Model Checking GDL through MOCHA: A Case Study

Ji Ruan Wiebe van der Hoek Michael Wooldridge
Department of Computer Science
University of Liverpool
Liverpool L69 3BX, UK

Abstract

The Game Description Language (GDL) is a special purpose declarative language for defining games. GDL is used in the AAAI General Game Playing Competition, which tests the ability of computer programs to play games in general, rather than just to play a specific game. Software participants in the competition are provided with a game specified in GDL, and then required to play this game, interpreting the GDL specification for themselves in order to determine the rules of the game. However, not all GDL specifications correspond to games, let alone meaningful, non-trivial games. We address the problem of verifying that games specified in GDL satisfy appropriate conditions, defining not just games, but meaningful games; we refer to these as playability conditions. Our approach is based on model checking formulae of Alternating-time Temporal Logic (ATL) over GDL specifications. Following an introduction to GDL and ATL, we present GDL2RML, a tool enabling model checking ATL formulae over GDL specifications using MOCHA – an ATL model checker. We illustrate the approach by a case study with experimental results.

Keywords: Game Description Language, Alternating-time Temporal Logic, Specification, Model checking

1 Introduction

Game playing competitions, particularly between humans and computers, have long been part of the culture of artificial intelligence. Indeed, the victory of IBM's Deep Blue computer over then world champion chess player Gary Kasparov in 1997 is regarded as one of the most significant events in the history of AI. However, a common objection to such specialized competitions and dedicated game playing systems is that they explore only one very narrow aspect of intelligence and rationality. To overcome these objections, in 2005 AAAI introduced a *general game playing competition*, intended to test *the ability to play games in general*, rather than just the ability to play a specific game [10, 5]. Participants in the competition are computer programs, which are provided with the rules to previously unknown games during the competition itself; they are required to play these games, and the overall winner was the one that fared best overall. Note that the participant programs were required to interpret the rules of the games *themselves*, without human intervention or interpretation. The *Game Description Language* (GDL) is a special purpose, computer processable language, which was developed in order to define the games played by participant programs. Thus, a participant must be able to interpret game descriptions expressed in GDL, and then play the game autonomously.

Since GDL is a language for defining games, it seems very natural to investigate the problem of reasoning about the games defined in GDL. Just as the designer of a computer communications

protocol might want to use model checking tools to investigate the properties of the protocol (ensure it is deadlock-free, etc [4]), so the GDL game designer will typically want to investigate the properties of games. In addition to checking protocol-like properties such as deadlock-freeness, the fact that GDL is used for describing *games* suggests a whole new class of properties to check: those relating to the *strategic properties* of the game being defined.

One formalism for reasoning about games that has attracted much interest is *Alternating-time Temporal Logic* (ATL) [2]. The basic construct of ATL is the *cooperation modality*, $\langle\langle C \rangle\rangle \varphi$, where C is a collection of agents, meaning that coalition C can cooperate to achieve φ; more precisely, that C have a winning strategy for φ. ATL has been widely applied to reasoning about game-like multi-agent systems in recent years, and has proved to be a powerful and expressive tool for this purpose [2, 6, 8, 9, 14, 13].

In [11], we made a concrete link between ATL and GDL. Specifically, (1) we demonstrated that GDL can be understood as a specification language for ATL models, and proved that the problem of interpreting ATL formulae over propositional GDL descriptions is EXPTIME-complete; (2) we classified the *playability* conditions which characterize when a given GDL description defines a (meaningful) game.

In this report we explore this topic more practically. Our main contribution is the development of an automated tool that transforms a GDL description into RML, the model description language for the MOCHA (an ATL model checker), thereby permitting the use of MOCHA for verifying playability conditions on the RML description. The report is organized as follows. Section 2 introduces the background of this work: GDL and ATL. Section 3 presents details on the GDL2RML translator with its design and evaluation. Section 4 gives a case study and experimental results. Section 5 concludes the report.

2 Background

In this section, we give the necessary background for understanding this report. For a more extensive treatment, refer to [11].

2.1 Game Descriptions and Game Models in GDL

GDL is a specialised language, intended for defining games [5, 7]. A game description must define the states of the game, a unique initial state, and the players in the game ("roles" in GDL parlance). For every state and every player, the game description must define the moves (a.k.a. actions) available to that player in that state, as well as the state transition function of the game – how moves transform the state of play. Finally, it must define what constitutes a win, and when a game is over. The approach adopted by GDL is to use a *logical* definition of the game. We introduce GDL by way of an example (Figure 1): a version of "Tic-Tac-Toe". In this game, two players take turns to mark a 3×3 grid, and the player who succeeds in placing three of its marks in a row, column, or diagonal wins.

GDL uses a prefix rule notation based on LISP. The Tic-Tac-Toe game in Figure 1 consists of 82 lines. The first two lines, (`role xplayer`) and (`role oplayer`), define the two players in this game. The following `init` lines (lines 03-12) define facts true in the initial state of the game (all the cells are blank, and `xplayer` has the control of the game). The following rule (line 13-15) defines the effect of making a move: if $cell(m,n)$ is blank (`cell ?m ?n b`), and `xplayer` marks it, then in the `next` state, it will be true that $cell(m,n)$ is marked by `x`: (`cell ?m ?n x`). The next rule (line 28-29) says that if the current state is under the control of `xplayer`, then the next state will be under the control of `oplayer`.

```
01 (role xplayer)                    54  (<= (legal ?w (mark ?x ?y))
02 (role oplayer)                    55      (true (cell ?x ?y b))
03 (init (cell 1 1 b))               56      (true (control ?w)))
    ...                              57  (<= (legal oplayer noop)
11 (init (cell 3 3 b))               58      (true (control xplayer)))
12 (init (control xplayer))              ...
13 (<= (next (cell ?m ?n x))         61  (<= (goal xplayer 100)
14     (does xplayer (mark ?m ?n))   62      (true (line x)))
15     (true (cell ?m ?n b)))            ...
    ...                              77  (<= terminal
28 (<= (next (control oplayer))      78      (line x))
29     (true (control xplayer)))     79  (<= terminal
30 (<= (row ?m ?x)                   80      (line o))
31     (true (cell ?m 1 ?x))         81  (<= terminal
32     (true (cell ?m 2 ?x))         82      (not open))
33     (true (cell ?m 3 ?x)))
   ...
```

Figure 1: A fragment of a game in the Game Description Language

Lines 30-33 define what it means to have a `row` of symbols (we omit a number of related rules). The `legal` rule (line 54-56) defines when it is `legal` for a player ?`w` to perform a mark action. The `goal` rule (line 61-62) defines the aim of the game: it says that the `xplayer` will get a reward of 100 if it brings about a line marked by `x`. The final, `terminal` rules (line 77-82) define when the game has ended.

