Alpaga: A Tool for Solving Parity Games with Imperfect Information

Dietmar Berwanger[1], Krishnendu Chatterjee[2], Martin De Wulf[3], Laurent Doyen[4], and Thomas A. Henzinger[4]

[1] LSV, ENS Cachan and CNRS, France
[2] CE, University of California, Santa Cruz, U.S.A.
[3] Université Libre de Bruxelles (ULB), Belgium
[4] École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Abstract. Alpaga is a solver for two-player parity games with imperfect information. Given the description of a game, it determines whether the first player can ensure to win and, if so, it constructs a winning strategy. The tool provides a symbolic implementation of a recent algorithm based on antichains.

1 Introduction

Alpaga is a tool for solving parity games with imperfect information. These are games played on a graph by two players; the first player has imperfect information about the current state of the play. We consider objectives over infinite paths specified by parity conditions that can express safety, reachability, liveness, fairness, and most properties commonly used in verification. Given the description of a game, the tool determines whether the first player has a winning strategy for the parity objective and, if this is the case, it constructs such a winning strategy.

The Alpaga implementation is based on a recent technique using *antichains* for solving games with imperfect information efficiently [2], and for representing the strategies compactly [1]. To the best of our knowledge, this is the first implementation of a tool for solving parity games with imperfect information.

In this paper, we outline the antichain technique which is based on fixed-point computations using a compact representation of sets. Our algorithm essentially iterates a *controllable predecessor* operator that returns the states from which a player can force the play into a given target set in one round. For computing this operator, no polynomial algorithms is known. We propose a new symbolic implementation based on BDDs to avoid the naive enumerative procedure.

Imperfect-information games arise in several important applications related to verification and synthesis of reactive systems. The following are some key applications: (a) synthesis of controllers for plants with unobservable transitions; (b) distributed synthesis of processes with private variables not visible to other processes; (c) synthesis of robust controllers; (d) synthesis of automata specifications where only observations of automata are visible, and (e) the decision and simulation problem of quantitative specification languages; (f) model-checking secrecy and information flow. We believe that the tool Alpaga will make imperfect information games a useful framework for designers in the above applications. In the appendix, we present a concrete example of distributed-system synthesis. Along the lines of [3], we consider the design of a mutual-exclusion protocol for two processes. The tool Alpaga is able to synthesize a winning strategy for a requirement of mutual exclusion and starvation freedom which corresponds to Peterson's protocol.

2 Games and Algorithms

Let $\mathit{\Sigma}$ be a finite alphabet of actions and let $\mathit{\Gamma}$ be a finite alphabet of observations. A *game structure with imperfect information* over $\mathit{\Sigma}$ and $\mathit{\Gamma}$ is a tuple $G = (L, l_0, \mathit{\Delta}, \gamma)$, where

– L is a finite set of locations (or states),
– $l_0 \in L$ is the initial location,
– $\mathit{\Delta} \subseteq L \times \mathit{\Sigma} \times L$ is a set of labelled transitions such that for all $\ell \in L$ and all $a \in \mathit{\Sigma}$, there exists $\ell' \in L$ such that $(\ell, a, \ell') \in \mathit{\Delta}$, i.e., the transition relation is total,
– $\gamma : \mathit{\Gamma} \to 2^L \setminus \emptyset$ is an observability function that maps each observation to a set of locations such that the set $\{\gamma(o) \mid o \in \mathit{\Gamma}\}$ partitions L. For each $\ell \in L$, let $\mathsf{obs}(\ell) = o$ be the unique observation such that $\ell \in \gamma(o)$.

