Translating the Game Description Language to Toss

Łukasz Kaiser
CNRS & LIAFA
Paris

Łukasz Stafiniak
Institute of Computer Science
University of Wrocław

Abstract

We show how to translate games defined in the Game Description Language (GDL) into the Toss format. GDL is a variant of Datalog used to specify games in the General Game Playing Competition. Specifications in Toss are more declarative than in GDL and make it easier to capture certain useful game characteristics. The presented translation must thus detect structural properties of games which are not directly visible in the GDL specification.

Introduction

General Game Playing (GGP) is concerned with the construction of programs able to play arbitrary games without specific information about the game present in the program. GGP programs compete against each other in the GGP Competition, where games are specified in the Game Description Language (GDL), cf. (Genesereth and Love 2005). A successful GGP agent must reason about the rules of the game and extract from them game-specific knowledge. To facilitate the creation of good general players, GDL was designed as a high-level, declarative language. Still, it is not directly clear from the GDL specification of a game what parts constitute a board, what the pieces are, etc. — the GDL defines only Datalog terms, no high-level game concepts.

For this reason, we introduce a new formalism, which is more tightly related to games and even higher-level and more declarative than GDL. In a companion paper (Kaiser and Stafiniak 2011) we show how these higher-level features of the introduced formalism can be used to build a competitive GGP player more easily than starting from GDL itself. But to compete against GGP players, it is necessary to translate games from GDL into the presented formalism, which in itself presents several challenges. In this work, we show how a good translation can be made, assuming certain restrictions on the GDL specification.

Games in the Toss Formalism

Since the Toss formalism is not very well known, we repeat here several definitions from the paper (Kaiser and Stafiniak 2011), but with support for concurrency and fixed-points.

The state of the game is represented in the Toss formalism by a finite relational structure, i.e. a labelled directed hypergraph. A relational structure $\mathfrak{A} = (A, R_1, \dots, R_k)$ is composed of a universe A and a number of relations $R_1, \dots, R_k$. We denote the arity of R_i by r_i, so $R_i \subseteq A^{r_i}$. The *signature* of $\mathfrak{A}$ is the set of symbols $\{R_1, \dots, R_k\}$.

Relational Structure Rewriting

The moves of the players are described by *structure rewriting rules*, a generalised form of term and graph rewriting. Structure rewriting has been introduced in (Rajlich 1973) and is most recognised in graph rewriting and software engineering communities, where it is regarded as easy to understand and well suited for visual programming.

In our setting, a rule $\mathfrak{L} \to_s \mathfrak{R}$ consists of two finite relational structures, $\mathfrak{L}$ and $\mathfrak{R}$, over the same signature, and of a partial function $s : \mathfrak{R} \to \mathfrak{L}$ specifying which elements of $\mathfrak{L}$ will be substituted by which elements of $\mathfrak{R}$. With each rule, we will also associate a set of relations symbols τ_e to be embedded exactly, as described below.

Definition 1. Let $\mathfrak{A}$ and $\mathfrak{B}$ be two relational structures over the same signature. A function $f : \mathfrak{A} \hookrightarrow \mathfrak{B}$ is a τ_e-*embedding* if f is injective, for each $R_i \in \tau_e$ it holds that

$$(a_1, \dots, a_{r_i}) \in R_i^{\mathfrak{A}} \Leftrightarrow (f(a_1), \dots, f(a_{r_i})) \in R_i^{\mathfrak{B}},$$

and for $R_j \notin \tau_e$ it holds that

$$(a_1, \dots, a_{r_j}) \in R_j^{\mathfrak{A}} \Rightarrow (f(a_1), \dots, f(a_{r_j})) \in R_j^{\mathfrak{B}}.$$

A τ_e-*match* of the rule $\mathfrak{L} \to_s \mathfrak{R}$ in another structure $\mathfrak{A}$ is a τ_e-embedding $\sigma : \mathfrak{L} \hookrightarrow \mathfrak{A}$.

Definition 2. The result of an application of $\mathfrak{L} \to_s \mathfrak{R}$ to $\mathfrak{A}$ on the match σ, denoted $\mathfrak{A}[\mathfrak{L} \to_s \mathfrak{R}/\sigma]$, is a relational structure $\mathfrak{B}$ with universe $(A \setminus \sigma(L)) \dot\cup R$, and relations given as follows. A tuple $(b_1, \dots, b_{r_i})$ is in the new relation $R_i^{\mathfrak{B}}$ if and only if either it is in the relation in $\mathfrak{R}$ already, $(b_1, \dots, b_{r_i}) \in R_i^{\mathfrak{R}}$, or there exists a tuple in the previous structure, $(a_1, \dots, a_{r_i}) \in R_i^{\mathfrak{A}}$, such that for each j either $a_j = b_j$ or $a_j = \sigma(s(b_j))$, i.e. either the element was there before or it was matched and b_j is the replacement as specified by the rule. Moreover, if $R_i \in \tau_e$ then we require in the second case that at least one b_l was already in the original structure, i.e. $b_l = a_l$ for some $l \in \{1, \dots, r_i\}$.

Logic and Constraints

The logic we use for specifying properties of states is an extension of first-order logic with least and greatest fixed-points, real-valued terms and counting operators, cf. (Grädel 2007; Grädel and Gurevich 1998).

Syntax. We use first-order variables $x_1, x_2, \ldots$ ranging over elements of the structure, second-order variables $X_1, X_2, \ldots$ ranging over relations, and we define formulas $\varphi \in \mathcal{F}_{\mathfrak{A}}$ and real-valued terms ρ by the following grammar ($n, m \in \mathbb{N}$), with second-order variables restricted to appear only positively, as usual in the least fixed-point logic.

$$\begin{aligned}\varphi &:= R_i(x_1, \ldots, x_{r_i}) \mid x_i = x_j \mid \rho < \rho \mid \neg\varphi \mid \varphi \vee \varphi \\ &\quad \mid \varphi \wedge \varphi \mid \exists x_i\, \varphi \mid \forall x_i\, \varphi \mid \mathrm{lfp}\, X_i\, \varphi \mid \mathrm{gfp}\, X_i\, \varphi\,, \\ \rho &:= \frac{n}{m} \mid \rho + \rho \mid \rho \cdot \rho \mid \chi[\varphi] \mid \sum_{\overline{x}|\varphi} \rho\,.\end{aligned}$$

Semantics. Most of the above operators are defined in the well known way, e.g. $\rho + \rho$ is the sum and $\rho \cdot \rho$ the product of two real-valued terms, and $\mathrm{lfp}\, X\, \varphi(X)$ is the least fixed-point of the equation $X = \varphi(X)$. Among less known operators, the term $\chi[\varphi]$ denotes the characteristic function of φ, i.e. the real-valued term which is 1 for all assignments for which φ holds and 0 for all other. The term $\sum_{\overline{x}|\varphi} \rho$ denotes the sum of the values of $\rho(\overline{x})$ for all assignments of elements of the structure to $\overline{x}$ for which $\varphi(\overline{x})$ holds. Note that these terms can have free variables, e.g. the set of free variables of $\sum_{\overline{x}|\varphi} \rho$ is the union of free variables of φ and free variables of ρ, minus the set $\{\overline{x}\}$.

