mathtt{AT} \setminus (\mathtt{AT_t} \cup \mathtt{AT_n} \cup \mathtt{AT_d} \cup \mathtt{AT_i}))$ (refer to the third part of the State Transition step) can be represented as definitions to save memory space for variables. The assignment statement $h := expr$ is swapped with definition *define* $h = expr$. MCK replaces h using the boolean expression $expr$ during its preprocessing stage, so h does not occupy memory during the main stage.

(2) Removing static atoms. We distinguish three special kinds of atoms in GDL-II: those (a) appearing in the rules with empty bodies, (b) never appearing in the heads of rules, (c) only appearing in the rules with (a) and (b). Under the GDL-II semantics, atoms in (a) are always true, those in (b) are always false, and those in (c) do not change their value during gameplay. Therefore we can replace them universally with their truth values. E.g., consider the following rules:

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

Both $\mathtt{succ(1,2)}, \mathtt{succ(2,3)}$ are always true, so we replace them using their truth values. Then we can further simplify this by removing the "*true*" conjuncts universally (and by removing the rules with a "*false*" conjunct in the body):

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

(3) Converting booleans to typed variables. The atoms in $\mathtt{AT} \setminus \mathtt{AT_d}$ are translated to booleans in our non-optimized translation. There often are sets of booleans B such that at each state exactly one of them is true. We can then convert the booleans in B into one single variable v_B with the type $\{b_1, \ldots, b_{|B|}\}$, where $|B|$ is the size of B. This results in a logarithmic space reduction on B: $2^{|B|}$ is reduced to $|B|$. Reusing the example just discussed, we can create a variable v_{step} with type $\{1, 2, 3\}$.

Translation Correctness

The above completes the translation from G to $\mathsf{trs}(G)$. As our main theoretical result, we show as follows that our translation is correct: first the game model derived from a GDL-II description G is proved to be isomorphic to the interpreted system that is derived from its translation $\mathsf{trs}(G)$, then a CTL*K$_n$ formula is shown to have an equivalent interpretation (i.e., the same truth value) over these two models.

We first extend the concept of finite developments in Definition 2 to infinite ones.

Definition 6 (Infinite Developments and GDL-II Models). *Let $\langle R, s_0, t, l, u, \mathcal{I}, g\rangle$ be the state transition system of a game description G, and $\delta = \langle s_0, M_1, s_1, \ldots, M_d, s_d\rangle$ a finite terminal development of G, then an* infinite extension *of δ is an infinite sequence $\langle s_0, M_1, s_1, \ldots, M_d, s_d, M_{d+1}, s_{d+1}, \ldots\rangle$ such that M_{d+k} is the joint move where all players take a special move* `STOP` *and $s_{d+k} = s_d$ for all $k \geq 1$.*

Given a GDL-II description G, the game model $\mathsf{GM}(G)$ *is a tuple $(D, \{\sim_i \mid i \in Agt\})$, where D is the set of infinite developments δ such that either δ is an infinite development without terminal states, or δ is an infinite extension of a finite terminal development; and $\sim_i$ is agent i's indistinguishability relation defined on the finite prefixes of $\delta|_k$ as in Definition 2.*

For a given δ, let $\delta(k)$ denote the k-th state s_k; $\delta(k)^M$ the k-th joint move M_k; and (δ, k) the pair (M_k, s_k).

Definition 7 (Isomorphism). *Let* $\mathsf{GM} = (D, \{\sim_i \mid i \in Agt\})$ *be a game model and $\mathcal{IS} = (\mathcal{R}, \pi)$ an interpreted system.* GM *is* isomorphic *to $\mathcal{IS}$ if there is a bijection* w *between the ground atoms of* GM *and the atomic propositions of $\mathcal{IS}$, and a bijection* z *between D and $\mathcal{R}$ satisfying the following:* $\mathsf{z}(\delta) = r$ *iff for any ground atom* $\mathtt{p}$: $\mathtt{p} \in \delta(k)$ *iff* $\mathsf{w}(\mathtt{p}) \in \pi(r, k)$*, and* $\mathtt{does}(i, a) \in \delta(k)^M$ *iff $did\_i == a$ is true in (r, k).*

Intuitively, z associates a point (δ, k) in a development to a point (r, k) in a run such that they coincide in the interpretation of basic and move variables. The following proposition is the first step in showing the correctness of our translation.

Proposition 1. *Given a GDL-II description G, let* trs *be the translation from GDL-II to MCK, then the game model* $\mathsf{GM}(G)$ *is isomorphic to the interpreted system* $\mathcal{IS}(\mathsf{trs}(G))$.

For the technical details of the proof we must refer to [17].