The game on G is played in rounds. Initially, a token is placed in location l_0. In every round, Player 1 first chooses an action $a \in \mathit{\Sigma}$, and then Player 2 moves the token to an a-successor ℓ' of the current location ℓ, i.e., such that $(\ell, a, \ell') \in \mathit{\Delta}$. Player 1 does not see the current location ℓ of the token, but only the observation $\mathsf{obs}(\ell)$ associated to it. A *strategy* for Player 1 in G is a function $\alpha : \mathit{\Gamma}^+ \to \mathit{\Sigma}$. The set of possible *outcomes* of α in G is the set $\mathsf{Outcome}(G, \alpha)$ of sequences $\pi = \ell_1 \ell_2 \dots$ such that $\ell_1 = l_0$ and $(\ell_i, \alpha(\mathsf{obs}(\ell_1 \dots \ell_i)), \ell_{i+1}) \in \mathit{\Delta}$ for all $i \geq 1$. A *visible parity condition* on G is defined by a function $p : \mathit{\Gamma} \to \mathbb{N}$ that maps each observation to a non-negative integer priority. We say that a strategy α for Player 1 is *winning* if for all $\pi \in \mathsf{Outcome}(G, \alpha)$, the least priority that appears infinitely often in π is even.

To decide whether Player 1 is winning in a game G, the basic approach consists in tracing the *knowledge* of Player 1, represented a set of locations called a *cell*. The initial knowledge is the cell $s_0 = \{l_0\}$. After each round, the knowledge s of Player 1 is updated according to the action a she played and the observation o she receives, to $s' = \mathsf{post}_a(s) \cap \gamma(o)$ where $\mathsf{post}_a(s) = \{\ell' \in L \mid \exists \ell \in s : (\ell, a, \ell') \in \mathit{\Delta}\}$.

Antichain algorithm. The antichain algorithm is based on the *controllable predecessor* operator $\mathsf{CPre} : 2^S \to 2^S$ which, given a set of cells q, computes the set of cells q' from which Player 1 can force the game into a cell of q in one round:

$$\mathsf{CPre}(q) = \{s \subseteq L \mid \exists a \in \mathit{\Sigma} \cdot \forall o \in \mathit{\Gamma} : \mathsf{post}_a(s) \cap \gamma(o) \in q\}. \quad (1)$$

The key of the algorithm relies on the fact that $\mathsf{CPre}(\cdot)$ preserves downward-closedness. A set q of cells is *downward-closed* if, for all $s \in q$, every subset $s' \subseteq s$ is also in q. Downward-closed sets q can be represented succinctly by their maximal elements $r =$

$\lceil q \rceil = \{s \in q \mid \forall s' \in q : s \not\subset s'\}$, which form an *antichain*. With this representation, the controllable predecessor operator is defined by

$$\mathsf{CPre}(r) = \Big\lceil \{s \subseteq L \mid \exists a \in \Sigma \cdot \forall o \in \Gamma \cdot \exists s' \in r : \mathsf{post}_a(s) \cap \gamma(o) \subseteq s'\} \Big\rceil. \qquad (2)$$

Strategy construction. The implementation of the strategy construction is based on [1]. The algorithm of [1] employs antichains to compute winning strategies for imperfect-information parity games in an efficient and compact way: the procedure is similar to the classical algorithm of McNaughton [4] and Zielonka [6] for perfect-information parity games, but, to preserve downwards closure, it avoids the complementation operation of the classical algorithms by recurring into subgames with an objective obtained as a boolean combination of reachability, safety, and reduced parity objectives.

Strategy simplification. A strategy in a game with imperfect information can be represented by a set $\Pi = \{(s_1, \mathsf{rank}_1, a_1), \dots, (s_n, \mathsf{rank}_n, a_n)\}$ of triples $(s_i, \mathsf{rank}_i, a_i) \in 2^L \times \mathbb{N} \times \Sigma$ where s_i is a cell, and a_i is an action. Such a triple assigns action a_i to every cell $s \subseteq s_i$; since a cell s may be contained in many s_i, we take the triple with minimal value of rank_i. Formally, given the current knowledge s of Player 1, let $(s_i, \mathsf{rank}_i, a_i)$ be a triple with minimal rank in Π such that $s \subseteq s_i$ (such a triple exists if s is a winning cell); the strategy represented by Π plays the action a_i in s.

Our implementation applies the following rules to simplify the strategies and obtain a compact representation of winning strategies in parity games with imperfect information.