The logic defined above is used in structure rewriting rules in two ways. First, it is possible to define a new relation $R(\overline{x})$ using a formula $\varphi(\overline{x})$ with free variables contained in $\overline{x}$. Defined relations can be used on left-hand sides of structure rewriting rules, but are not allowed on right-hand sides. The second way is to add *constraints* to a rule. A rule $\mathfrak{L} \to_s \mathfrak{R}$ can be constrained using two sentences (i.e. formulas without free variables): φ_{pre} and φ_{post}. In φ_{pre} we allow additional constants l for each $l \in \mathfrak{L}$ and in φ_{post} special constants for each $r \in \mathfrak{R}$ can be used. A rule $\mathfrak{L} \to_s \mathfrak{R}$ with such constraints can be applied on a match σ in $\mathfrak{A}$ only if the following holds: At the beginning, the formula φ_{pre} must hold in $\mathfrak{A}$ with the constants l interpreted as $\sigma(l)$, and, after the rule is applied, the formula φ_{post} must hold in the resulting structure with each r interpreted as the newly added element r (cf. Definition 2).

Structure Rewriting Games

One can in principle describe a game simply by providing a set of allowed moves for each player. Still, in many cases it is natural to specify the control flow directly. For this reason, we define games as labelled graphs as follows.

Definition 3. A *structure rewriting game* with k players is a finite directed graph in which in each vertex, called *location*, we assign to each player from $\{1, \ldots, k\}$ a real-valued *payoff term*. Each edge of the graph represents a possible move and is labelled by a tuple

$$(p, \mathfrak{L} \to_s \mathfrak{R}, \tau_{\mathrm{e}}, \varphi_{\mathrm{pre}}, \varphi_{\mathrm{post}}),$$

which specifies the player $p \in \{1, \ldots, k\}$ who moves and the rewriting rule to be used with relations to be embedded and a pre- and post-condition. Multiple edges with different labels are possible between two locations.

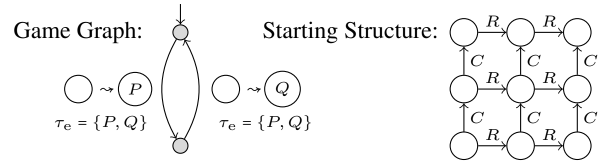

Figure 1: Tic-tac-toe as a structure rewriting game.

Play Semantics. A play of a structure rewriting game starts in a fixed initial location of the game graph and in a state given by a starting structure. A player chooses an edge on which she moves and a match allowed by the label of the edge such that it can be applied, i.e. both the pre- and the post-condition holds. The play proceeds to the location to which the edge leads and the new state is the structure after the application of the rule on the chosen match. If in some location and state it is not possible to apply any of the rules on the outgoing edges, either because no match can be found or because of the constraints, then the play ends. Payoff terms from that location are evaluated on the state and determine the outcome of the game for all players.

Example 4. Let us define Tic-tac-toe in our formalism. The starting structure has 9 elements connected by binary row and column relations, R and C, as depicted on the right in Figure 1. To mark the moves of the players we use unary relations P and Q. The allowed move of the first player is to mark any unmarked element with P and the second player can mark with Q. Thus, there are two locations in the game graph (representing which player's turn it is) and two corresponding rules, both with one element on each side (left in Figure 1). The two diagonals can be defined by

$$D_A(x, y) = \exists z (R(x, z) \wedge C(z, y)),$$

$$D_B(x, y) = \exists z (R(x, z) \wedge C(y, z))$$

and a line of three by

$$\begin{aligned}L(x, y, z) = (R(x, y) \wedge R(y, z)) \vee (C(x, y) \wedge C(y, z)) \vee \\ (D_A(x, y) \wedge D_A(y, z)) \vee (D_B(x, y) \wedge D_B(y, z)).\end{aligned}$$

Using these definitions, we specify the goal of the first player

$$\varphi = \exists x, y, z \big(P(x) \wedge P(y) \wedge P(z) \wedge L(x, y, z)\big)$$

and the goal of the other player by an analogous formula φ' in which P is replaced by Q. The payoff terms are $\chi[\varphi] - \chi[\varphi']$ for the first player and $\chi[\varphi'] - \chi[\varphi]$ for the other, and to ensure that the game ends when one of the players has won, we take as a precondition of each move the negation of the goal formula of the other player.

The complete code for Tic-tac-toe, with the starting structure in a special syntax we use for grids with row and column relations, is given in Figure 2. Note that we write `:(φ)` for $\chi[\varphi]$ and e.g. `Q:1 {}` for an empty relation Q of arity 1. Please refer to (Toss) for the complete input syntax and definitions of other, more complex games, e.g. Chess.

```
PLAYERS X, O
REL Row3(x, y, z) = R(x, y) and R(y, z)
REL Col3(x, y, z) = C(x, y) and C(y, z)
REL DgA(x, y) = ex z (R(x, z) and C(z, y))
REL DgB(x, y) = ex z (R(x, z) and C(y, z))
REL DgA3(x, y, z) = DgA(x, y) and DgA(y, z)
REL DgB3(x, y, z) = DgB(x, y) and DgB(y, z)
REL Conn3(x, y, z) =
  Row3(x, y, z) or Col3(x, y, z) or
  DgA3(x, y, z) or DgB3(x, y, z)
REL WinP()= ex x,y,z(P(x) and P(y) and P(z)
                     and Conn3(x, y, z))
REL WinQ()= ex x,y,z(Q(x) and Q(y) and Q(z)
                     and Conn3(x, y, z))
RULE Cross:
  [a | - | - ] -> [a | P (a) | - ]
  emb P, Q pre not WinQ()
RULE Circle:
  [a | - | - ] -> [a | Q (a) | - ]
  emb P, Q pre not WinP()
LOC 0 {
  PLAYER X {PAYOFF :(WinP()) - :(WinQ())
            MOVES [Cross -> 1] }
  PLAYER O {PAYOFF :(WinQ()) - :(WinP())}
}
LOC 1 {
  PLAYER X {PAYOFF :(WinP()) - :(WinQ())}
  PLAYER O {PAYOFF :(WinQ()) - :(WinP())
            MOVES [Circle -> 0]}
}
MODEL [ | P:1 {}; Q:1 {} | ] "
        .  .  .
        .  .  .
        .  .  .
"
```

Figure 2: Tic-tac-toe in the Toss formalism.