Overall, a GDL description consists of a list of such rules, and the semantics of these rules are similar to logic programming languages. Certain operators in a GDL description have a special meaning: `role` (used to define the players of the game); `init` (defining initial facts); `legal` (defining pre-conditions for actions); and `goal` (defining rewards for agents). An additional operator, `true`, is sometimes used, to make explicit that a particular expression should be true in the current state of the game.

While GDL in [5, 7] permits predicates such as (`cell` ?`m` ?`n` `b`), we simplify this by allowing only nullary predicates, i.e., propositions. We can do this via instantiation of the predicates, i.e., replacing variables with their values. For example, variables like ?`m`, ?`n` are replaced by elements in their domain $\{1, 2, 3\}$. Thus (`cell` ?`m` ?`n` `b`) is instantiated as (`cell 1 1 b`), (`cell 1 2 b`), $\cdots$, (`cell 3 3 b`). It is easy to see that the rule in (line 13-15) is replaced by 9 rules with no predicates, and in general, there will inevitably be an undesirable blow-up in the number of rules when translating from arbitrary predicate form; nevertheless, the translation is possible, a point that is implicitly used in what follows. We refer to (`cell 1 1 b`) as a nullary predicate, or an atomic proposition. This fragment of GDL will be referred to as propositional GDL in the remainder of the report.

We now formally define GDL syntax and game descriptions.

Definition 2.1 (GDL Syntax) *Let a primitive set of proposition symbols Prim* $= \{\hat{\mathtt{p}}, \hat{\mathtt{q}} \cdots\}$*, a set of agents Ag, a set of actions Ac, a set of strings S, and a set of integers [0..100]*[1] *be given. The set of* atomic propositions *of* GDL*, denoted* At_{GDL}*, is defined as the smallest set that satisfies the following conditions:*

- *Prim* $\subseteq At_{\text{GDL}}$*;*
- *a special atom* `terminal` $\in At_{\text{GDL}}$*;*

[1] This set of integers is used to indicate the payoffs of agents in each state, following [7].

- *for two strings* $\mathtt{s}_1, \mathtt{s}_2 \in S$, $(\mathtt{distinct}\ \mathtt{s}_1\ \mathtt{s}_2) \in At_{\text{GDL}}$;
- *for every agent* $\mathtt{i} \in Ag$ *and action* $\mathtt{a} \in Ac$, $(\mathtt{legal}\ \mathtt{i}\ \mathtt{a}) \in At_{\text{GDL}}$;
- *for every agent* i *and a* v *integer in [0..100]),* $(\mathtt{goal}\ \mathtt{i}\ \mathtt{v}) \in At_{\text{GDL}}$.

The set of atomic expressions $AtExpr_{\text{GDL}}$ *of* GDL*, is defined as the smallest set that satisfies the following conditions:*

- *for* $\mathtt{p} \in At_{\text{GDL}}$, $\{\mathtt{p}, (\mathtt{init}\ \mathtt{p}), (\mathtt{next}\ \mathtt{p}), (\mathtt{true}\ \mathtt{p})\} \subseteq AtExpr_{\text{GDL}}$;
- *for every agent* i *and action* a, $\{(\mathtt{role}\ \mathtt{i}), (\mathtt{does}\ \mathtt{i}\ \mathtt{a})\} \subseteq AtExpr_{\text{GDL}}$.

$LitAt_{\text{GDL}}$ *is* $\{\mathtt{p}, (\mathtt{true}\ \mathtt{p}), (\mathtt{not}\ \mathtt{p}), (\mathtt{not}\ (\mathtt{true}\ \mathtt{p})) \mid \mathtt{p} \in At_{\text{GDL}}\}$. $LitExpr_{\text{GDL}}$ *is* $AtExpr_{\text{GDL}} \cup LitAt_{\text{GDL}}$.

A *game description* specifies the atoms from At_{GDL} that are true, either in the initial state, or as a result of global constraints, or as the effect of performing some joint actions in a given state.

Definition 2.2 (Game Descriptions) *A* GDL *game description* Γ *is a set of rules r of the form* [2] $(\Leftarrow (\mathtt{h})(\mathtt{e}_1)\ldots(\mathtt{e}_m))$ *where* h*, the head* $hd(r)$ *of the rule, is an element of* $AtExpr_{\text{GDL}}$ *and each* $\mathtt{e}_i$ *(*$i \in [1..m]$*) in the body* $bd(r)$ *of r is a literal expression from* $LitExpr_{\text{GDL}}$*. If* $m = 0$*, we say that r has* an empty body.

We can split every game description Γ *into four different types of rules where:*

- $\Gamma_{\mathtt{role}}$ *contains all claims of the form* $(\Leftarrow (\mathtt{role}\ \mathtt{x}))$*. They specify the agents in the game.*
- $\Gamma_{\mathtt{init}}$ *is a set of constraints of the form* $(\Leftarrow (\mathtt{init}\ \mathtt{p}))$*. They have an empty body and their heads represent initial constraints of the game.*
- $\Gamma_{\mathtt{glob}}$ *is a set of global constraints, i.e., rules of the form* $(\Leftarrow (\mathtt{p})\ (\mathtt{e}_1)\ \ldots\ (\mathtt{e}_m))$*, where* $\mathtt{p} \in At_{\text{GDL}}$ *and each body* $\mathtt{e}_i$ *(*$i \in [1..m]$*) is from* $LitAt_{\text{GDL}}$*.*
- $\Gamma_{\mathtt{next}}$ *contains all rules with a* $(\mathtt{next}\ \mathtt{p})$ *in the head:* $(\Leftarrow (\mathtt{next}\ \mathtt{p})(\mathtt{e}_1)\ldots(\mathtt{e}_m))$ *where each* $\mathtt{e}_i (i \in [1..m])$ *is from* $LitAt_{\text{GDL}}$ *or of the form* $(\mathtt{does}\ \mathtt{i}\ \mathtt{a})$*.*

Given a GDL game description, we can compute the corresponding game model. In general, a game model can be seen as a game tree, where we have a set of nodes representing states of the game, and a labeled edge from one state to another representing a transition from one state to another caused by the performance of actions/moves by players. For the description of Game Models G, our approach is equivalent to that of [5]. Instead of *roles* we will refer to a set $Ag = \{1, \ldots, n\}$ of *agents* or *players*.