(*Rule* 1) In a strategy Π, retain only elements that are maximal with respect to the following order: $(s, \mathsf{rank}, a) \succeq (s', \mathsf{rank}', a')$ if $\mathsf{rank} \leq \mathsf{rank}'$ and $s' \subseteq s$. Intuitively, the rule specifies that we can delete (s', rank', a') whenever all cells contained in s' are also contained in s; since $\mathsf{rank} \leq \mathsf{rank}'$, the strategy can always choose (s, rank, a) and play a.

(*Rule* 2) In a strategy Π, delete all triples $(s_i, \mathsf{rank}_i, a_i)$ such that there exists $(s_j, \mathsf{rank}_j, a_j) \in \Pi$ $(i \neq j)$ with $a_i = a_j$, $s_i \subseteq s_j$ (and hence $\mathsf{rank}_i < \mathsf{rank}_j$ by Rule 1), such that for all $(s_k, \mathsf{rank}_k, a_k) \in \Pi$, if $\mathsf{rank}_i \leq \mathsf{rank}_k < \mathsf{rank}_j$ and $s_i \cap s_k \neq \emptyset$, then $a_i = a_k$. Intuitively, the rule specifies that we can delete $(s_i, \mathsf{rank}_i, a_i)$ whenever all cells contained in s_i are also contained in s_j, and the action a_j is the same as the action a_i. Moreover, if a cell $s \subseteq s_i$ is also contained in s_k with $\mathsf{rank}_i \leq \mathsf{rank}_k < \mathsf{rank}_j$, then the action played by the strategy is also $a_k = a_i = a_j$.

3 Implementation

Computing $\mathsf{CPre}(\cdot)$ is likely to require time exponential in the number of observations (a natural decision problem involving $\mathsf{CPre}(\cdot)$ is NP-hard [1]). Therefore, it is natural to let the BDD machinery evaluate the universal quantification over observations in (2). We present a BDD-based algorithm to compute $\mathsf{CPre}(\cdot)$.

Let $L = \{\ell_1, \dots, \ell_n\}$ be the state space of the game G. A cell $s \subseteq L$ can be represented by a valuation v of the boolean variables $\bar{x} = x_1, \dots, x_n$ such that $\ell_i \in s$

iff $v(x_i) = \mathsf{true}$, for all $1 \leq i \leq n$. A BDD over $x_1, \dots, x_n$ is called a *linear encoding*, it encodes a set of cells. A cell $s \subseteq L$ can also be represented by a BDD over boolean variables $\bar{y} = y_1, \dots, y_m$ with $m = \lceil \log_2 n \rceil$. This is called a *logarithmic encoding*, it encodes a single cell.

We represent the transition relation of G by the $n \cdot |\Sigma|$ BDDs $T_a(\ell_i)$ ($a \in \Sigma$, $1 \leq i \leq n$) with logarithmic encoding over $\bar{y}$. So, $T_a(\ell_i)$ represents the set $\{\ell_j \mid (\ell_i, a, \ell_j) \in \Delta\}$. The observations $\Gamma = \{o_1, \dots, o_p\}$ are encoded by $\lceil \log_2 p \rceil$ boolean variables $b_0, b_1, \dots$ in the BDD B_Γ defined by

$$B_\Gamma \equiv \bigwedge_{0 \leq j \leq p-1} \bar{b} = [j]_2 \to C_{j+1}(\bar{y}),$$

where $[j]_2$ is the binary encoding of j and $C_1, \dots, C_p$ are BDDs that represent the sets $\gamma(o_1), \dots, \gamma(o_p)$ in logarithmic encoding.

Given the antichain $q = \{s_1, \dots, s_t\}$, let S_k ($1 \leq k \leq t$) be the BDDs that encode the set s_k in logarithmic encoding over $\bar{y}$. For each $a \in \Sigma$, we compute the BDD CP_a in linear encoding over $\bar{x}$ as follows:

$$\mathsf{CP}_a \equiv \forall \bar{b} \cdot \bigvee_{1 \leq k \leq t} \bigwedge_{1 \leq i \leq n} x_i \to \big[\forall \bar{y} \cdot \big(T_a(\ell_i) \wedge B_\Gamma\big) \to S_k\big].$$