Game Description Language

The game description language, GDL, is a variant of Datalog used to specify games in a compact, prolog-like way. The GDL syntax and semantics are defined in (Genesereth and Love 2005; Love et al. 2008), we refer the reader there for the definition and will only recapitulate some notions here.

The state of the game in GDL is defined by the set of propositions true in that state. These propositions are represented by terms of limited height. The moves of the game, i.e. the transition function between the states, are described using Datalog rules — clauses define which predicates hold in the subsequent state. In this way a transition system is specified in a compact way. Additionally, there are 8 special relations in GDL: `role, init, true, does, next, legal, goal` and `terminal`, which are used to describe the game: initial state, the players, their goals, and thus like.

We say that *GDL state terms* are the terms that are possible arguments of `true`, `next` and `init` relations in a GDL specification, i.e. those terms which can define the state of the game. The *GDL move terms* are ground instances of the second arguments of `legal` and `does` relations, i.e. those terms which are used to specify the moves of the players.

The complete Tic-tac-toe specification in GDL is given in Figure 3. While games can be formalised in various ways in both systems, Figures 2 and 3 give natural examples of a formalisation, similar to several other games.

Notions Related to Terms

Since GDL is a term-based formalism, we will use the standard term notions, as e.g. in the preliminaries of (Comon et al. October 2007). We understand terms as finite trees with ordered successors and labelled by the symbols used in the current game, with leafs possibly labelled by variables.

Substitutions. A *substitution* is an assignment of terms to variables. Given a substitution σ and a term t we write $\sigma(t)$ to denote the result of applying σ to t, i.e. of replacing all variables in t which also occur in σ by the corresponding terms. We extend this notation to tuples in the natural way.

MGU. We say that a tuple of terms $\bar{t}$ is *more general* than another tuple $\bar{s}$ of equal length, written $\bar{s} \leq \bar{t}$, if there is a substitution σ such that $\bar{s} = \sigma(\bar{t})$. Given two tuples of terms $\bar{s}$ and $\bar{t}$ we write $\bar{s} \doteq \bar{t}$ to denote that these tuples *unify*, i.e. that there exists a substitution σ such that $\sigma(\bar{s}) = \sigma(\bar{t})$. In such case there exists a most general substitution of this kind, and we denote it by $\mathrm{MGU}(\bar{s}, \bar{t})$.

Paths. A *path* in a term is a sequence of pairs of function symbols and natural numbers denoting which successor to take in turn, e.g. $p = (f, 1)(g, 2)$ denotes the second child of a node labelled by g, which is the first child of a node labelled by f. For a term t we write $t\downarrow_p$ to denote the subterm of t at path p, and that t has a path p, i.e. that the respective sequence of nodes exists in t with exactly the specified labels. Using $p = (f, 1)(g, 2)$ as an example, $f(g(a, b), c) \downarrow_p = b$, but $g(f(a, b), c) \downarrow_p$ is false. Similarly, for a formula φ, we write $\varphi(t \downarrow_p)$ to denote that t has path p and the subterm $r = t \downarrow_p$ satisfies $\varphi(r)$. A path can be an empty sequence ε and $t \downarrow_\varepsilon = t$ for all terms t.

For any terms t, s and any path p existing in t, we write $t[p \leftarrow s]$ to denote the result of *placing s at path p in t*, i.e. the term t' such that $t' \downarrow_p = s$ and on all other paths q, i.e. ones which neither are prefixes of p nor contain p as a prefix, t' is equal to t, i.e. $t' \downarrow_q = t \downarrow_q$. We extend this notation to sets of paths as well: $t[P \leftarrow s]$ places s at all paths from P in t.

Translation

In this section, we describe our main construction. Given a GDL specification of a game G, which satisfies the restrictions described in the last section, we construct a Toss game $T(G)$ which represents exactly the same game. Moreover, we define a bijection μ between the moves possible in G and in $T(G)$ in each reachable state, so that the following correctness theorem holds.

```
(role x)
(role o)
(init (cell a a b))
(init (cell b a b))
(init (cell c a b))
(init (cell a b b))
(init (cell b b b))
(init (cell c b b))
(init (cell a c b))
(init (cell b c b))
(init (cell c c b))
(init (control x))
(<= (next (control ?r)) (does ?r noop))
(<= (next (cell ?x ?y ?r))
  (does ?r (mark ?x ?y)))
(<= (next (cell ?x ?y ?c))
  (true (cell ?x ?y ?c))
  (does ?r (mark ?x1 ?y1))
  (or (distinct ?x ?x1) (distinct ?y ?y1)))
(<= (legal ?r (mark ?x ?y))
      (true (control ?r))
      (true (cell ?x ?y b)))
(<= (legal ?r noop) (role ?r)
      (not (true (control ?r))))
(<= (goal ?r 100) (conn3 ?r))
(<= (goal ?r 50) (role ?r)
      (not exists_line3))
(<= (goal x 0) (conn3 o))
(<= (goal o 0) (conn3 x))
(<= terminal exists_line3)
(<= terminal (not exists_blank))
(<= exists_blank (true (cell ?x ?y b)))
(<= exists_line3 (role ?r) (conn3 ?r))
(<= (conn3 ?r) (or (col ?r) (row ?r)
                   (diag1 ?r) (diag2 ?r)))
(<= (row ?r)
  (true (cell ?a ?y ?r)) (nextcol ?a ?b)
  (true (cell ?b ?y ?r)) (nextcol ?b ?c)
  (true (cell ?c ?y ?r)))
(<= (col ?r)
  (true (cell ?x ?a ?r)) (nextcol ?a ?b)
  (true (cell ?x ?b ?r)) (nextcol ?b ?c)
  (true (cell ?x ?c ?r)))
(<= (diag1 ?r)
  (true (cell ?x1 ?y1 ?r))
  (nextcol ?x1 ?x2) (nextcol ?y1 ?y2)
  (true (cell ?x2 ?y2 ?r))
  (nextcol ?x2 ?x3) (nextcol ?y2 ?y3)
  (true (cell ?x3 ?y3 ?r)))
(<= (diag2 ?r)
  (true (cell ?x1 ?y5 ?r))
  (nextcol ?x1 ?x2) (nextcol ?y4 ?y5)
  (true (cell ?x2 ?y4 ?r))
  (nextcol ?x2 ?x3) (nextcol ?y3 ?y4)
  (true (cell ?x3 ?y3 ?r)))
(nextcol a b)
(nextcol b c)
```

Figure 3: Tic-tac-toe in the Game Description Language.

