The Game Description Language is Turing Complete

Abdallah Saffidine

***Abstract*—We show that the Game Description Language (GDL) is *Turing complete*. In particular, we show how to simulate a Turing Machine (TM) as a single-player game described in GDL. Positions in the game correspond to configurations of the machine, and the TM accepts its input exactly when the agent has a winning strategy from the initial position.**

As direct consequences of the Turing completeness of GDL, we show that well-formedness as well as some other properties of a GDL description are undecidable.

We propose to strengthen the recursion restriction of the original GDL specification into a *general recursion restriction*. The restricted language is not Turing complete and the aforementioned properties become decidable. Checking whether a game description satisfies the suggested restriction is as easy as checking that the game is syntactically correct. Finally, we argue that practical expressivity is not affected as all syntactically correct games in a collection of more than 500 games having appeared in previous General Game Playing competitions belong to the proposed GDL fragment.

***Index Terms*—Game Description Language, Computability, Expressivity, General Game Playing, Turing Complete, Fragment.**

I. Introduction

The Game Description Language (GDL) is the domain modeling language for the General Game Playing (GGP) competitions [1], [2]. It can also be used to describe a variety of multi-agent situations (*games* for short).

In a match, each GGP engine has to select actions for one agent (its *role*) trying to maximize its utility (the *goal* value). At the start of the match, each GGP engine receives the rules of a game in GDL and its *role*. To be able to submit legal actions, the engine needs to understand GDL so as to interpret the rules of the game.

In this paper, we give a reduction from a deterministic Turing Machine (TM) to a single-agent game expressed in GDL. The reduction is quite short as it is linear in the size of the transition table of the TM. Positions in the game correspond to configurations of the machine, and the TM accepts the initial configuration if and only if there is a winning strategy for the agent from the initial position. The same reduction shows that determining whether a program in GDL is well-formed is undecidable.

We then move on defining a fragment of GDL called *bounded GDL*. The fragment we propose has several attractive features. Deciding well-formedness as well as other important properties is decidable for the fragment. The fragment is easy to recognize: there is an efficient algorithm determining whether an input program belongs to the fragment. The fragment is sufficiently expressive for practical purposes as all syntactically correct games from previous GGP competitions fall in the fragment.

The fact that GDL is Turing complete was already known to some authors ([3, page 9–10]) but this paper gives the first explicit reduction. A similar construction can be used for other languages such as Ludocore's modeling language [4].

abdallahs@cse.unsw.edu.au
CSE, The University of New South Wales, Sydney 2052, Australia

II. The Game Description Language

The Game Description Language (GDL) is a modeling language inspired by Datalog [2]. However GDL and Datalog are incomparable and have different semantics. We assume familiarity of the reader with the General Game Playing setting as well as with the Game Description Language, but we recall the syntax of the language as well as a few important properties satisfied by so-called well-formed programs. The natural Multi-Agent Environment (MAE) semantics of the language have been formally established [5], and an extension for imperfect information settings has recently been proposed [6].

A. Syntax

The concrete syntax used for GDL is drawn from the Knowledge Interchange Format and is presented in Definition 1 [2]. A program is composed of rules and a rule consists of a head and a body. The head is a term and the body is a logical constructions on literals. The keywords '`and`' and '`or`' keep their usual meaning, '`not`' is interpreted as negation-as-failure, and the '`distinct`' keyword allows to compare instantiated subterms.

Definition 1. The syntax of the Game Description Language is defined as follows, where *word* is an alphanumeric string.