Definition 2.3 (GDL Game Model) *Given the set of atomic propositions* At_{GDL}*, a* GDL Game Model *is a structure:*

$$G = \langle S, s_0, Ag, Ac_1, \cdots, Ac_n, \tau, \pi \rangle$$

where S is a finite, non-empty set of game states; $s_0 \in S$ *is the initial state of G; Ag is a finite, non-empty set of agents, or players in the game;* Ac_i *is a finite, non-empty set of possible actions or moves for agent i;* $\tau : Ac_1 \times \cdots \times Ac_n \times S \to S$ *is such that* $\tau(\langle a_1, \ldots, a_n \rangle, s) = u$*, means that if in game state s, agent i chooses action* a_i*,* $(i \leq n)$*, the system will change to its successor state u – we require all states, except the initial state, have only one predecessor; and finally,* $\pi : S \to 2^{At_{\text{GDL}}}$ *is an interpretation function, which associates with each state the set of atomic propositions in* At_{GDL} *that are true in that state. We will often abbreviate an action profile* $\langle a_1, \ldots, a_n \rangle$ *to* $\vec{a}$*.*

[2] We do not allow disjunctions in the body of a GDL rule as done in [7]. A rule like $(\Leftarrow (\mathtt{h})(\mathtt{e}_1 \vee \mathtt{e}_2))$ can be replaced by its equivalence, two rules $(\Leftarrow (\mathtt{h})(\mathtt{e}_1))$ and $(\Leftarrow (\mathtt{h})(\mathtt{e}_2))$.

Note that we do not include the subset $T \subseteq S$ included in the game models of [5, 7]. This subset is supposed to denote the terminal states: we can obtain this set in G by simply collecting all the states that satisfy `terminal`.

We are able to specify when a GDL game model G is a model for a GDL game description Γ (See [11] for detail). Moreover, a GDL game model can be associated with an ATL game model, enabling the use of ATL for the reasoning of games in GDL.

2.2 ATL Language and Semantics

We now introduce a logic for reasoning about games defined using GDL. For this, we believe ATL is ideally suited [2]. The key construct in ATL is $\langle\langle C\rangle\rangle T\varphi$, where C is a coalition, (a set of agents), and $T\varphi$ a temporal formula, meaning "coalition C can act in such a way that $T\varphi$ is guaranteed to be true". Temporal formulae are built using the unary operators $\bigcirc$, $\Box$, $\Diamond$, and $\mathcal{U}$, where $\bigcirc$ means "in the next state", $\Box$ means "always", $\Diamond$ means "eventually", and the binary operator $\mathcal{U}$ means "until".

Definition 2.4 (ATL Language) *The language of* ATL *(with respect to a set of agents Ag, and a set of atomic propositions Φ), is given by the following grammar:*

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \langle\langle C\rangle\rangle \bigcirc \varphi \mid \langle\langle C\rangle\rangle \Box \varphi \mid \langle\langle C\rangle\rangle \varphi \,\mathcal{U}\, \varphi$$

where $p \in \Phi$ is a propositional variable and $C \subseteq Ag$ is a set of agents.

ATL has a number of equivalent semantics; since *moves*, or *actions*, play such a prominent role in game playing, we use *Action-based Alternating Transition Systems*.

Definition 2.5 *An Action-based Alternating Transition System (*AATS*) is a tuple*

$$\mathcal{A} = \langle Q, q_0, Ag, Ac_1, \ldots, Ac_n, \rho, \tau, \Phi, \pi \rangle$$

where: Q is a finite, non-empty set of states*; $q_0 \in Q$ is the* initial state*; $Ag = \{1, \ldots, n\}$ is a finite, non-empty set of* agents*; Ac_i is a finite, non-empty set of* actions*, for each $i \in Ag$, where $Ac_i \cap Ac_j = \emptyset$ for all $i \neq j \in Ag$; $\rho : Ac_{Ag} \rightarrow 2^Q$ is an* action precondition function*, which for each action $a \in Ac_{Ag} (= \bigcup_{i \in Ag} Ac_i)$ defines the set of states $\rho(a)$ from which a may be executed; $\tau : Ac_1 \times \cdots \times Ac_n \times Q \rightarrow Q$ is a partial* system transition function*, which defines the state $\tau(\vec{a}, q)$ that would result by the performance of $\vec{a}$ from state q – note that, as this function is partial, not all joint actions are possible in all states (cf. the precondition function above); Φ is a finite, non-empty set of* atomic propositions*; and $\pi : Q \rightarrow 2^\Phi$ is an interpretation function, which gives the set of atomic propositions satisfied in each state: if $p \in \pi(q)$, then this means that the propositional variable p is satisfied in state q.*

In order to be consistent with GDL, we require that AATSs satisfy the following coherence constraints: (*Non-triviality*) agents always have at least one legal action – $\forall q \in Q, \forall i \in Ag, \exists a \in Ac_i$ s.t. $q \in \rho(a)$; and (*Consistency*) the ρ and τ functions agree on actions that may be performed: $\forall q, \forall \vec{a} = \langle a_1, \cdots, a_n \rangle, (\vec{a}, q) \in \text{dom}\ \tau$ iff $\forall i \in Ag, q \in \rho(a_i)$.