Theorem 5 (Correctness)**.**
Let S be any state of G reached from the initial one by a sequence of moves $m_1 \dots m_n$. We write $\mu(S)$ for the state of $T(G)$ reached by $\mu(m_1) \dots \mu(m_n)$. The following conditions are satisfied.

- *The function μ defines a bijection between the moves possible in S and in $\mu(S)$ for each player.*
- *If no move is possible in S (and in $\mu(S)$), then the payoffs in G evaluate to the same value as those in $T(G)$.*

We will not prove this theorem here, but the construction presented below should make it clear why the exact correspondence holds. For the rest of this section let us fix the GDL game specification G we will translate. We begin by transforming G itself: eliminating variables clearly referring to players (i.e. arguments of positive `role` atoms, first arguments to positive `does` atoms and to `legal`) by substituting them by players of G (i.e. arguments of `role` facts), duplicating the clauses. From this transformed specification, we derive the elements of the Toss structure (next subsection), the relations (subsection after the next), the rewriting rules (further subsection) and finally the move translation function (last subsection).

Elements of the Toss Structure

By definition of GDL, the state of the game is described by a set of propositions true in that state. Let us denote by $\mathcal{S}$ the set of all GDL state terms which are true at some game state reachable from the initial state of G.

For us, it is enough to approximate $\mathcal{S}$ from above. To approximate $\mathcal{S}$, we currently perform an *aggregate playout*, i.e. a symbolic play in where all players take all their legal moves in a state. Since an approximation is sufficient, we check only the positive part of the legality condition of each move.

To construct the elements of the structure from state terms, and to make that structure a good representation of the game in Toss, we first determine which state terms always have common subtrees.

Definition 6. For two terms s and t we say that a set of paths P *merges* s and t if each $p \in P$ exists both in s and t and $t[P \leftarrow c] = s[P \leftarrow c]$ for all terms c. We denote by $d\mathcal{P}(s,t)$ the unique set P of paths merging s and t for which the size of $t[P \leftarrow c]$ is maximal and no subset of which merges s and t. Intuitively, $t[d\mathcal{P}(s,t) \leftarrow c]$ is the largest common subtree of s and t, the bigger its size the more similar s and t are.

Let Next_e be the set of `next` clauses in G with all atoms of `does` expanded (inlined) by the `legal` clause definitions, duplicating the `next` clause when more than one head of `legal` unifies with the `does` atom. Intuitively, these are expanded forms of clauses defining game state change.

For each clause $\mathcal{C} \in \mathrm{Next}_e$, we select two terms $s_\mathcal{C}$ and $t_\mathcal{C}$ in the following way. The term $s_\mathcal{C}$ is simply the second part of the head of the clause `(next` $s_\mathcal{C}$`)`. The term $t_\mathcal{C}$ is the argument of `true` in the body of $\mathcal{C}$ which is most similar to s in the sense of Definition 6, and of equally similar has smallest $d\mathcal{P}(s,t)$ (if there are several, we pick one in an arbitrary way).

We often use the word *fluent* for changing objects, and so we define the set of *fluent paths*, $\mathcal{P}_f$, in the following way. We say that a term t is a *negative true* in a clause $\mathcal{C}$ if it is the argument of a negative occurrence of `true` in $\mathcal{C}$. We write $\mathcal{L}(t)$ for the set of path to all constant leaves in t. The set

$$\mathcal{P}_f = \bigcup_{\mathcal{C} \in \mathrm{Next}_e} d\mathcal{P}(s_\mathcal{C}, t_\mathcal{C}) \cup \bigcup_{\mathcal{C} \in \mathrm{Next}_e,\ t_\mathcal{C} \text{ negative true in } \mathcal{C}} \mathcal{L}(t_\mathcal{C}).$$

Note that $\mathcal{P}_f$ contains all merge sets for the selected terms in Next_e clauses, and additionally, when $t_\mathcal{C}$ is a negative true, we add the paths to all constant leaves in $t_\mathcal{C}$.

Example 7. There are three `next` clauses in Figure 3. $\mathcal{C}_1$:

```
(<= (next (cell ?x ?y ?c))
  (true (cell ?x ?y ?c))
  (does ?r (mark ?x1 ?y1))
  (or (distinct ?x ?x1) (distinct ?y ?y1)))
```

does not lead to any fluent paths, as `(cell ?x ?y ?c)` is $s_{\mathcal{C}_1} = t_{\mathcal{C}_1}$ and thus $d\mathcal{P}(s_{\mathcal{C}_1}, t_{\mathcal{C}_1}) = \emptyset$. The clause:

```
(<= (next (cell ?x ?y ?r))
  (does ?r (mark ?x ?y)))
```

expands to:

```
(<= (next (cell ?x ?y x))
  (true (control x))
  (true (cell ?x ?y b)))
(<= (next (cell ?x ?y o))
  (true (control o))
  (true (cell ?x ?y b)))
```

These generate the fluent path $(\texttt{cell}, 3)$. The clause:

```
(<= (next (control ?r)) (does ?r noop))
```

expands to:

```
(<= (next (control x))
      (not (true (control x))))
(<= (next (control o))
      (not (true (control o))))
```

These generate the fluent path $(\texttt{control}, 1)$ since `(control x)` and `(control o)` are negative trues. In the end $\mathcal{P}_f = \{(\texttt{cell}, 3), (\texttt{control}, 1)\}$.

The fluent paths define the partition of GDL state terms into elements of the Toss structures in the following way.

Definition 8. We define the *element mask equivalence* $\sim$ by:

$$t \sim s \quad \Leftrightarrow \quad t[P_f \leftarrow c] = s[P_f \leftarrow c] \text{ for all terms } c.$$

The set of elements A of the initial Toss structure $\mathfrak{A}$ consists of equivalence classes of $\sim$. For $a \in A$ we write $[\![a]\!]$ to denote the corresponding subset of equivalent terms from $\mathcal{S}$.