⟨*program*⟩ ::= ⟨*rule*⟩+

⟨*rule*⟩ ::= ⟨*term*⟩ | '`(`' '`<=`' ⟨*term*⟩ ⟨*logic*⟩+'`)`'

⟨*logic*⟩ ::= ⟨*literal*⟩ | '`(`' '`or`' ⟨*logic*⟩+ '`)`' | '`(`' '`and`' ⟨*logic*⟩+ '`)`'

⟨*literal*⟩ ::= ⟨*term*⟩ | '`(`' '`not`' ⟨*term*⟩ '`)`'
| '`(`' '`distinct`' ⟨*subterm*⟩ ⟨*subterm*⟩'`)`'
| '`(`' '`not`' '`(`' '`distinct`' ⟨*subterm*⟩ ⟨*subterm*⟩ '`)`''`)`'

⟨*term*⟩ ::= ⟨*predicate*⟩ | '`(`'⟨*predicate*⟩ ⟨*subterm*⟩+'`)`'

⟨*subterm*⟩ ::= ⟨*variable*⟩ | ⟨*funsymb*⟩ | '`(`'⟨*funsymb*⟩ ⟨*subterm*⟩+ '`)`'

⟨*variable*⟩ ::= '`?`'⟨*word*⟩

⟨*funsymb*⟩ ::= ⟨*word*⟩

⟨*predicate*⟩ ::= '`base`' | '`does`' | '`goal`' | '`init`' | '`input`'
| '`legal`' | '`next`' | '`role`' | '`true`' | '`terminal`' | ⟨*word*⟩

For a program in the syntax of Definition 1 to be a valid GDL description, two restrictions need to be satisfied [2]. The *safety* restriction requires that every variable appearing in the head of a rule or in a non-positive literal of the body should

```
(⇐ (doop (= t1 t2) (= nt1 nt2) op n)
   (doop t1 nt1 op n)
   (doop t2 nt2 op n))
```

Fig. 1. Rule from `themathematician_easy` that does not satisfy the *recursion restriction*.

also appear in a positive literal of the body. The *recursion restriction* ensures that it is not possible to build terms of unbounded size within a frame. More precisely, for a rule R with head p and such that the body contains a predicate q, if p and q are part of some cycle of the dependency graph, then at least one of the following must hold for every argument v_j of q in R. Either v_j is ground, or v_j appears as an argument of p in R, or v_j appears as an argument of another predicate r of R such that r and p are not part of any single cycle of the dependency graph.

Example 1. *Consider the GDL rule presented in Fig. 1. The predicate* `doop` *appears both in the body and in the head of the rule, and so the head of the rule and the first hypothesis are part of a cycle in the dependency graph.* t_1 *is a non-ground argument of the* `doop` *predicate in the body, but* t_1 *does not appear in any other predicate of the body. Although it appears somewhere in the head,* t_1 *is not a direct argument of the head predicate. Therefore, the rule violates the* recursion restriction.

B. Typical properties of GDL programs

While the above definition allows to represent multi-agent systems, it is desirable for General Game Playing purposes that they satisfy additional constraints. The following definitions are adapted from Definition 21 to 25 in the GDL specification [2].[1]

Definition 2 (Termination)**.** A game description in GDL *terminates* if all infinite sequences of legal moves from the initial state of the game reach a terminal state after a finite number of steps.

Definition 3 (Playability)**.** A game description in GDL is *playable* if and only if every role has at least one legal move in every non-terminal state reachable from the initial state.

Definition 4 (Utility structure)**.** A game description in GDL has a *utility structure* if and only if every role has exactly one goal value in every reachable terminal state.

Definition 5 (Winnability)**.** A game description in GDL is *weakly winnable* if and only if, for every role, there is a sequence of joint moves of all roles that leads to a terminal state where that role's goal value is maximal.

Definition 6 (Well-formed Games)**.** A game description in GDL is *well-formed* if it terminates, has a utility structure, and is both playable and weakly winnable.

[1]We have replaced the original *monotonicity* property (Definition 23 from [2]) with the weaker *utility structure* property (Definition 4). Monotonicity is actually not a sensible requirement when modeling competitive games, in particular constant-sum games. Additionally most game descriptions in practice are not monotonic but do have a utility structure.