Given an agent $i \in Ag$ and a state $q \in Q$, we denote the *options* available to i in q – the actions that i may perform in q – by $options(i, q) = \{a \mid a \in Ac_i \text{ and } q \in \rho(a)\}$. For a coalition C, we define $options(C, q) = \bigcup\{options(i, q) \mid i \in C\}$. We then define a *strategy* for an agent $i \in Ag$ as a function $\sigma_i : Q \rightarrow Ac_i$ which must satisfy the *legality* constraint that $\sigma_i(q) \in options(i, q)$ for all

$q \in Q$. In this definition, a strategy is memoryless in the sense that an action is chosen only for states, not for a history of states. A *strategy profile* for a coalition $C = \{i_1, \ldots, i_k\} \subseteq Ag$ is a tuple of strategies $\langle \sigma_1, \ldots, \sigma_k \rangle$, one for each agent $i \in C$. We denote by Σ_C the set of all strategy profiles for coalition $C \subseteq Ag$; if $\sigma_C \in \Sigma_C$ and $i \in C$, then we denote i's component of σ_C by σ_C^i. Given a strategy profile $\sigma_C \in \Sigma_C$ and state $q \in Q$, let $out(\sigma_C, q)$ denote the set of possible states that may result by the members of the coalition C acting as defined by their components of σ_C for one step from q:

$$out(\sigma_C, q) = \{q' \mid \tau(\vec{a}, q) = q' \text{ where } (\vec{a}, q) \in \text{dom } \tau \text{ and } \sigma_C^i(q) = a_i \text{ for } i \in C\}$$

Notice that the set $out(\sigma_{Ag}, q)$ is a singleton. Also, $out(\cdot, \cdot)$ only deals with one-step successors, and we interchangeably write $out(\sigma_C, q)$ and $out(Ac_C, q)$: for the one step future, a strategy carries the same information as an action. A q_0-*computation* is an infinite sequence of states $\lambda = q_0, q_1, \ldots$. If $u \in \mathbb{N}$, then we denote by $\lambda[u]$ the component indexed by u in λ.

Given a strategy profile σ_C for some coalition C, and a state $q \in Q$, we define $comp(\sigma_C, q)$ to be the set of possible runs that may occur if every agent $i \in C$ follows the corresponding strategy σ_i, starting when the system is in state $q \in Q$. That is, the set $comp(\sigma_C, q)$ will contain all possible q-computations that the coalition C can "enforce" by cooperating and following the strategies in σ_C.

$$comp(\sigma_C, q) = \{\lambda \mid \lambda[0] = q \text{ and } \forall u \in \mathbb{N} : \lambda[u+1] \in out(\sigma_C, \lambda[u])\}.$$

Again, note that for any state $q \in Q$ and any grand coalition strategy σ_{Ag}, the set $comp(\sigma_{Ag}, q)$ will be a singleton, consisting of exactly one infinite computation.

We can now give the rules defining the satisfaction relation "$\models_{\text{ATL}}$" for ATL, which holds between pairs of the form $\mathcal{A}, q$ (where $\mathcal{A}$ is an AATS and q is a state in $\mathcal{A}$), and formulae of ATL.

Definition 2.6 (ATL Semantics) *Given an $\mathcal{A}$, a state q, the semantics of* ATL *is defined as follows:*

$\mathcal{A}, q \models_{\text{ATL}} p$ *iff* $p \in \pi(q)$ *(where $p \in \Phi$);*

$\mathcal{A}, q \models_{\text{ATL}} \neg\varphi$ *iff* $\mathcal{A}, q \not\models_{\text{ATL}} \varphi$;

$\mathcal{A}, q \models_{\text{ATL}} \varphi \vee \psi$ *iff* $\mathcal{A}, q \models_{\text{ATL}} \varphi$ *or* $\mathcal{A}, q \models_{\text{ATL}} \psi$;

$\mathcal{A}, q \models_{\text{ATL}} \langle\langle C \rangle\rangle \bigcirc \varphi$ *iff* $\exists \sigma_C \in \Sigma_C$, *such that* $\forall \lambda \in comp(\sigma_C, q)$, *we have* $\mathcal{A}, \lambda[1] \models_{\text{ATL}} \varphi$;

$\mathcal{A}, q \models_{\text{ATL}} \langle\langle C \rangle\rangle \Box \varphi$ *iff* $\exists \sigma_C \in \Sigma_C$, *such that* $\forall \lambda \in comp(\sigma_C, q)$, *we have* $\mathcal{A}, \lambda[u] \models_{\text{ATL}} \varphi$ *for all* $u \in \mathbb{N}$;

$\mathcal{A}, q \models_{\text{ATL}} \langle\langle C \rangle\rangle \varphi \,\mathcal{U}\, \psi$ *iff* $\exists \sigma_C \in \Sigma_C$, *such that* $\forall \lambda \in comp(\sigma_C, q)$, *there exists some* $u \in \mathbb{N}$ *such that* $\mathcal{A}, \lambda[u] \models_{\text{ATL}} \psi$, *and for all* $0 \leq v < u$, *we have* $\mathcal{A}, \lambda[v] \models_{\text{ATL}} \varphi$.

The remaining classical logic connectives ("$\wedge$", "$\rightarrow$", "$\leftrightarrow$") are assumed to be defined as abbreviations in terms of $\neg, \vee$, in the conventional manner. And $\langle\langle C \rangle\rangle \Diamond \varphi$ is defined as $\langle\langle C \rangle\rangle \top \,\mathcal{U}\, \varphi$. For readability, we omit set brackets in cooperation modalities, for example writing $\langle\langle 1 \rangle\rangle$ instead of $\langle\langle \{1\} \rangle\rangle$ and writing $\langle\langle \rangle\rangle$ in stead of $\langle\langle \{\} \rangle\rangle$.

2.3 Linking GDL and ATL

We can see that GDL and ATL are intimately related at the semantic level: GDL is a language for defining games, while ATL is a language for expressing properties of such games. The difference between the two languages is that GDL takes a relatively *constructive*, *internal* approach to a game description, essentially defining how states of the game are constructed and related by possible moves. In contrast, ATL takes an *external*, *strategic* view: while it seems an appropriate language with which to express potential strategic properties of games, it is perhaps not very appropriate for defining games.

We can build a link between GDL and ATL as follows:

- On the semantic level, every GDL description Γ has an ATL *model* associated with it.
- On the syntactic level, every GDL description Γ has an ATL *theory* associated with it.