We define *paths within mask* $\mathcal{P}_m$ as such paths p that, for all $a \in A$, if, for any $t \in [\![a]\!]$, $t \downarrow_p$, then for all $s, t \in [\![a]\!]$, $s \downarrow_p = t \downarrow_p$. For $p \in \mathcal{P}_m$ we can therefore define the *mask subterm* $a \downarrow_p^m$ as $t \downarrow_p$ for $t \in [\![a]\!]$.

Example 9. Continuing the example of the Tic-tac-toe specification from Figure 3, we construct the set A. The terms in $\mathcal{S}$ are either $(\texttt{cell}\ s\ t\ p)$ or $(\texttt{control}\ q)$, where s and t range over `a, b, c`, p over `x, o, b` and q can be `x` or `o`. Since $\mathcal{P}_f = \{(\texttt{cell}, 3), (\texttt{control}, 1)\}$, we consider as $\sim$-equivalent all `cell` terms which differ only on p and all `control` terms which differ on q. Thus, the set A consists of 10 elements: the element a_{ctrl} for the single equivalence class of `control` terms, and 9 elements $a_{s,t}$ for the equivalence classes of $(\texttt{cell}\ s\ t\ p)$ with fixed s and t.

$$\begin{aligned} A = \{a_{ctrl},\ & a_{\texttt{a},\texttt{a}},\ a_{\texttt{a},\texttt{b}},\ a_{\texttt{a},\texttt{c}}, \\ & a_{\texttt{b},\texttt{a}},\ a_{\texttt{b},\texttt{b}},\ a_{\texttt{b},\texttt{c}}, \\ & a_{\texttt{c},\texttt{a}},\ a_{\texttt{c},\texttt{b}},\ a_{\texttt{c},\texttt{c}}\}. \end{aligned}$$

Note the similarity to the starting structure in Figure 1, up to the control element. The set of paths within masks for this specification is $\mathcal{P}_m = \{(\texttt{cell}, 1), (\texttt{cell}, 2)\}$.

Relations in the Structure

Having defined the elements A as equivalence classes of state terms, let us now define the relations in the initial structure $\mathfrak{A}$.

Subterm equality relations. For all pairs of paths $p, q \in \mathcal{P}_m$ we introduce the *subterm equality relation* $Eq_{p,q}$:

$$Eq_{p,q}(a_1, a_2) \iff a_1 \downarrow_p^m = a_2 \downarrow_q^m.$$

Fact relations. For all relations R of G that do not (directly or indirectly) depend on the state, and all tuples of paths $p_1, \dots, p_n \in \mathcal{P}_m$, we introduce the *fact relation* $R_{p_1,\dots,p_n}$:

$$R_{p_1,\dots,p_n}(a_1, \dots, a_n) \iff R(a_1 \downarrow_{p_1}^m, \dots, a_n \downarrow_{p_n}^m) \text{ in any state.}$$

Anchor predicates. For all paths $p \in \mathcal{P}_m$ and subterms $s = t \downarrow_p, t \in \mathcal{S}$, we introduce the *anchor predicate* $Anch_p^s(a)$:

$$Anch_p^s(a) \iff a \downarrow_p^m = s.$$

Fluent predicates. Let $\mathcal{S}^{\text{init}} = \{s \mid \texttt{init}(s) \in G\}$ be the set of state terms under `init`. For all paths $p \in \mathcal{P}_f$ and subterms $s = t \downarrow_p, t \in \mathcal{S}$, we introduce the *fluent predicate* $Flu_p^s(a)$:

$$Flu_p^s(a) \iff t \downarrow_p = s \text{ for some } t \in [\![a]\!] \cap \mathcal{S}^{\text{init}}.$$

Mask predicates. We say that a term m is a *mask term* if the paths to all variables of m are contained in $\mathcal{P}_m \cup \mathcal{P}_f$ and for each $p \in \mathcal{P}_m \cup \mathcal{P}_f$ if p exists in m then $m \downarrow_p$ is a variable. We say that m *masks* a term t if m is a mask term and there exists a substitution σ such that $\sigma(m) = t$. For all mask terms $m \in \mathcal{S}$ we introduce the *mask predicate* $Mask_m$. Mask predicates are similar to the anchor predicates, but instead of matching against a subterm, they match against the mask.

$$Mask_m(a) \iff m \text{ masks all } t \in [\![a]\!].$$

Example 10. To list the relations derived for the Tic-tac-toe specification, recall that $\mathcal{P}_m = \{(\texttt{cell}, 1), (\texttt{cell}, 2)\}$ consists of two paths. To shorten notation, we will just use the index i for $(\texttt{cell}, i)$.

Subterm equality relations. The relation $Eq_{i,j}$ contains all pairs of elements for which the ith coordinate of the first one equals the jth coordinate of the second one. For example

$$\begin{aligned} Eq_{1,1} = \{&(a_{\texttt{a},\texttt{a}}, a_{\texttt{a},\texttt{a}}), (a_{\texttt{a},\texttt{a}}, a_{\texttt{a},\texttt{b}}), (a_{\texttt{a},\texttt{a}}, a_{\texttt{a},\texttt{c}}), \\ &\dots \\ &(a_{\texttt{c},\texttt{c}}, a_{\texttt{c},\texttt{a}}), (a_{\texttt{c},\texttt{c}}, a_{\texttt{c},\texttt{b}}), (a_{\texttt{c},\texttt{c}}, a_{\texttt{c},\texttt{c}})\} \end{aligned}$$

describes the identity of the first coordinate of two cells.

Fact relations. The only relation in the example specification is nextcol and we thus get the relations $\texttt{nextcol}_{i,j}$. For example, the relation

$$\texttt{nextcol}_{2,2} = \{(a_{\texttt{a,a}}, a_{\texttt{a,b}}), (a_{\texttt{a,a}}, a_{\texttt{b,b}}), (a_{\texttt{a,a}}, a_{\texttt{c,b}}), \ldots, (a_{\texttt{c,b}}, a_{\texttt{a,c}}), (a_{\texttt{c,b}}, a_{\texttt{b,c}}), (a_{\texttt{c,b}}, a_{\texttt{c,c}})\}$$

contains pairs in which the second element is in the successive row of the first one. Note that, for example, the formula $Eq_{1,1}(x_1, x_2) \wedge \texttt{nextcol}_{2,2}(x_1, x_2)$ specifies that x_2 is directly right of x_1 in the same row.