Let w be a bijection from the set of ground atoms of G to the set of atomic propositions of CTL*K$_n$ and w^{-1} be its inverse. The semantics of CTL*K$_n$ over GDL-II Game Models can be given as relation $\mathsf{GM}(G), (\delta, m) \models \varphi$ in analogy to the semantics of CTL*K$_n$ over interpreted systems; e.g., $\mathsf{GM}(G), (\delta, m) \models p$ iff $\mathsf{w}^{-1}(p) \in \delta(m)$, and $\mathsf{GM}(G), (\delta, m) \models K_i\varphi$ iff for all states (δ', m') of $\mathsf{GM}(G)$ that satisfy $\delta|_m \sim_j \delta'|_{m'}$ we have $\mathsf{GM}(G), (\delta', m') \models \varphi$.

The following proposition then shows that checking φ against a game model of G is equivalent to checking φ against the interpreted system of $\mathsf{trs}(G)$.

Proposition 2. *Given a GDL-II description G, let* trs *be the translation from GDL-II to MCK; φ a CTL*K$_n$ formula over the set of atomic propositions in* $\mathsf{trs}(G)$*; and* w, z *the bijections from the isomorphism between* $\mathsf{GM}(G)$ *and* $\mathcal{IS}(\mathsf{trs}(G))$ *then:*

$$\mathsf{GM}(G), (\delta, m) \models \varphi \;\; \textit{iff} \;\; \mathcal{IS}(\mathsf{trs}(G)), (\mathsf{z}(\delta), m) \models \varphi.$$

This follows from Proposition 1 by an induction on the structure of φ and completes the proof of our main result.

Our optimization techniques do not affect the isomorphism. So we can follow a similar argument as Proposition 1 and 2 to show that the optimized translation is correct.

4 Experimental Results

We present experimental results on four GDL-II games from the repository at *general-game-playing.de*: Monty Hall (MH), Krieg-TicTacToe (KTTT), Transit, and Meier. Same games were also used in Haufe and Thielscher [9]. MCK (v1.0.0) runs on Intel 3.3 GHz CPU and 8GB RAM with GNU Linux 2.6.32.

Temporal and epistemic specifications. The temporal logic formulas can be used to specify the *objective* aspects of a game. The following three properties represent the basic requirements from [7]. (Let $Legal_i$ and $Goal_i$ be the set of legal moves and goals of i respectively.)

$$AF\ terminal \tag{1}$$

$$AG(\neg terminal \to \bigwedge_{i\in Agt} \bigvee_{p\in Legal_i} p) \tag{2}$$

$$\bigwedge_{i\in Agt} \neg AG \neg goal\_i\_100 \tag{3}$$

Property (1) says that the game always terminates. Property (2) expresses *playability*: at every non-terminal state, each player has a legal move. Property (3) expresses *fairness*: every player has a chance to win, i.e., to eventually achieve the maximal goal value 100. These properties apply both to GDL and GDL-II games. The next three properties concern the *subjective* views of the players under incomplete-information situations, hence are specific to GDL-II games.

$$\bigwedge_{i\in Agt} G(terminal \to K_i terminal) \tag{4}$$

$$\bigwedge_{i\in Agt} G\,(\neg terminal \to \bigwedge_{p\in Legal_i} (K_i p \vee K_i \neg p)) \tag{5}$$

$$\bigwedge_{i\in Agt} G\,(terminal \to \bigwedge_{p\in Goal_i} (K_i p \vee K_i \neg p)) \tag{6}$$

Property (4) says that once the game has terminated, all players know this. Property (5) says that any player always knows its legal moves in non-terminal states; and property (6) says that in a terminal state, all players know their outcome.

φ	MH	KTTT	Transit	Meier	Meier′	φ	MH	KTTT	Transit	Meier	Meier′
(1)	0.47	1864.81	12.17	6.41	8079.52	**(4)**	0.60	22847.06	14.91	7.00	NA
(2)	0.48	3528.14	7.54	9.75	13192.91	**(5)**	0.56	22643.12	14.39	23.28	NA
(3)	0.67	303.04	11.02	17.06	15056.29	**(6)**	0.43	5498.03	45.15	11.01	NA

The table above shows the runtimes (in seconds) on five translations. The first four translations use all three optimization techniques on the four games. The last translation, Meier′, is partially optimized with the third technique applied only for the variables in $t(\mathtt{AT_s})$. As a consequence, Meier′ uses 126 booleans for what in the fully optimized Meier is represented by 4 enumerated type variables of a size equivalent to about 22 booleans, i.e., the state space of Meier is only $(1/2)^{104}$ of the state space of Meier′. The time is measured in seconds and "NA" indicates that MCK did not return a result after

10 hours. A comparison of the two translations of Meier shows that our optimization can be very effective. Somehow surprisingly, the result shows that the game Meier is not well-formed as it does not satisfy property (1). The last three properties were also checked by Haufe and Thielscher [9], but for Transit, their approach could not prove or disprove these properties; in contrast, our approach obtains the results fully. Note that although we only show the experiment results for four games, our approach is not a specialised solution for these four games only. It is general enough to deal with all GDL and GDL-II games.