With this link, we can answer the following question: how complex is it to interpret a property, represented by an ATL formula, over a game represented by a GDL description?

Theorem 2.1 ATL *model checking over propositional* GDL *game descriptions is EXPTIME-complete.*

For the detail of the proof, refer to [11]. Note that, although this seems a negative result, it means that interpreting ATL over propositional GDL descriptions is no more complex than interpreting ATL over apparently simpler model specification languages such as the Simple Reactive Systems Language [12].

3 The GDL2RML Translator

In this section, we describe our work on how to verify the games in GDL using an ATL model checker. The main purpose of our work is to show a method using existing ATL model checking tools on the verification of GDL games, rather than developing a model checking tool from the scratch. In this way, we can add values to the work that has been done by other people.

We built a translator, GDL2RML, from GDL descriptions to representations in the Reactive Modules Language (RML). RML is the model description language of the ATL model checker MOCHA. Using GDL2RML, we can verify properties expressed in ATL via MOCHA.

3.1 MOCHA and RML

The ATL model checker we use is MOCHA, which was developed by Alur et al. [3, 1]. The input language of MOCHA, Reactive Modules Language (RML), is rich enough to model systems with heterogeneous components: synchronous, asynchronous, speed-independent or real-time, finite or infinite state, etc. Here we briefly introduce RML.

An RML specification consists of a set of modules. A module can be seen as a function; it consists of a set of variables and a set of rules to define the evolution of the variables that are controlled by the module. The input variables are called *external* variables, and the output variables are called *interface* variables. A module controls its interface variables and its private variables. Within a module, the basic construct is an atom. A simple example of an atom is given in Figure 2.

This definition has three parts: 1) a declaration of the variables that are *controlled, read,* or *awaited*; 2) an `init` part; and 3) an `update` part. An atom can write the variables that it `controls`, read the ones it `reads` or it `awaits`. Awaiting the value of a variable y means it reads the value that y will receive in the next round, determined by another atom. For instance, in the example, the next value of x is evaluated using the current value of z, and *after* the next value of y is specified. The `init` part initializes the value of x by a set of guarded commands starting with a `'[]'`. A guarded command statement consists of two parts: *a guard*, that is a boolean expression specifying when the guarded command can be executed, and a list of *commands*, used to specify the next value of the controlled variables. If several guards are true, the system randomly chooses an associated command. The `next` part is different to the `init` in two ways: first, it can repeatedly execute after the first round, while `init` only executes in the first round; second, it can take boolean expressions with variables as guards, while `init` can only have the guard `true`.

```
atom xyz
 controls x
 reads x, z
 awaits y
init
 [] true -> x':=0
 [] true -> x':=1
update
 [] y'=true & z = false -> x':=x+1
 [] y' = true & z = true -> x':= x
 [] y'=false -> x':=x-1
endatom
```

Figure 2: An example of an atom

The state of a system at one time point is completely captured by the valuations of the variables that the system controls. The evolution of the state of the system is decided by the initial state and the update commands in each atom.

3.2 Design of the GDL2RML translator

Given a GDL description, how can we obtain a representation in RML which characterizes the same system (game structure)? Basically, we have to take care of this for a *state* and the *change* of a state. Both in GDL and RML, a state is represented by a set of propositions or variables. And for the change of a state, GDL uses a set of rules in a logical programming language, while RML uses guarded commands. GDL rules are different from RML guarded commands in two ways: 1) a GDL rule can specify the value of one proposition or variable only, but an RML guarded command can specify more than one; 2) all GDL rules will be executed if their conditions are true, but only one RML guarded command will be executed within the same atom on any given round.

In RML, we need to specify which propositions or variables belong to which module, where modules can be seen as agents. So the main tasks of our GDL2RML translator are:

- to specify the roles in GDL as modules in RML,
- to specify the propositions controlled by each module,
- to specify the initial state and the corresponding update mechanism.

We separate a GDL description Γ into four main parts: $\Gamma_{\texttt{role}}, \Gamma_{\texttt{init}}, \Gamma_{\texttt{next}}, \Gamma_{\texttt{glob}}$, where $\Gamma_{\texttt{role}}$ is a collection of the rules with keyword `role`, $\Gamma_{\texttt{init}}$ is a collection of the rules with keyword `init`, $\Gamma_{\texttt{next}}$ is a collection of the rules with keyword `next`, and $\Gamma_{\texttt{glob}}$ contains the rest. Our GDL2RML translator is written in Java, and processes the rules in these four categories as follows.

Roles For every rule in $\Gamma_{\texttt{role}}$, we associate it with a module; moreover, we introduce a special module called Gmaster, which takes the same responsibility as the game master in the General Game Playing competition (GGP) [5]. The main duties of the Gmaster are: to serve the players with the current game board state, to read the actions of the players, and to update the board state accordingly. The behaviour of Gmaster is deterministic, and it will not influence the outcome of a game. In terms of ATL, we have that, for any coalition C, $\langle\langle C \cup \{Gm\}\rangle\rangle \Diamond\varphi \leftrightarrow \langle\langle C\rangle\rangle \Diamond\varphi$.

Propositions and Variables Each player module controls their own action variables, and all the rest are controlled by the Gmaster. In other words, the players decide about their move, and all its consequences are then determined. To be more specific, we use a variable DONE_X for each player X, and the scope of this variable is easily identified by the clauses with keywords 'does X' in $\Gamma_{\texttt{next}}$. For example, in the Tic-Tac-Toe game in Figure 1, we have a clause (does xplayer (mark ?m ?n)), so we add MARK_1_1, . . . , MARK_3_3 to the domain of DONE_XPLAYER, given that the scope of ?m and ?n are determined by the context. The reason to choose a DONE prefix is related to the update mechanism, which will be introduced shortly.

The propositions for the Gmaster module are directly obtained from the propositions in Γ. For example, we have in Figure 1 a rule (<= terminal (line x)), so we take TERMINAL and LINE_X as propositions. Theoretically, we can represent the state using only propositions, but to make our representation more efficient, we choose some variables to have a richer domain. This will be explained in more detail in the case study in section 4.