Then, we define $\mathsf{CP} \equiv \bigvee_{a \in \Sigma} \mathsf{CP}_a(q)$, and we extract the maximal elements in $\mathsf{CP}(\bar{x})$ as follows, with ω a BDD that encodes the relation of (strict) set inclusion $\subset$:

$$\omega(\bar{x}, \bar{x}') \equiv \Big(\bigwedge_{i=1}^{n} x_i \to x_i'\Big) \wedge \Big(\bigvee_{i=1}^{n} x_i \neq x_i'\Big),$$

$$\mathsf{CP}^{\min}(\bar{x}) \equiv \mathsf{CP}(\bar{x}) \wedge \neg \exists \bar{x}' \cdot \omega(\bar{x}, \bar{x}') \wedge \mathsf{CP}(\bar{x}').$$

Finally, we construct the antichain $\mathsf{CPre}(q)$ as the following set of BDDs in logarithmic encoding: $\mathsf{CPre}(q) = \{s \mid \exists v \in \mathsf{CP}^{\min} : s = \{\ell_i \mid v(x_i) = \mathsf{true}\}\}$.

Features of the tool. The input of the tool is a file describing the transitions and observations of the game graph. The output is the set of maximal winning cells, and a winning strategy in compact representation. We have also implemented a simulator to let the user play against the strategy computed by the tool. The user has to provide an observation in each round (or may let the tool choose one randomly). The web page of the tool is `http://www.antichains.be/alpaga`. We provide the source code, the executable, an online demo, and several examples. Details of the tool features and usage are given in the appendix.

References

1. D. Berwanger, K. Chatterjee, L. Doyen, T. A. Henzinger, and S. Raje. Strategy construction for parity games with imperfect information. Technical Report MTC-REPORT-2008-005, http://infoscience.epfl.ch/record/125011, EPFL, 2008.

2. K. Chatterjee, L. Doyen, T. A. Henzinger, and J.-F. Raskin. Algorithms for omega-regular games of incomplete information. *Logical Methods in Computer Science*, 3(3:4), 2007.
3. K. Chatterjee and T. A. Henzinger. Assume-guarantee synthesis. In *Proc. of TACAS*, LNCS 4424, pages 261–275. Springer, 2007.
4. R. McNaughton. Infinite games played on finite graphs. *Annals of Pure and Applied Logic*, 65(2):149–184, 1993.
5. Fabio Somenzi. Cudd: Cu decision diagram package. http://vlsi.colorado.edu/~fabio/CUDD/.
6. W. Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theoretical Computer Science*, 200:135–183, 1998.

Details of Tool Features

4 Practical implementation

In this section we describe the implementation details of the tool Alpaga.

4.1 Programming Language

Alpaga is written in Python, except for the BDD package which is written in C. We use the CUDD BDD library [5], with its PYCUDD Python binding. There is some performance overhead in using Python, but we chose it for enhanced readability and to make the code easy to change. We believe this is important in the context of academic research, as we expect other researchers to experiment with the tool, tweak the existing algorithms and add their own.