Anchor predicates. Since the terms inside cell at positions 1 and 2 range over a, b, c, we get 6 anchor predicates $Anch_i^{\texttt{a}}, Anch_i^{\texttt{b}}, Anch_i^{\texttt{c}}$ for $i = 1, 2$. They mark the corresponding terms, e.g.

$$Anch_2^{\texttt{a}} = \{a_{\texttt{a,a}}, a_{\texttt{b,a}}, a_{\texttt{c,a}}\}$$

describes the bottom row.

Fluent predicates. The fluent paths $\mathcal{P}_f = \{(\texttt{cell}, 3), (\texttt{control}, 1)\}$ and the terms appearing there are b, x, o for $(\texttt{cell}, 3)$ and x, o for $(\texttt{control}, 1)$, resulting in 5 fluent predicates. For example, $Flu^{\texttt{o}}_{(\texttt{cell},3)}(a)$ will hold exactly for the elements a which are marked by the player o. In the initial structure, the only nonempty fluent predicates are

$$Flu^{\texttt{b}}_{(\texttt{cell},3)} = A \smallsetminus \{a_{ctrl}\} \quad \text{and} \quad Flu^{\texttt{x}}_{(\texttt{control},1)} = \{a_{ctrl}\}.$$

Mask predicates. For the specification we consider, there are two mask terms: $m_1 = (\texttt{control}\ x)$ and $m_2 = (\texttt{cell}\ x\ y\ z)$. The predicate $Mask_{m_1} = \{a_{ctrl}\}$ holds exactly for the control element, and $Mask_{m_2} = A \smallsetminus \{a_{ctrl}\}$ contains these elements of A which are not the control element, i.e. the board elements.

In Toss, *stable relations* are relations that do not change in the course of the game, and *fluents* are relations that do change. Roughly speaking, a fluent occurs in the symmetric difference of the sides of a structure rewrite rule. In the translation, the fluent predicates Flu_p^s are the only introduced fluents, i.e. these predicates will change when players play the game and all other predicates will remain intact.

Structure Rewriting Rules

To create the structure rewriting rule for the Toss game, we first construct two types of clauses and then transform them into structure rewriting rules. Let $(p_1, \ldots, p_n)$ be the players in G, i.e. let there be (role p_1) up to (role p_n) facts in G, in this order.

Move Clauses By GDL specification, a legal joint move of the players is a tuple of player term – move term pairs which satisfy the legal relation. For a joint move $(m_1, \ldots, m_n)$ to be allowed, it is necessary that there is a tuple of legal clauses $(\mathcal{C}_1, ..., \mathcal{C}_n)$, with head of $\mathcal{C}_i$ being (legal p_i l_i), and the legal arguments tuple being more general than the joint move tuple, i.e. $m_i \le l_i$ for each $i = 1, \ldots, n$.

The move transition is computed from the next clauses whose all does relations are matched by respective joint move tuple elements as follows.

Definition 11. Let $\mathcal{N}$ be a next clause. The $\mathcal{N}$ *does facts*, $d_1(\mathcal{N}), \ldots, d_n(\mathcal{N})$, are terms, one for each player, constructed from $\mathcal{N}$ in the following way. Let (does p_i d_i^j) be all does facts in $\mathcal{N}$.

- If there is exactly one d_i for player p_i we set $d_i(\mathcal{N}) = d_i$.
- If there is no does fact for player p_i in $\mathcal{N}$ we set $d_i(\mathcal{N})$ to a fresh variable.
- If there are multiple $d_i^1, \ldots, d_i^k$ for player p_i we compute $\sigma = \mathrm{MGU}(d_i^1, \ldots, d_i^k)$ and set $d_i(\mathcal{N}) = \sigma(d_i^1)$.

We have $m_i \le d_i(\mathcal{N})$ for each next clause $\mathcal{N}$ contributing to the move transition, since otherwise the body of $\mathcal{N}$ would not match the state enhanced with (does p_i m_i) facts.

Example 12. In the Tic-tac-toe example, there are three clauses where the control player is o, which after renaming of variables look as follows.

```
N1 = (<= (next (control x)) (does x noop)),
N2 = (<= (next (cell ?x2 ?y2 o))
         (does o (mark ?x2 ?y2))),
N3 = (<= (next (cell ?x3 ?y3 ?c))
         (true (cell ?x3 ?y3 ?c))
         (does o (mark ?x1 ?y1))
         (or (distinct ?x3 ?x1) (distinct ?y3 ?y1))).
```

The does facts are, respectively,

$$d_1(\mathcal{N}_1) = \texttt{noop} \quad \text{and} \quad d_2(\mathcal{N}_1) = x_{f_1},$$
$$d_1(\mathcal{N}_2) = x_{f_2} \quad \text{and} \quad d_2(\mathcal{N}_2) = (\texttt{mark}\ x_2\ y_2),$$
$$d_1(\mathcal{N}_3) = x_{f_3} \quad \text{and} \quad d_2(\mathcal{N}_3) = (\texttt{mark}\ x_1\ y_1).$$

Each rewrite rule of the translated game is generated from a tuple of legal clauses $\mathcal{C}_1, \ldots, \mathcal{C}_n$ and a selection of next clauses $\mathcal{N}_1, \ldots, \mathcal{N}_m$, with variables renamed so that no variable occurs in multiple clauses, and such that

$$l_i \doteq d_i(\mathcal{N}_1) \doteq \ldots \doteq d_i(\mathcal{N}_m)$$

for each player p_i. We will consider all tuples $\overline{\mathcal{C}}, \overline{\mathcal{N}}$ for which the the above MGU exists and we will denote it by $\sigma_{\overline{\mathcal{C}},\overline{\mathcal{N}}}$. We apply $\sigma_{\overline{\mathcal{C}},\overline{\mathcal{N}}}$ to the clauses and we will refer to the result simply as *the* legal *and* next *clauses of the rule*.

Technically, for completeness, we need to generate a rule for a set of next clauses even if we generate a rule for its superset, and then for correctness, we need to preclude application of the more general rule when the more concrete rule is applicable, adding distinct conditions to clauses of the otherwise more general rule. In the current implementation, we only consider maximal sets of next clauses.