The Initial State As we mentioned earlier, a state is a full characterization of the system in a particular time point. In $\Gamma_{\texttt{init}}$, GDL specifies the propositions that are true initially, but $\Gamma_{\texttt{init}}$ does not necessarily includes *all* the propositions that are true initially, because some global rules in $\Gamma_{\texttt{glob}}$ can make some propositions true as well. For example, suppose Γ_1 consists of the following two rules: (init p) and (<= q p). In the initial state, we first know p is true, and then know that q is true by the global rule (<= q p). So we need to do some computation, w.r.t $\Gamma_{\texttt{init}} \cup \Gamma_{\texttt{glob}}$, to get a complete picture of the initial state. But RML does not make it possible to do computations for the initial state, as all the guards in the *init* part can only be true (see Figure 2).

We have two obvious design choices here: either we figure out all the initial values of the variables and then specify their values directly using [] true -> x' :=value in RML, or we add an extra round to allow the modules' update part to compute the full initial state. We take the later approach, as we want to delegate all the work of constructing the game system to MOCHA. Therefore, we introduce a special variable preinit, the idea being that we make preinit true initially and then false always afterwards. If we call this special state produced by RML the 'pre-init' state, the real init state is then the computed successor of the pre-init state. In the example of Γ_1 above, p would be true in pre-init, and p and q in init. We make sure that there is only one init state. In the following, when we refer to the "init state", it should be understood that we are *not* referring to the pre-init state.

The Update Mechanism In RML, state changes are made via the update construction, which is specified with keyword update. There are two types of update rules in GDL, namely $\Gamma_{\texttt{glob}}$, which gives constrains globally, and $\Gamma_{\texttt{next}}$, which talks about the future. Accordingly, we have to deal with both of them in RML.

For the rules in $\Gamma_{\texttt{glob}}$, we add one rule in the atom which has the head of the rule as a controlled variable. And we use primed versions of the variables as they all update in the same round, and the ones in the guards are updated earlier than those in the commands. The dependency requires that there is no circularity, and this is checked by MOCHA automatically. Here is an example from Figure 1: for the GDL rule (<= terminal (line x)), we have an update rule in RML: LINE_X' -> TERMINAL' :=true, which says that if LINE_X is true in the next round, then TERMINAL is true in the next round as well. Note that LINE_X is an *awaited* variable.

For the rules in $\Gamma_{\texttt{next}}$, we also add one rule in the atom which has the head of the rule as a controlled variable. For example: a GDL rule

```
(<= (next(cell 1 1 x)) (does xplayer (mark 1 1)) (cell 1 1 b))
```

can be translated to an update rule:

```
CELL_1_1=B & DONE_XPLAYER'=MARK_1_1 -> CELL_1_1':=X.
```

This rule says if the Cell(1,1) is blank currently, and Xplayer marks Cell(1,1), then in the next state, the Cell(1,1) becomes X.

How does the whole system evolve? In GDL, the players (roles) make a choice and the game master uses them to update the state, according to the $\Gamma_{\texttt{glob}}$ and $\Gamma_{\texttt{next}}$ rules. This continues until a terminal state is reached. In RML, we will do the same thing, but an important question here is how to record the players' actions in a state. For example, suppose in the current state, player X is allowed to make a move `MARK_1_1`. Shall we have a proposition `DOES_X_MARK_1_1` to indicate that player X will make this move in next state? No, because (1) this would cause the current state to only have one successor, and (2) we do not intend to say that X *does* the `MARK_1_1` move in the current state, but only like to reason hypothetically what would happen if he *would* make that move. The fact that X makes a certain move should be recorded in the *successor* state associated with that move, and not in the current state. Therefore, we introduce the `DONE_X` variable for each player X, to record the actions made by X in the previous round. The update process in RML starts as follows: all the `DONE_X` variables are given a value in the players' module, and then the Gmaster module uses the rules from $\Gamma_{\texttt{next}}$ to specify the variables in the head of these rules using the update construct, and finally Gmaster does the same with rules translated from $\Gamma_{\texttt{glob}}$. For instance, the effect of X performing a `MARK_1_1` move is captured by the following atom in the module Gmaster.

```
update
[]DONE_Xplayer'=MARK_1_1 & CELL_1_1=B -> CELL_1_1':=X
[]DONE_Oplayer'=MARK_1_1 & CELL_1_1=B -> CELL_1_1':=O
[]~(CELL_1_1=B) -> CELL_1_1' :=CELL_1_1
[]~(DONE_Xplayer'=MARK_1_1 | DONE_Oplayer'=MARK_1_1)
   & CELL_1_1=B -> CELL_1_1':=B
```

This says that if Xplayer's chosen action is to mark Cell(1,1), and this cell is currently blank, it will become marked with X, and similarly for Oplayer and the symbol O. If Cell(1,1) was already not blank, it keeps its value, and, finally, it stays blank if it was blank and nobody wrote on it in this round.

3.3 Correctness and Evaluation

How can we ensure the correctness of the GDL2RML translator? Here the 'correctness' means that the original GDL description specifies the same game model as its GDL2RML translation. In the previous section, we have formally defined the game models for the GDL descriptions. Ideally, we shall also define the game models of the RML descriptions, and formally prove that the game model of a GDL description Γ corresponds to the game model of the translation of Γ in RML. This requires a formalisation of RML with game semantics, which is out of the scope of this report. But we do have two approaches to ensure certain degree of the correctness. The *first approach* is to check whether all the propositions, variables and the rules have been mapped correctly. We get this level of assurance by checking the design of the GDL2RML translator. This might still prone to human errors, so we have a *second approach*, which is to use the model checker MOCHA to verify properties of the GDL2RML translations. If the MOCHA results agree with the truth of those properties in the original game, then we get certain degree of assurance that our translation is correct. Of course when the MOCHA results

do not conform the truth of those properties, this approach alone does not tell whether the original GDL description does not describe the game properly, or the GDL2RML translator is problematic.