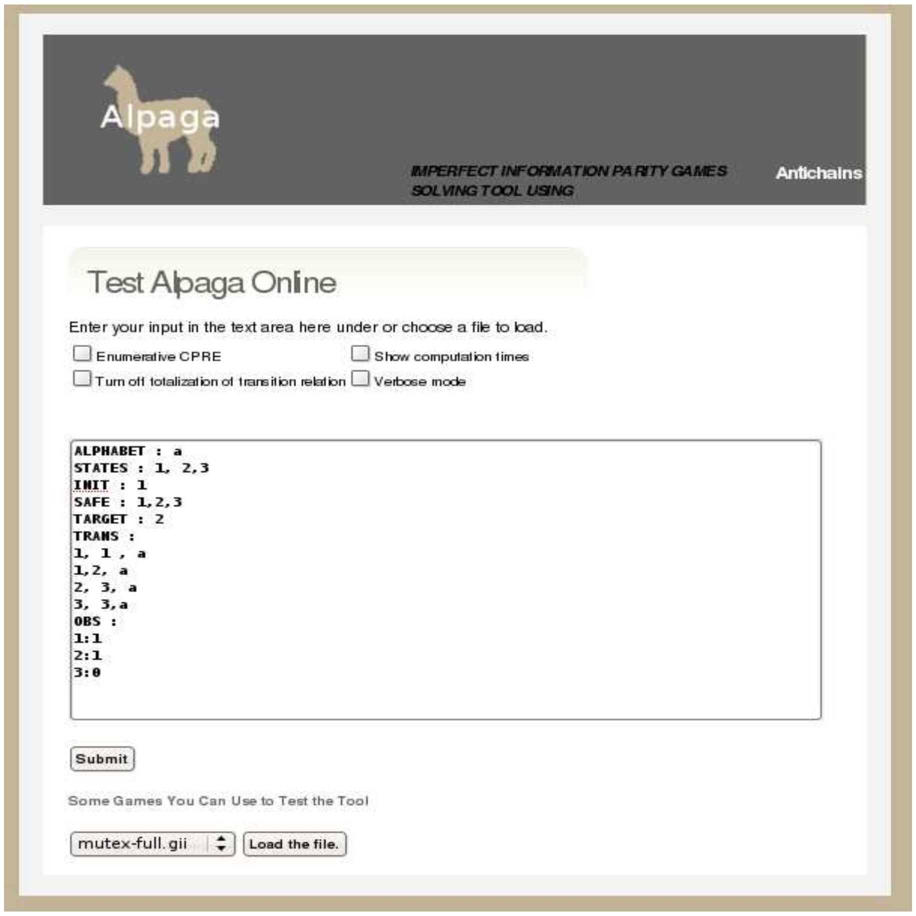
Alpaga is available for download at `http://www.antichains.be/alpaga` for Linux stations. For convenience, the tool can also be tested through a web interface (see Fig. 1 for a glimpse to this interface).

4.2 Code architecture

The code consists of four main classes:

1. `Game` is the main class of Alpaga. It encompasses all necessary information describing a game: BDDs for initial sets, target sets, observations, transition relations. The class offers two implementations of the controllable predecessors operator: (a) the"enumerative" CPre implementation which closely follows the definition of the CPre operator (enumerating labels, states and sets of the antichains, computing desired antichain intersections and unions as it progresses) and (b) the CPre implementation following the BDD technique explained in Section 3.

   Furthermore, the class offers a large set of utility functions to compute, for example, the successors of a set of states, its controllable predecessors, and to manipulate antichains of sets of states of the game. At a higher level, the class offers methods to compute strategies for specific kinds of objectives (ReachAndSafe: solving conjunction of reachability and safety objectives, and ReachOrSafe: solving disjunction of reachability and safety objectives). Finally it includes the implementation of the algorithm of [1] using all previous functions.
2. `Parser` produces an instance of the class `Game` from an input file. The parser also offers a good amount of consistency checking (it checks, for example, that every state belongs to one and only one observation).
3. `Strategy` is the class with data structure for strategy representation. The description of a strategy is based on the notion of *rank* (similar to rank of μ-calculus formulas), and a strategy maps a cell with a rank to a label and a cell with smaller rank.
4. `StrategyPlayer` is the class implementing the interactive mode of Alpaga. It takes as argument a game and a strategy and allows the user of Alpaga to replay the strategy interactively (see below).


Alpaga
IMPERFECT INFORMATION PARITY GAMES SOLVING TOOL USING
Antichains
Test Alpaga Online

Enter your input in the text area here under or choose a file to load.

☐ Enumerative CPRE ☐ Show computation times
☐ Turn off totalization of transition relation ☐ Verbose mode

```
ALPHABET : a
STATES : 1, 2,3
INIT : 1
SAFE : 1,2,3
TARGET : 2
TRANS :
1, 1 , a
1,2, a
2, 3, a
3, 3,a
OBS :
1:1
2:1
3:0
```

Submit
Some Games You Can Use to Test the Tool
mutex-full.gii
Load the file.

Fig. 1. Alpaga web interface.