GGP competition organizers have tried to restrict the benchmark games to well-formed GDL. However, we will show that determining whether an arbitrary game description is well-formed is actually an undecidable problem.

III. Turing completeness

We will show the computational power underlying GDL by demonstrating that it can be used to simulate a TM. The resulting GDL program is not necessarily well-formed but it is safe and it satisfies the recursion restriction.

A large variety of computational models have been shown to be as expressive as TMs. For instance, lambda calculus, cellular automata, and most programming languages (assuming unbounded memory) are all Turing complete.

Obtaining that a formalism F is Turing complete can be seen both as good news and as bad news. On the one hand, it shows that F is expressive enough to represent any situation that can be computed in the physical world. On the other hand, it signifies that a quantity of properties on programs of F are undecidable in general.

A. Turing Machines

Many equivalent definitions of a deterministic TM have been proposed [7]. Since they are equivalent, one usually chooses the definition that impedes the lightest notation burden for the intended application. In this paper we use Definition 7.

Let Γ be a set, we use $\Gamma^{<\mathbb{N}}$ to denote the set of finite sequences of elements of Γ. That is, $\Gamma^{<\mathbb{N}} = \{(t_1, \ldots, t_n) | n \in \mathbb{N}, t_1 \in \Gamma, \ldots, t_n \in \Gamma\}$. Note that such sets of finite sequences contain in particular the empty sequence $()$.

Definition 7. A one-tape deterministic *Turing Machine (TM)* is a structure $\langle Q, \Gamma, b, q_\top, q_\bot, \delta, q_0, t\rangle$ where

- Q is a finite, non-empty set of states
- Γ is a finite, non-empty set of the tape alphabet
- $b \in \Gamma$ is the blank symbol
- $q_\top \in Q$ is the accepting state and $q_\bot \in Q$ is the rejecting state, these two states are distinct: $q_\top \neq q_\bot$
- $\delta : Q \setminus \{q_\top, q_\bot\} \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function, where L is left shift, R is right shift.
- $q_0 \in Q$ is the initial state.
- $t = (t_0, \ldots, t_{n-1}) \in \Gamma^{<\mathbb{N}}$ for some integer n is the input.

A quadruple $\langle q, l, a, r\rangle$ with $q \in Q$, $l = (l_1, \ldots, l_{n_l}) \in \Gamma^{<\mathbb{N}}$, $a \in \Gamma$, and $r = (r_1, \ldots, r_{n_r}) \in \Gamma^{<\mathbb{N}}$ is called a *configuration*. The first element is the *current state*, the second element is the *left part of the tape*, the third element is the currently *read symbol*, and the fourth element is *the right part of the tape*. Although the tape is assumed to be infinite, it only contains a finite number of non-blank symbols so it can be described with two finite sequences.

The initial configuration is $\langle q_0, (), t_0, (t_1, \ldots, t_{n-1})\rangle$ when the input is $(t_0, \ldots, t_{n-1})$. If the input is $()$ then the initial configuration is $\langle q_0, (), b, ()\rangle$. This means that we use the input to determine the initially read symbol and the right part of the tape. The left part of the tape is initially empty.

A configuration is *accepting* if the current state is $q_\top$, *rejecting* if the current state is $q_\bot$, and non-terminal otherwise.

If the machine is in a non-terminal configuration, then the transition function is used to determine the next configuration that the machine is in. The first element of a triple returned by the transition function is the next state that the machine is in, the second element is the symbol that will overwrite the previously read symbol, and the third element is the shifting symbol which tells whether the reading head of the machine moves towards the right or the left.

For a non-terminal configuration $\langle q, l, a, r\rangle$ with $q \notin \{q_\top, q_\bot\}$, let $(q', a', s) = \delta(q, a)$. If the shifting symbol is left, $s = L$, then we define the *next configuration* as $\langle q', (l_2, \ldots, l_{n_l}), l_1, (a', r_1, \ldots, r_{n_r})\rangle$. If the shifting symbol is right, the next configuration is defined in a symmetrical way. We may add blank symbols as needed when we reach the end of the specified right or left part of the tape.