As for the evaluation of the GDL2RML translator, we have tested it with a number of examples, such as Maze, Buttons and Tic-Tac-Toe, from the game depository[3] of the General Game Playing Website. All these examples were translated within several seconds in a Dural-Core Linux Machine, and the verification results in MOCHA are all as desired. In the next section, we will present a concrete case study to show that our translation of Tic-Tac-Toe has produced desired results (see Figure 3). Compared with the programming-oriented brute-force method mentioned in [7], our method has two advantages. First, we do not need to write a program to expand the game models, as the model checkers automatically generate the game models from RML descriptions. Second, we can specify the properties in ATL in a more abstract way than specifying them in a programming language, so that we do not need to deal with the details in the level of game states. Our GDL2RML translator is still a prototype tool; in theory, it shall automatically translate any GDL descriptions to RML descriptions, but in practice we still need to manually add some tags into GDL descriptions to reduce the numbers of variables in the translation, in order to reduce the model checking time in MOCHA.

4 Case Study and Experimental Results

In this section we do a case study in the context of Tic-Tac-Toe using our GDL2RML translator and the MOCHA model checker. We briefly introduce the translation from a GDL description for Tic-Tac-Toe to an RML description. Then we focus on the properties expressed as ATL specifications and verify them with MOCHA.

For the translation from a GDL description to an RML description, we have illustrated the main idea in the previous section. The only thing we mention here is the controlled variables for Gmaster. Most of them are boolean variables, and only a few can take multiple values, e.g., `CELL_1_1` $\in$ { `B,X,O`}. We can also represent this using three booleans: `CELL_1_1_B`, `CELL_1_1_X`, and `CELL_1_1_O`. Then the equivalent expression of `CELL_1_1=X` is

```
CELL_1_1_B=false & CELL_1_1_X=true & CELL_1_1_O=false.
```

We choose the former representation for the sake of compactness.

4.1 Playability Conditions of Tic-Tac-Toe in MOCHA

The general properties (also referred to as playability conditions) that we want to verify are presented in [11]. The properties will be expressed precisely and unambiguously as ATL logical formulae. A GDL game description satisfies such a formal property if and only if the ATL game model that arises from such description satisfies this property under ATL semantics. Our top-level classification of game properties distinguishes between properties relating to the *coherence* of a game and those relating to *fairness*. Now we tailor them specifically for Tic-Tac-Toe. We select a few representative properties and give concrete representations that are accepted by the model checker MOCHA. The purpose is to show how our work is used in practice. Throughout this report, unless stated otherwise, properties that we discuss are evaluated in the beginning of the game.

[3]URL: http://visionary.stanford.edu:4000

Coherence Properties The first coherence property we pick is *Playability*:

$$\langle\langle\rangle\rangle\Box(\neg terminal \rightarrow \bigwedge_{i\in Ag} has\_legal\_move_i) \qquad \text{(Playability)}$$

For $i = Xplayer$, $has\_legal\_move_i$ can be represented as

```
( LEGAL_X_MARK_11 | LEGAL_X_MARK_12 | LEGAL_X_MARK_13
| LEGAL_X_MARK_21 | LEGAL_X_MARK_22 | LEGAL_X_MARK_23
| LEGAL_X_MARK_31 | LEGAL_X_MARK_32 | LEGAL_X_MARK_33 | LEGAL_X_NOOP)
```

The remaining part is straightforward.

The second coherence property is GameOver:

$$\langle\langle\rangle\rangle\Box((terminal \wedge \varphi) \rightarrow \langle\langle\rangle\rangle\Box(terminal \wedge \varphi)) \qquad \text{(GameOver)}$$

Let us look at an instantiation of φ: suppose φ here means that Xplayer wins. Its representation in MOCHA is

```
<<>>X(<<>>G((TERMINAL & GOALX=g100)=> <<>>G(TERMINAL & GOALX=g100)))
```

Note that we have some small notional differences. Here `X` corresponds to $\bigcirc$, `G` to $\Box$, `&` to $\wedge$, and `=>` to $\rightarrow$. The reason to have `<<>>X` at the beginning is that we have an extra, "pre-initial" initial state. We have explained the reason to have such state in Section 3.2.

The third coherence property we pick is Turn:

$$\langle\langle\rangle\rangle\Box(turn_i \leftrightarrow \neg legal(i, noop)) \qquad \text{(Turn)}$$

The case with *i* being *Xplayer* is:

```
<<>>X <<>> G (turn=Xplayer <=> ~LEGAL_Xplayer_NOOP) .
```

The last coherence property we pick is Termination:

$$\langle\langle\rangle\rangle\Diamond terminal \qquad \text{(Termination)}$$

The MOCHA representation is straightforward: `<<>> X <<>> F terminal`, where `F` is the MOCHA notation for $\Diamond$.

Fairness Properties Now we pick two fairness properties:

$$\bigvee_{i\in Ag} \langle\langle i\rangle\rangle\Diamond win_i \qquad \text{(Strong Winnability)}$$

and

$$\bigwedge_{i\in Ag} \langle\langle Ag\rangle\rangle\Diamond win_i. \qquad \text{(Weak Winnability)}$$

The representations are

```
<<>>X(<<Xplayer>>F GOALX=g100 | <<Oplayer>>F GOALO=g100)
```

and

```
<<>>X(<<Xplayer,Oplayer>>F GOALX=g100 & <<Xplayer,Oplayer>>F GOALO=g100)
```

respectively.

4.2 Playing Tic-Tac-Toe via Model checking

Although our main motivation in this work is to consider the analysis of games from the view point of a game designer, it is also worth speculating about the use of our approach to play GDL games via Model checking. Let us suppose the following situation in Tic-Tac-Toe (Xplayer moves first).

X	O	
	X	
		O

Now it is Xplayer's turn. The questions are:

1. Is there a winning strategy for Xplayer in the current state?
2. If so, which move should Xplayer take?

There is indeed a winning strategy for Xplayer, namely, by marking the $Cell(2,1)$ (see below). In that case, no matter how Oplayer responds, Xplayer can mark either $Cell(2,3)$ or $Cell(3,1)$ in its next turn.

X	O	
X	X	
		O

We show that we can answer these questions via model checking.