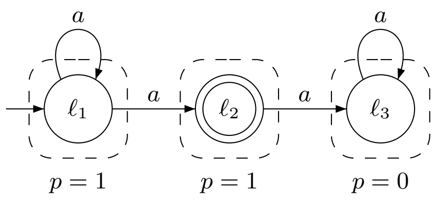
4.3 User Manual

In this section we describe the syntax of the input file, how to read the output, the various options of the tool, and finally we describe the interactive use of the tool.

Input. The syntax of the tool is straightforward and follows the formal description of imperfect information parity games as described in Section 2. Our algorithm solves games with objectives that are of the following form: parity objectives in conjunction with a safety objective, along with the disjunction with a reachability objective. The parity objective can be obtained as a special case when the safe set is the full set of states, and the target set for reachability objective is empty. In the description below, we have the safe and target set for the safety and reachability objectives, respectively.

We present the following example:

```
ALPHABET : a
STATES : 1, 2,3
INIT : 1
SAFE : 1,2,3
TARGET : 2
TRANS :
1, 1 , a
1,2, a
2, 3, a
3, 3,a
OBS :
1:1
2:1
3:0
```



The input file describing a parity game with imperfect information is constructed as follows:

– the sets of labels, states, initial states, safe states, and target states are all specified on a single line introduced by the corresponding keyword ALPHABET, STATES, INIT, SAFE or TARGET. The name of the states and labels can be any string accepted by Python that does not include a blank space or the # character (which is used for comments). However, the keyword SINK is reserved (see below).

– the transition relation is defined on a sequence of lines introduced by the keyword TRANS on a single line. After the TRANS keyword, each line specifies on a single line a transition, by giving the initial state, the destination state and the label of the transition, all separated by commas.

– Finally, the observations and corresponding priorities are specified in a similar fashion. They are introduced by the keyword OBS on a single line. Then follows the specification of the observations. Each observation is specified on a single line as a set of comma-separated states, followed by its priority (a positive integer number) which must be preceded by a colon.

– Blank lines are allowed anywhere as empty comments. Nonempty comments start with the character # and extend to the end of the line.

Output. The tool output for our example is in Fig. 2. The winning strategy computed by the tool is represented by a list of triples $(a, \mathsf{rank}, s) \in \Sigma \times \mathbb{N} \times 2^L$ where a is an action, and s is a cell. The strategy is represented in the compact form after applying Rule 1 and Rule 2 for simplification of strategies. The strategy representation can be used to find the action to play, given the current knowledge s' of Player 1 as follows: play the action a such that (a, rank, s) is a triple in the list with minimal rank such that $s' \subseteq s$ (such a triple must exist if s' is a winning cell).

Tool options. We now describe the various options with which the tool can be used.

```
alpaga.py [options] file
```

The possible options are the following:

- -h Shows an help message and exits.
- -i After computing a strategy, launches the interactive strategy player which allows to see how the strategy computed by the tools executes in the game. In this mode, the tool shows which move is played by the strategy, given the current knowledge (i.e., a set of states in which the player can be sure that the game is - the initial knowledge is the set of initial states). Then the tool allows to choose the next observations among the observations that are compatible with the current knowledge.
- -e Uses the enumerative CPre in all computations. There are two different implementations for the controllable predecessor operator (CPre), one temporarily using a linear encoding of the resulting antichain for the time of the computation, and an enumerative algorithm following closely the definition of the CPre operator.
- -n Turns off the totalization of the transition relation. By default, Alpaga completes the transition relation so that it becomes total, which means that a transition of every label exists from each state. Therefore, Alpaga first adds a state named SINK with priority 1 (corresponding to a new observation), from which every label loops back to SINK, and then adds a transition

  ```
  s, SINK, lab
  ```