A *run* of the TM is a finite or infinite sequence of configuration $c_0, c_1, \ldots$ such that c_0 is the initial configuration, for each i, c_{i+1} is the next configuration of c_i, and if the sequence is finite then the last configuration is accepting or rejecting. Note that for a deterministic TM, the transition function and the input uniquely determine the run. A TM is said to *accept the input* if the corresponding run is finite and ends in an accepting configuration.

Theorem 1 (Halting problem for Turing Machines [8], [7])**.** *In general, the problem of determining whether an arbitrary deterministic Turing Machine accepts the input is undecidable.*

The results remains true even when the reject state cannot be reached. This means that determining whether the run is finite at all is undecidable in general.

B. Reduction to GDL

We now show how a TM can be specified using GDL. A GDL description representing a TM comprises two parts, a preamble independent of the machine and a machine-specific part.

The preamble is displayed in Fig. 2 using the concrete GDL syntax to describe the rules, with variables italicized rather than starting with a question mark for the sake of legibility.

- i, j, and k are symbol variables, their domain is one-to-one correspondence with Γ.
- q is a state variable, its domain corresponds to Q.
- d is a direction variable, its domain is $\{\texttt{left}, \texttt{right}\}$.
- l and r are variables that respectively represent the left and right parts of the tape. They represent finite sequences with a lisp-like chained list.
- Finally, `zero` represents the blank symbol, `accept` and `reject` represent the terminal states $q_\top$ and $q_\bot$.

The machine-specific part is based on the transition function and the initial configuration. We add a rule for each possible transition. For every state $q \in Q \setminus \{q_\top, q_\bot\}$, for every symbol a, if the transition gives $\delta(q, a) = (q', a', s)$ then we add a rule with head `(legal pl (move` q' a' s`))` and with two positive literals in the body `(true (state` q`))` and `(read` a`)`.

The initial configuration of the TM is defined through the set up of initial values for fluents. The initial state q_0 is directly translated as `(init (state` q_0`))` and the input $(t_0, \ldots, t_{n-1})$ is translated as `(init (tape nil` t_0 `(cons` t_1 `(...(cons` t_{n-1} `nil)...)))))`. If $n = 0$ then we have `(init (tape nil zero nil))` instead.

```
(role pl)
(⇐ (next (tape (cons j l) k r))
   (true (tape l i (cons k r)))
   (does pl (move q j right)))
(⇐ (next (tape (cons j l) zero nil))
   (true (tape l i nil))
   (does pl (move q j right)))
(⇐ (next (tape l i (cons j r)))
   (true (tape (cons i l) k r))
   (does pl (move q j left)))
(⇐ (next (tape nil zero (cons j r)))
   (true (tape nil k r))
   (does pl (move q j left)))
(⇐ (next (state q))
   (does pl (move q j d)))
(⇐ terminal (true (state accept)))
(⇐ terminal (true (state reject)))
(⇐ (goal pl 100) (true (state accept)))
(⇐ (goal pl   0) (true (state reject)))
(⇐ (read j) (true (tape l j r)))
```

Fig. 2. Generic rules where i, j, and k are symbol variables, l and r are tape variables, and q is a state variable. d is a direction variable.

TABLE I
TRANSITION TABLE FOR THE 3-STATE, 2-SYMBOL BUSY BEAVER. COLUMNS INDICATE THE CURRENT STATE, ROWS INDICATE THE SYMBOL CURRENTLY READ, AND CELLS INDICATE THE TRANSITION TO BE TAKEN.

	q_0	q_1	q_2
0	$(q_1, 1, R)$	$(q_2, 0, R)$	$(q_2, 1, L)$
1	$(q_\top, 1, R)$	$(q_1, 1, R)$	$(q_0, 1, L)$