Example 13. Let $\mathcal{C}_1 = \texttt{noop}$ and $\mathcal{C}_2 = (\texttt{mark}\ x\ y)$. The clauses $\mathcal{N}_1, \mathcal{N}_2, \mathcal{N}_3$ introduced above form a maximal set,

$$\sigma_{\overline{\mathcal{C}},\overline{\mathcal{N}}} = \{x_{f_1} \mapsto (\texttt{mark}\ x\ y), \quad x_{f_2} \mapsto \texttt{noop}, \quad x_2 \mapsto x, \quad y_2 \mapsto y, \quad x_1 \mapsto x, \quad y_1 \mapsto y\}.$$

With all tuples $\overline{\mathcal{C}}, \overline{\mathcal{N}}$ selected and the MGU $\sigma_{\overline{\mathcal{C}},\overline{\mathcal{N}}}$ computed, we are almost ready to construct the rewriting rules. Still, for a fixed tuple $\overline{\mathcal{C}}, \overline{\mathcal{N}}$, we first need to compute erasure clauses to prevent constructing too general rules in the end.

Erasure Clauses So far, we have not accounted for the fact that rewrite rules of Toss only affect the matched part of the structure, while the GDL game definition explicitly describes the construction of the whole successive structure. We will say that a `next` clause is a *frame clause* if and only if it contains a `true` relation applied to a term equal to the `next` argument. Negating the frame clauses from the tuple $\overline{\mathcal{N}}$ and transforming them into *erasure clauses* will keep track of the elements that possibly lose fluents and ensure correct translation.

From the frame clauses in $\sigma_{\overline{\mathcal{C}},\overline{\mathcal{N}}}(\mathcal{N}_1),\ldots,\sigma_{\overline{\mathcal{C}},\overline{\mathcal{N}}}(\mathcal{N}_m)$, we select all (maximal) subsets J such that, clauses in J having the form (<= (`next` $\mathtt{s_i}$) $\mathtt{b_i}$), it holds

$$s_1 \doteq_f \ldots \doteq_f s_{|J|},$$

i.e. the arguments of `next` unify. Note that we use $\doteq_f$ instead of the standard unification, and by that we mean that the variables shared with the `legal` clauses $\overline{\mathcal{C}}$ are treated as constants. The reason is that these variables are not local to the clauses and must therefore remain intact.

Intuitively, the selected sets J describe a partition of the state terms that could possibly be copied without change by the rule we will generate for $\overline{\mathcal{C}},\overline{\mathcal{N}}$.

Let us write ρ for the f-MGU of $s_1,\ldots,s_{|J|}$. To compute the bodies of the erasure clauses, we negate the disjunction of substituted bodies of the frame clauses and bring this Boolean combination to disjunctive normal form (DNF), i.e. we compute conjunctions $e_1,\ldots,e_l$ such that

$$\neg(\rho(b_1) \vee \cdots \vee \rho(b_{|J|})) \equiv (e_1 \vee e_2 \ldots \vee e_l).$$

As the head of each erasure clause we use $\rho(s_1) = \cdots = \rho(s_{|J|})$, with the one technical change that we ignore the fluent paths in this term. We replace these fluent paths with `BLANK` and thus allow them to be deleted in case they are not preserved by other `next` clauses of the rule. Let us denote by h the term $\rho(s_1)$ after the above replacement. The erasure clauses $\mathcal{E}_{\overline{\mathcal{C}},\overline{\mathcal{N}}}(J) = \{(\texttt{<= h e}_1)\ldots(\texttt{<= h e}_l)\}$, and we write $\mathcal{E}_{\overline{\mathcal{C}},\overline{\mathcal{N}}}$ for the union of all $\mathcal{E}_{\overline{\mathcal{C}},\overline{\mathcal{N}}}(J)$, i.e. for the set of all $\overline{\mathcal{C}},\overline{\mathcal{N}}$ erasure clauses.

Example 14. In our example, $\mathcal{N}_3$ and its counterpart for the other player are the only frame clauses in G. After negation, $\sigma(\mathcal{N}_3)$ splits into several clauses e_i. The relevant one is `(<= (next (cell ?x3 ?y3 ?c)) (?x3 = ?x) (?y3 = ?y))`, i.e. `(<= (next (cell ?x ?y ?c)))`. The resulting erasure clause is `(<= (next (cell ?x ?y BLANK)))`. If no other clause had the form `(<= (next (cell ?x ?y ...)) ...)`, this clause would cause the erasure of any fluent at coordinates (x,y). Other erasure clauses derived from $\sigma(\mathcal{N}_3)$ turn out to be contradictory with remaining clauses, and thus will not contribute to any rewrite rule in the translation, due to filtering described below.

Rewriting Rule Creation For each suitable tuple $\overline{\mathcal{C}},\overline{\mathcal{N}}$ we have now created the unifier $\sigma_{\overline{\mathcal{C}},\overline{\mathcal{N}}}$ and computed the erasure clauses $\mathcal{E}_{\overline{\mathcal{C}},\overline{\mathcal{N}}}$. At this point, clauses $\overline{\mathcal{C}},\overline{\mathcal{N}}$ are optionally divided according to the player of the `does` relation atom in them. To create the rules, we need to further partition the *rule clauses* $\sigma_{\overline{\mathcal{C}},\overline{\mathcal{N}}}(\mathcal{C}_i),\sigma_{\overline{\mathcal{C}},\overline{\mathcal{N}}}(\mathcal{N}_i)$ and $\mathcal{E}_{\overline{\mathcal{C}},\overline{\mathcal{N}}}$, and augment them with further conditions. The reason is that the prepared rule clauses may have different matches in different game states, while the Toss rule has to be built from all the rule clauses that would match when the Toss rule matches. Therefore, we need to build a Toss rule for each subset of rule clauses that are "selected" by some game state (i.e. are exactly the rule clauses matching in that state), but add to it separation conditions that prevent the Toss rule from matching in game states where more rule clauses can match.

We select groups of atoms (collected from rule clauses) that separate rule clauses, and generate a Toss rule candidate for every partition of the groups into true and false ones: we collect the rule clauses that agree with the given partition. The selected atoms, some negated according to the partition, form the separation condition.

For each candidate, we will now construct the Toss rule in two steps. In the first step we generate the *matching condition*: we translate the conjunction of the bodies of rule clauses and the separation condition. This translation follows the definitions of atomic relations presented in Section , but we skip the full definition here.

Later we *filter* the rule candidates by checking for satisfiability in the initial structure of the stable part of the matching condition.