Note that we can express "Xplayer has a winning strategy" in ATL as $\langle\langle Xplayer\rangle\rangle\Diamond win_{Xplayer}$. Its equivalent MOCHA expression is `<<Xplayer>>F GOALX=g100`. Given game model G, and current state s, the question 1 amounts to checking whether $G, s \models_{\text{ATL}} \langle\langle Xplayer\rangle\rangle\Diamond win_{Xplayer}$. In MOCHA, we can only check a property with respect to the initial state, namely s_0, but we can get around using the following approach. We characterize a state s by a formula $\varphi(s)$, so instead of checking $G, s \models_{\text{ATL}} \langle\langle Xplayer\rangle\rangle\Diamond win_{Xplayer}$, we can check $G, s_0 \models_{\text{ATL}} \langle\langle\rangle\rangle\Box(\varphi(s) \rightarrow \langle\langle Xplayer\rangle\rangle\Diamond win_{Xplayer})$. For the above example, $\varphi(s)$ can be $Cell(1,1,X) \wedge Cell(1,2,O) \wedge Cell(2,2,X) \wedge Cell(3,3,O) \wedge \bigwedge_{x,y=rest} Cell(x,y,B)$. We denote the MOCHA represent of $\langle\langle\rangle\rangle\Box(\varphi(s) \rightarrow \langle\langle Xplayer\rangle\rangle\Diamond win_{Xplayer})$ as *sXWin*.

Now, suppose we got a positive answer to the question 1. To answer the question 2, we use an action variable *DONEX* to guide the search for a proper move. The idea is to select a legal move for Xplayer, and then to check whether Xplayer still has a winning strategy under this move. If so Xplayer shall take it; if not, Xplayer will check the a different legal move; the existence of a winning strategy guarantees that there is such a move. To be more specific, suppose Xplayer chooses $mark(2,1)$, it is to check:

$$G, s_0 \models_{\text{ATL}} \langle\langle\rangle\rangle\Box(\varphi(s) \rightarrow \langle\langle Xplayer\rangle\rangle\bigcirc(DONEX = MARK\_2\_1 \wedge \langle\langle Xplayer\rangle\rangle\Diamond win_{Xplayer})).$$

We denote the MOCHA version of this formula as *sXWin_by_mark*21. If the answer is positive, it means Xplayer's move *Mark*_2_1 is indeed a move leading towards a winning position.

There is of course a question "what if there is no winning strategy in the current position?". We believe that we could explore a position evaluation function and and its connection with ATL properties. But this is out of the scope of this report.

Property	Results	Time
GameOver	passed	2sec
Turn	passed	0.3sec
Termination	passed	4min49sec
Playability	passed	0.4sec
Strong Winnability	failed	23sec
Weak Winnability	passed	4min01s
sXWin	passed	1min06sec
sXWin_by_mark21	passed	1min59sec

Figure 3: Verification results of Tic-Tac-Toe

4.3 Experimental results on Tic-Tac-Toe

Here we present experimental results to show that the analysis described above can be done in reasonable time with moderate computing resources. For these experiments, we ran MOCHA under Linux kernel 2.6.20 i686 with a Dural-Core 1.8Ghz CPU and 2GB RAM. The table in Figure 3 gives timings for checking the various properties listed in the previous section. These results indicate that our tool can generate correct results in a reasonable amount of time.

5 Conclusions

There has been much interest recently in the connections between logic and games, and in particular in the use of ATL-like logics for reasoning about game-like multi-agent systems. In this report, we presented an automated tool that transforms a GDL description into an RML specification, so that we can verify the playability properties on the RML description using an off-the-shelf ATL model checker, MOCHA. In future research, we will apply our work to formal verification of further GDL descriptions: the GDL game designer can express desirable properties of games using ATL, and then automatically check whether these properties hold of their GDL descriptions. The main issues are likely to be the efficiency and scalability of our automated tools. We believe that there is much room for improvement with respect to the current results. In particular, it might be useful in future to consider investing some effort in optimising the translation process from GDL to RML, particularly with respect to the number of variables produced in the translation. Even moderate optimisations might yield substantial time and memory savings.

References

[1] R. Alur, L. de Alfaro, T. A. Henzinger, S. C. Krishnan, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, and S. Taşiran. MOCHA user manual. University of Berkeley Report, 2000.

[2] R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 49(5):672–713, September 2002.

[3] R. Alur, T. A. Henzinger, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, and S. Taşiran. Mocha: Modularity in model checking. In *CAV 1998: Tenth International Conference on Computer-aided Verification, (LNCS Volume 1427)*, pages 521–525. Springer-Verlag: Berlin, Germany, 1998.

[4] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press: Cambridge, MA, 2000.

[5] M. Genereseth and N. Love. General game playing: Overview of the AAAI competition. Technical report, Stanford University, Stanford, 2005.

[6] V. Goranko. Coalition games and alternating temporal logics. In J. van Benthem, editor, *Proceeding of the Eighth Conference on Theoretical Aspects of Rationality and Knowledge (TARK VIII)*, pages 259–272, Siena, Italy, 2001.

[7] Nathaniel Love, Timothy Hinrichs, and Michael Genesereth. General game playing: Game description language specification. Technical report, Stanford University, Stanford, April 2006.

[8] M. Pauly. *Logic for Social Software*. PhD thesis, University of Amsterdam, 2001. ILLC Dissertation Series 2001-10.

[9] M. Pauly and M. Wooldridge. Logic for mechanism design — a manifesto. In *Proceedings of the 2003 Workshop on Game Theory and Decision Theory in Agent Systems (GTDT-2003)*, Melbourne, Australia, 2003.

[10] B. Pell. *Strategy Generation and Evaluation for Meta-Game Playing*. PhD thesis, Trinity College, University of Cambridge, 1993.

[11] J. Ruan, W. van der Hoek, and M. Wooldridge. Verification of games in the game description language, 2009. Accepted by the Journal of Logic and Computation.

[12] W. van der Hoek, A.R. Lomuscio, and M.J. Wooldridge. On the complexity of practical ATL model checking. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS 2006)*, pages 201–208. ACM, 2006.

[13] W. van der Hoek, M. Roberts, and M. Wooldridge. Social laws in alternating time: Effectiveness, feasibility, and synthesis. *Synthese*, 2007.

[14] W. van der Hoek and M. Wooldridge. Time, knowledge, and cooperation: Alternating-time temporal epistemic logic and its applications. *Studia Logica*, 75(1):125–157, 2003.