Example 2. *A busy-beaver with k states and m symbols is a TM that accepts the empty input after the longest finite run among all machines with k states (not counting the accepting and rejecting states) and m symbols. For instance, the busy-beaver for 3-state, 2-symbol deterministic TM is $\langle\{q_0, q_1, q_2, q_\top, q_\bot\}, \{0, 1\}, 0, q_\top, q_\bot, \delta, q_0, ()\rangle$ where δ is given by the transition table presented in Table I. The GDL simulating it is given by taking the machine-agnostic preamble in Fig. 2 together with the specific code in Fig. 3.*

It is important to note that the description obtained with the proposed reduction satisfy the recursion restriction. This is clear from the fact that there is no cycle in the dependency graph of the rules. Thus, the construction generates syntactically correct GDL descriptions.

There is exactly one occurrence of the `state` and `tape` fluents in any reachable non-terminal position in the game defined by this translation. As a result, the player `pl` has always exactly one legal move. Similarly, we can see that any reachable terminal position has exactly one `state` fluent, either `accept` or `reject`. Therefore, exactly one goal value is associated to the unique role in terminal states. This shows that the proposed description is *playable* and has a *utility*

```
(⇐ (legal pl (move q1 one right))
    (true (state q0))
    (read zero))
(⇐ (legal pl (move accept one right))
    (true (state q0))
    (read one))
(⇐ (legal pl (move q2 zero right))
    (true (state q1))
    (read zero))
(⇐ (legal pl (move q1 one right))
    (true (state q1))
    (read one))
(⇐ (legal pl (move q2 one left))
    (true (state q2))
    (read zero))
(⇐ (legal pl (move q0 one left))
    (true (state q2))
    (read one))
(init (tape nil zero nil))
(init (state q0))
```

Fig. 3. GDL code corresponding to the transition function and the initial configuration of the 3-state 2-symbol busy-beaver TM given in Example 2.

structure, but it does not guarantee well-formedness.

Theorem 2. *Deciding whether a GDL description is well-formed is undecidable in general.*

Proof. Being able to tell whether a description leads to games that always terminates would solve the halting problem for TM. Thus, knowing whether a given GDL program terminates is undecidable. Since a GDL description needs to terminate to be well-formed, the hardness is propagated. □

IV. Fragments of the Game Description Language

While the Turing completeness of a language shows that the language is very expressive, it comes with a cost. Testing for many properties becomes undecidable (see Section V). One way to address this trade-off between expressivity and computability is to define fragments of the language.

A fragment is particularly interesting if it satisfies the following three criteria.

Decidability The fragment is constrained enough that interesting properties are (efficiently) computable.

Recognition It is easy to determine whether a given program belongs to the fragment.

Expressivity The fragment is expressive enough to represent problems people are interested in.

These criteria are not formal but we can get a better sense of the intended meaning by looking at a few examples. The GDL fragments detailed in Example 3, 4, and 5 are three natural fragments but each of them fails to satisfy one of the desired criteria.

Example 3. *The* full GDL *fragment is simply the Game Description Language without any additional restriction. Any game expressed in GDL belongs to the fragment so the recognition criterion is satisfied. Similarly, the expressivity criterion is also fulfilled. However this fragment is not constrained at all which has a negative impact on the decidability criterion. In particular, it is undecidable whether a program is* well-formed*.*

Example 4. Well-formed GDL *is yet another plausible fragment. It contains all games expressed in GDL that satisfy the properties in Definition 6. It can be shown that most properties are decidable on this fragment, and GDL programmers have always tried to write well-formed games so we can consider that the decidability and the expressivity criteria are satisfied for this fragment. Unfortunately, a consequence of the reduction described in Section III-B is that determining whether a program terminates is undecidable in general. Thus, the recognition of this fragment is undecidable.*

Example 5. *The propositional GDL fragment offers another trade-off. Game descriptions are in* propositional GDL *when they make no use of variables. Recognition is easy and most interesting properties of propositional programs are decidable. However, relatively few games are expressed in propositional form as it can need an exponentially larger description compared to representations with variables.*

We have seen that descriptions satisfying the *recursion restriction* could only grow terms of bounded size within a frame. Still, it is possible to grow terms of unbounded size through the course of successive frames. To prevent this, we define a new restriction adapted from the recursion restriction.