In the second step, we build a Toss rewrite rule itself. From the heads of rule clauses of a rule candidate, we build the $\mathfrak{R}$-structure: each `next` term, with its fluent paths replaced by `BLANK`, is an $\mathfrak{R}$ element, and the fluent predicates holding for the `next` state terms are the relations of $\mathfrak{R}$. The $\mathfrak{L}$-structure and the precondition of the Toss rule is built from the matching condition, based on elements of $\mathfrak{R}$. Quantification over variables corresponding to $\mathfrak{R}$ elements (which are the same as $\mathfrak{L}$ elements) is dropped, and atoms involving only these variables and not occurring inside disjunctions are extracted to be relations tuples in $\mathfrak{L}$.

Having constructed and filtered the rewriting rule candidates, we have almost completed the definition of $T(G)$. Payoff formulas are derived by instantiating variables standing for the `goal` values. The formulas defining the `terminal` condition and specific `goal` value conditions are translated as mentioned before, from disjunctions of bodies of their respective clauses.

Translating Moves between Toss and GDL

To play as a GDL client, we need to translate legal moves from G into Toss rule embeddings for $T(G)$, and conversely, the rule embeddings from $T(G)$ into moves of G.

In the incoming move case, we augment the Toss rewrite rules with constraints provided in the incoming move, try to embed each of the augmented rules, and return the single rule that matches and its unique embedding. Augmenting the rule is done in the following simple way: If the head of a `legal` clause of the rule contains a variable v at path q, a Toss variable x was derived from a game state term t such that $t \downarrow_p = v$, and the incoming move has term s at path q, then we add $Anch^s_p(x)$ to the precondition.

To translate the outgoing move, we recall the heads of the `legal` clauses of the rule that is selected, as we only need to substitute all their variables. To eliminate a variable v contained in the head of a `legal` clause of the rule, we look at the rule embedding; if $x \mapsto a$, x was derived from a game state term t such that $t \downharpoonleft_p = v$, and $a \downharpoonleft_p^m = s$, then we substitute v by s. The move translation function μ is thus constructed.

Game Simplification in Toss

Games automatically translated from GDL, as described above, are verbose compared to games defined manually for Toss. They are also inefficient, since the current solver in Toss works fast only for sparse relations.

Both problems are remedied by joining co-occurring relations. Relations which always occur together in a conjunction are replaced by their join when they are over the same tuple. Analogically, we eliminate pairs of atoms when the arguments of one relation are reversed arguments of the other.

In an additional simplification, we remove an atom of a stable relation which is included in, or which includes, another relation, when an atom of the other relation already states a stronger fact. For example, if $Positive \subseteq Number$, then $Positive(x) \wedge Number(x)$ simplifies to $Positive(x)$, and $Positive(x) \vee Number(x)$ simplifies to $Number(x)$.

The above simplifications can be applied to any Toss definition. We perform one more simplification targeted specifically at translated games: We eliminate $Eq_{p,q}(x, y)$ atoms when we detect that $Eq_{p,q}$-equivalence of x and y can be deduced from the remaining parts of the formula.

The described simplifications are stated in terms of manipulating formulas; besides formulas, we also apply analogous simplifications to the structures of the Toss game: the initial game state structure, and each $\mathfrak{L}$ and $\mathfrak{R}$ rule structures.

Experimental Results

The presented translation was implemented in (Toss), an open-source program with various logic functions as well as the presented game model and a GUI for the players.

After the simplification step described above, the translated games were very similar to the ones we defined manually in Toss. As far as a game could be translated (see the restrictions below), the resulting specification was almost as good for playing as the manual one. In Table 1 we show the results of playing a game translated from GDL against a manually specified one — the differences are negligible.

	Manual Wins	Translated Wins	Tie
Breakthrough	55%	45%	0%
Connect5	0%	0%	100%

Table 1: Manual against Translated on 2 sample games.

Discussion and Perspectives

The major restriction of the proposed translation scheme is its notion of fluent paths. It requires that the `next` clauses build terms locally, referring among others to a similar original `true` term, instead of propagating changes only from different terms. Consider an example from a specification of the game Connect4:

```
(<= (next (cell ?c ?y2 b)) (succ ?y1 ?y2)
 (true (cell ?c ?y1 b)) (distinct ?y1 6))
```

Here, without further modifications, $(\texttt{cell}, 2)$ is incorrectly detected as a fluent path. A sufficient condition to meet this restriction is that fluent paths $d\mathcal{P}(s_\mathcal{C}, t_\mathcal{C})$ point only to constant terms in $s_\mathcal{C}$. A possible workaround for games violating this condition is guessing that the offending clause is a frame clause and ignoring it, as is the case above.

Even if a game meets the major restriction above, the translation may not handle it or may take a long time. The reason is that it does not use Toss defined relations: it either translates GDL definition relations extensively as stable relations in the game state structure, or it expands (inlines) them before translation. To lift this restriction, we need to use definitions in the translation in an efficient way and translate relations depending on game state as defined relations.

Finally, the number of elements in the resulting game state structure can be prohibitive. It is a consequence of mapping state terms to Toss elements in such way that all fluents are predicates. Currently, an element a is identified with a mask m and a ground substitution for its $\mathcal{P}_m$ paths. If the number of elements falling under a mask m is too large, we would instead like to identify elements by the mask plus ground substitution of a subset of its $\mathcal{P}_m$ paths (for subsets that together cover all its $\mathcal{P}_m$ paths). Fluents corresponding to $\mathcal{P}_f$ paths in m would then be of arity equal to the number of such subsets. We would then define $[\![\cdot]\!] : A \to 2^{\mathcal{S}}$ in terms of masks and subsets of $\mathcal{P}_m$, which is closer to the implementation, rather than in terms of $\mathcal{P}_f$ as is done in this work.

References

Comon, H.; Dauchet, M.; Gilleron, R.; Löding, C.; Jacquemard, F.; Lugiez, D.; Tison, S.; and Tommasi, M. October 2007. Tree automata techniques and applications.

Genesereth, M. R., and Love, N. 2005. General game playing: Overview of the AAAI competition. *AI Magazine* 26(2):62–72.

Grädel, E., and Gurevich, Y. 1998. Metafinite model theory. *Information and Computation* 140:26–81.

Grädel, E. 2007. Finite model theory and descriptive complexity. In *Finite Model Theory and Its Applications*. Springer. 125–230.

Kaiser, L., and Stafiniak, L. 2011. First-order logic with counting for general game playing. In *Proc. of AAAI'11*.

Love, N.; Hinrichs, T.; Haley, D.; Schkufza, E.; and Genesereth, M. 2008. General game playing: Game description language specification. Technical report.

Rajlich, V. 1973. Relational structures and dynamics of certain discrete systems. In *Proc. of MFCS'73*, 285–292.

Toss. `http://toss.sourceforge.net`.