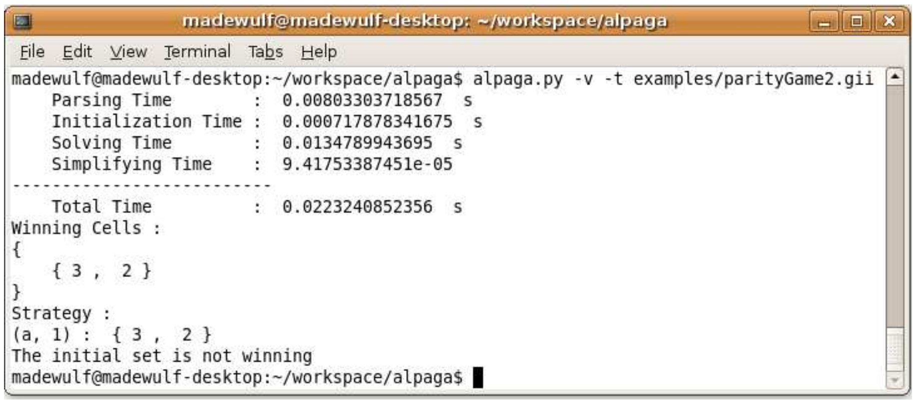
  for each pair (s, lab) such that there does not exist a transition from state s on label lab. Note that the name SINK is reserved.
- -r Turns on the display of stack traces in case of error.
- -s Turns off the simplification of the strategies before display.
- -t Displays computation times, which includes time for parsing the file (and constructing the initial BDDs), time for initializing the linear encoding, for computing a strategy, and for simplifying that strategy.
- -v Turns on the display of warnings, which mainly list the transitions added by the totalization procedure.

Interactive mode. After computing a strategy for a parity game, the tool can switch to interactive mode, where the user can “replay” the strategy, to check that the modelization was correct. The user of Alpaga plays the role of Player 2, choosing the observation among the compatible observations available, and getting the resulting knowledge of player 1 and which move she will play.

Practically, in interactive mode, type help for the list of commands: the standard way for playing a strategy is the following: launch alpaga with option -i, type go at the interactive prompt, type the number of an observation, type enter twice, repeat. Fig. 3 shows an interactive Alpaga session.

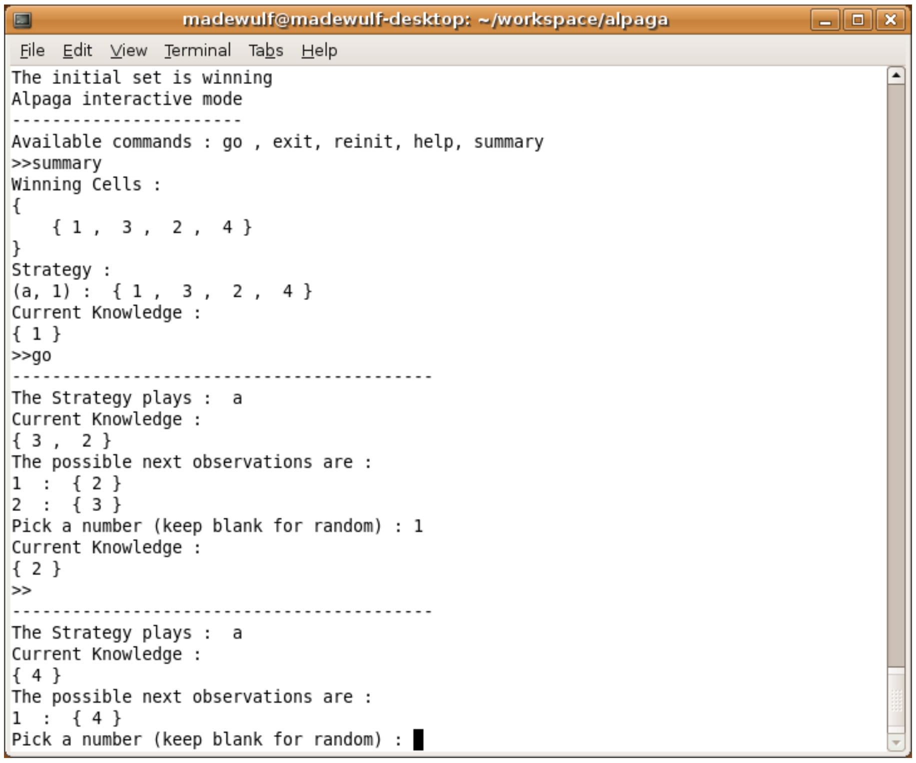
5 Example: mutual-exclusion protocol

We demonstrate the use of games with imperfect information to synthesize reactive programs in distributed systems. We consider the design of a mutual-exclusion protocol