Definition 8. Let Δ be a GDL program. Consider Δ' to be the GDL program Δ extended with the three following rules.

- `(⇐ (true` f`) (next` f`))`,
- `(⇐ (true` f`) (init` f`))`, and
- `(⇐ (does` r m`) (legal` r m`))`.

We say that Δ satisfies the *general recursion restriction* exactly when Δ' satisfies the recursion restriction. We call the fragment of such GDL programs *bounded GDL*.

The additional rules indicate that function constants appearing under the predicates `next` or `init` could appear under the predicate `true`, and similarly, anything appearing under `legal` could also appear under `does`.

Example 6. *Turing machines constructed with the reduction proposed in Section III-B do not belong to* bounded GDL*. Indeed, consider the following rules found in the extended program of any such machine.*

```
(⇐ (next (tape (cons j l) k r))
    (true (tape l i (cons k r)))
    (does pl (move q j right)))
(⇐ (true f) (next f))
```

We see that `next` *and* `true` *are part of a cycle of the dependency graph. Additionally, the argument* l *of the predicate* `true` *in the body does not appear in any other predicate of the body, and it does not appear as a direct argument of the head predicate* `next`*. Therefore, the extended program does not satisfy the recursion restriction and the original program does not satisfy the general recursion restriction.*

Adapting the recursion restriction from the original GDL

specification makes bounded GDL very easy to recognize as the general recursion restriction is only a syntactic property of the description of a program.

Games in bounded GDL also enjoy nice decidability properties. Since it is not possible to grow terms arbitrarily, then the domain of each predicate and each function constant is finite, therefore a finite grounding of the rules is possible. As a consequence, the number of distinct game states is finite.

Therefore, a game is guaranteed to terminate exactly when there are no cycles in the state space. This acyclicity can be checked with a depth-first traversal of the state space.

A depth-first traversal of the state space will also reveal whether the properties needed for well-formedness always hold (see Definition 6). Thus, well-formedness is decidable.

We now give evidence that the proposed fragment of GDL is sufficiently expressive for most practical purposes.

A good indication of practical expressivity is the proportion of existing programs that belong to the fragment. Many GGP competitions have been organized since the first one at AAAI and a large number of games were defined each time. Games that have appeared in various competitions can be retrieved from Sam Schreiber's GGP website http://games.ggp.org.

We implemented a tool to detect whether a program satisfied the general recursion restriction and used it to classify games available online. The collection contains a bit more than 550 game descriptions, but 489 descriptions are actually unique up to white space addition and removal.

In this collection, two games are not syntactically correct GDL. These games are `themathematician_easy` and `themathematician_medium`, two variants of the same puzzle where a single agent is required to perform algebraic manipulations to prove that a term, represented as a tree, can be transformed in another one using mathematical properties such as associativity or commutativity. These two games include the rule presented in Fig. 1, so they do not satisfy the recursion restriction. All 487 other games were found to satisfy the general recursion restriction.

It is possible to create games that are well-formed GDL but do not belong to bounded GDL, the busy-beaver given in Fig. 2 and 3 being an example. However, for all practical purpose, we conclude that restricting oneself to bounded GDL does not significantly impair the usability of GDL to express games.

V. Consequences

Theorem 2 stands in contradiction with a claim in the original GDL specification [2].

> Checking game descriptions to see if they are well-formed can certainly be done in general by using brute-force methods (exploring the entire game tree).

The reason brute force cannot be applied is that there is no guarantee that the game tree is finite when the corresponding GDL description is not assumed to be well-formed. Even if it was not for cycles in the state space, there is not even a guarantee that the state space itself is finite.