5 Related Work and Further Research

There are a few papers on reasoning about games in GDL and GDL-II. Haufe *et al.* [8] use Answer Set Programming for verifying temporal invariance properties against a given game description by structural induction. Haufe and Thielscher [9] extend [8] to deal with epistemic properties for GDL-II. Their approach is restricted to positive-knowledge formulas unlike ours, which can handle more expressive epistemic and temporal formulas.

Ruan *et al.* [15] provide a reasoning mechanism for strategic and temporal properties but restricted to the original GDL for complete information games. Ruan and Thielscher [16] examine the epistemic logic behind GDL-II and in particular show that the situation at any stage of a game can be characterized by a multi-agent epistemic (i.e., S5-) model. Ruan and Thielscher [18] provide both semantic and syntactic characterizations of GDL-II descriptions in terms of a strategic and epistemic logic, and show the equivalence of these two characterizations. The current paper does not handle strategies but is able to provide practical results by using a model checker.

Kissmann and Edelkamp [10] instantiate GDL descriptions and utilise BDDs to construct a symbolic search algorithm to solve single- and two-player turn-taking games with complete information. This is related to our work in the sense that we also do an instantiation of GDL descriptions and uses the BDD-based symbolic model checking algorithms of MCK to verify properties. But our approach is more general and in particular handles games with incomplete information.

Other existing work is related to our paper in that they too deal with declarative languages. Chang and Jackson [1] show the possibility of embedding declarative relations and expressive relational operators into a standard CTL symbolic model checker. Whaley *et al.* [22] propose to use Datalog (which GDL is based upon) with Binary Decision Diagrams (BDDs) for program analysis.

We conclude by pointing out some directions for further research. Firstly our results suggest that the optimization we have applied allows us to verify some formulas quickly, but it is still difficult to deal with a game like Blind TicTacToe. However a hand-made version of this game (with more abstraction) in MCK does suggest that MCK has no problem to cope with the number of reachable states in this game. So the question is, what other optimization techniques can we find for the translation? Secondly, we would like to investigate how to make MCK language more expressive by allowing declarative relations such as shown in [1]. Our current translation maps GDL-II to MCK's input, and MCK internally encodes that into BDDs for symbolic checking. So a more direct map from GDL-II to BDDs may result in a significant efficiency gain. Thirdly, we want to explore the use of bounded model checking as MCK has implemented some model checking algorithms for this.

Acknowledgements. We thank the anonymous reviewers for their helpful comments. This research was supported by the Australian Research Council (ARC; project DP 120102023). Michael Thielscher is the recipient of an ARC Future Fellowship (FT 0991348). He is also affiliated with the University of Western Sydney. Most of the work was completed while Ji Ruan was working at the University of New South Wales.

References

1. Chang, F.S.H., Jackson, D.: Symbolic model checking of declarative relational models. In: Osterweil, L.J., Rombach, H.D., Soffa, M.L. (eds.) ICSE, pp. 312–320. ACM (2006)
2. Clark, K.L.: Negation as Failure. In: Gallaire, H., Minker, J. (eds.) Logic and Data Bases, pp. 292–322. Plenum Press, New York (1978)
3. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982)
4. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning About Knowledge. The MIT Press, Cambridge (1995)
5. Gammie, P., van der Meyden, R.: MCK: Model checking the logic of knowledge. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 479–483. Springer, Heidelberg (2004)
6. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Kowalski, R., Bowen, K. (eds.) Proceedings of IJCSLP, pp. 1070–1080. MIT Press, Seattle (1988)
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 KR (2012)
10. Kissmann, P., Edelkamp, S.: Gamer, a general game playing agent. KI 25(1), 49–52 (2011)
11. Lloyd, J.: Foundations of Logic Programming, 2nd edn. Springer (1987)
12. Love, N., Hinrichs, T., Haley, D., Schkufza, E., Genesereth, M.: General Game Playing: Game Description Language Specification. Tech. Rep. LG–2006–01, Stanford (2006)
13. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems. Springer, Berlin (1992)
14. van der Meyden, R., Gammie, P., Baukus, K., Lee, J., Luo, C., Huang, X.: User manual for mck 1.0.0. Tech. rep., University of New South Wales (2012)
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 AAAI, San Francisco, pp. 840–845 (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 ECAI, Montpellier, pp. 696–701 (2012)

19. Schiffel, S., Thielscher, M.: Fluxplayer: A successful general game player. In: Proceedings of AAAI, 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 IJCAI, Barcelona, pp. 1107–1112 (2011)
22. Whaley, J., Avots, D., Carbin, M., Lam, M.S.: Using datalog with binary decision diagrams for program analysis. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 97–118. Springer, Heidelberg (2005)