madewulf@madewulf-desktop: ~/workspace/alpaga
File Edit View Terminal Tabs Help

```
madewulf@madewulf-desktop:~/workspace/alpaga$ alpaga.py -v -t examples/parityGame2.gii
    Parsing Time        :  0.00803303718567  s
    Initialization Time :  0.000717878341675  s
    Solving Time        :  0.0134789943695  s
    Simplifying Time    :  9.41753387451e-05
--------------------------
    Total Time          :  0.0223240852356  s
Winning Cells :
{
    { 3 ,  2 }
}
Strategy :
(a, 1) :  { 3 ,  2 }
The initial set is not winning
madewulf@madewulf-desktop:~/workspace/alpaga$
```

Fig. 2. Output of Alpaga.


madewulf@madewulf-desktop: ~/workspace/alpaga
File Edit View Terminal Tabs Help

```
The initial set is winning
Alpaga interactive mode
-----------------------
Available commands : go , exit, reinit, help, summary
>>summary
Winning Cells :
{
    { 1 ,  3 ,  2 ,  4 }
}
Strategy :
(a, 1) :  { 1 ,  3 ,  2 ,  4 }
Current Knowledge :
{ 1 }
>>go
------------------------------------------
The Strategy plays :  a
Current Knowledge :
{ 3 ,  2 }
The possible next observations are :
1  :  { 2 }
2  :  { 3 }
Pick a number (keep blank for random) : 1
Current Knowledge :
{ 2 }
>>
------------------------------------------
The Strategy plays :  a
Current Knowledge :
{ 4 }
The possible next observations are :
1  :  { 4 }
Pick a number (keep blank for random) :
```

Fig. 3. Interactive strategy player of Alpaga.

```
do {                                 do {
unbounded_wait;                      unbounded_wait;
flag[1]:=true;                       flag[2]:=true;
turn:=2;                             turn:=1;

| while(flag[1]) nop;           (C1) while(flag[1] & turn=1) nop;
| while(flag[2]) nop;           (C2)
| while(turn=1) nop;            (C3)
| while(turn=2) nop;            (C4)
| while(flag[1] & turn=2) nop;  (C5)
| while(flag[1] & turn=1) nop;  (C6)
| while(flag[2] & turn=1) nop;  (C7)
| while(flag[2] & turn=2) nop;  (C8)

fin_wait;  // Critical section       fin_wait;  // Critical section
flag[1]:=false;                      flag[2]:=false;
} while(true)                        } while(true)
```

Fig. 4. Mutual-exclusion protocol synthesis.

for two processes, following the lines of [3]. We assume that one process (on the right in Fig. 4) is completely specified. The second process (on the left in Fig. 4) has freedom of choice in line 4. It can use one of 8 possible conditions `C1`–`C8` to guard the entry to its critical section in line 5. The boolean variables `flag`$[1]$ and `flag`$[2]$ are used to place a request to enter the critical section. They are both visible to each process. The variable `turn` is visible and can be written by the two processes. Thus, all variables are visible to the left process, except the program counter of the right process.

There is also some nondeterminism in the length of the delays in lines 1 and 5 of the two processes. The processes are free to request or not the critical section and thus may wait for an arbitrary amount of time in line 1 (as indicated by `unbounded_wait`), but they have to leave the critical section within a finite amount of time (as indicated by `fin_wait`). In the game model, the length of the delay is chosen by the adversary.

Finally, each computation step is assigned to one of the two processes by a *scheduler*. We require that the scheduler is fair, i.e. it assigns computation steps to both processes infinitely often. In our game model, we encode all fair schedulers by allowing each process to execute an arbitrary finite number of steps, before releasing the turn to the other process. Again, the actual number of computation steps assigned to each process is chosen by the adversary.

The mutual exclusion requirement (that the processes are never simultaneously in their critical section) and the starvation freedom requirement (that whenever the left process requests to enter the critical section, then it will eventually enter it) can be encoded using three priorities.

When solving this game with our tool, we find that Player 1 is winning, and that choosing C_8 is a winning strategy.