We believe the misunderstanding in the specification comes from the fact that the recursion restriction only guarantees a bounded growth in the nesting depth of function constants from one frame to the next. In particular, it does not prevent an unbounded growth over the whole game.

A few approaches have been put forward to ground GDL rules, including using an off-the-shelf grounder from the Answer Set Programming community, using Prolog queries, and using dependency graphs [9], [10]. However, since the state space might be infinite when the description is not assumed to be well-formed, there might be no finite groundings of the rules. Hence, any attempt to show the correctness of such a general grounding approach needs to explicitly use well-formedness of the input GDL rules in the proof.

Beside well-formedness, it is possible to look for other properties of GDL descriptions that might be undecidable. For instance, a model-checking approach to GDL was suggested based on the Alternating-time Temporal Logic (ATL) [11]. In the ATL approach, the rules were assumed to be given in a finite propositional form. If that assumption was lifted so as to address general GDL programs with variables, undecidability of most of the presented model-checking problems would lie around the corner.

Games have been studied from a computational complexity point of view for long time [12]. Typical complexity results address generalizations of single games. A direction for future work would be to establish direct relationships between fragments of GDL and complexity classes.

Acknowledgment

The author would like to thank the reviewers for their useful suggestions. This research was supported under Australian Research Council's (ARC) *Discovery Projects* funding scheme (project DP 120102023).

References

[1] M. Genesereth and N. Love, "General Game Playing: Overview of the AAAI competition," *AI Magazine*, vol. 26, pp. 62–72, 2005.

[2] N. C. Love, T. L. Hinrichs, and M. R. Genesereth, "General Game Playing: Game Description Language specification," LG-2006-01, Stanford Logic Group, Tech. Rep., 2006.

[3] S. Haufe, S. Schiffel, and M. Thielscher, "Automated verification of state sequence invariants in General Game Playing," *Artificial Intelligence*, vol. 187, pp. 1–30, 2012.

[4] A. M. Smith, M. J. Nelson, and M. Mateas, "Ludocore: A logical game engine for modeling videogames," in *2010 IEEE Symposium on Computational Intelligence and Games (CIG)*, 2010, pp. 91–98.

[5] S. Schiffel and M. Thielscher, "A multiagent semantics for the Game Description Language," in *Agents and Artificial Intelligence (ICAART)*, ser. Communications in Computer and Information Science, J. Filipe, A. Fred, and B. Sharp, Eds. Berlin / Heidelberg: Springer, 2010, vol. 67, pp. 44–55.

[6] M. Thielscher, "A general game description language for incomplete information games," in *24th AAAI Conference on Artificial Intelligence (AAAI)*. Atlanta: AAAI Press, Jul. 2010, pp. 994–999.

[7] S. Arora and B. Barak, *Computational Complexity — a Modern Approach*. Cambridge University Press, Jun. 2009.

[8] A. M. Turing, "On computable numbers, with an application to the Entscheidungsproblem," *Proceedings of the London mathematical society*, vol. 42, no. 2, pp. 230–265, 1936.

[9] M. Thielscher, "Answer Set Programming for single-player games in General Game Playing," in *25th International Conference on Logic Programming (ICLP)*, ser. Lecture Notes in Computer Science, P. M. Hill and D. S. Warren, Eds., vol. 5649. Springer, 2009, pp. 327–341.

[10] P. Kissmann and S. Edelkamp, "Instantiating general games using Prolog or dependency graphs," in *KI 2010: Advances in Artificial Intelligence*. Springer, 2010, pp. 255–262.

[11] J. Ruan, W. Van Der Hoek, and M. Wooldridge, "Verification of games in the Game Description Language," *Journal of Logic and Computation*, vol. 19, no. 6, pp. 1127–1156, 2009.
[12] R. A. Hearn and E. D. Demaine, *Games, Puzzles, and Computation*. A K Peters, Jul. 2009.