AltaRica Examples

Winning the Nim game*

A. Griffault, G. Point
LaBRI - CNRS - Université Bordeaux

April 17, 2024

Contents

Nim is a game with two players. Matches are dispatched on rows. Each turn a player chooses a row and remove at least one match from the row. The game terminates when no more matches can be taken. Depending of winning rule, the player that takes last matches is either the winner of the loser.

In the sequel we show how to compute winning strategies. Such strategies ensure a player to win the game. Beyond the entertainment of playing (and winning) games, the computation of winning strategies plays important role in the synthesis of controllers that guarantee safety requirements of the system. These controllers apply a winning strategy against the environment of the system. In this particular games, the winning condition is not on matches but to respect critical requirements.

1 The model

The AltaRica model of this game is simple and can be easily generated using PHP preprocessor according to some number of lines N. For $N = 3$, the model is given on the listing below.

To simplify the model, all lines can contain up to $2 * N - 1$ matches. Each line declares for $k = 1, \ldots, 2 * N - 1$, an event `take_`k whose semantics is the removal of k matches from the line. These events decrement of k the state variable `n` that stores the remaining number of matches.

*This example is extracted from **A**ltaRica **H**andbook.

The model consists essentially in the declaration of the N lines. The top-level node `main` set the initial number of matches in each line. The first one contains 1 match, the second 3, the third 5 and so on using an increment of 2.

```
1  const  N = 3;
2  const  MaxMatches = 2 * N - 1;
3  domain Matches = [0,MaxMatches];
4
5  node Line
6    state n : Matches : public;
7    event take_1 : public;
8    trans
9      n >= 1 |- take_1 -> n := n - 1;
10   event take_2 : public;
11   trans
12     n >= 2 |- take_2 -> n := n - 2;
13   event take_3 : public;
14   trans
15     n >= 3 |- take_3 -> n := n - 3;
16   event take_4 : public;
17   trans
18     n >= 4 |- take_4 -> n := n - 4;
19   event take_5 : public;
20   trans
21     n >= 5 |- take_5 -> n := n - 5;
22 edon
23
24 node Nim
25   sub
26     L : Line[N];
27   init
28     L[0].n := 1, L[1].n := 3, L[2].n := 5;
29 edon
```

Listing 1: Model of the Nim game with 3 raws.

2 Validation

Even if we are confident in our description of the game, we request `arc` to validate now properties of the model.

- There is only one initial state.
- There is only one sink configuration. We specify this set of configuration in the same manner as *deadlocks* i.e., states that are node the origin of a transition.

  ```
  sink := any_s - src(any_t - self_epsilon);
  ```

- The model has $2^N \times N!$ configurations. It suffices to check if the cardinality of `any_s` corresponds.
- The length of the longest play is N^2. The board of the game contains N^2 matches and the longest play is the one when only one match is take at each turn. To get this length with `arc`, we compute the `trace` from the initial state to the sink configuration using only `take_1` transitions. For $N = 3$, the specification is:

```
take_one := label L[0].take_1
         or label L[1].take_1
         or label L[2].take_1;
maxpath := trace (initial, take_one, sink);
```

Results are those excepted. For instance, with $N = 6$ we obtain:

```
TEST(initial,1d)        [PASSED]
TEST(sink,1d)   [PASSED]
TEST(any_s,46080d)      [PASSED]
TEST(maxpath,36d)       [PASSED]
```

3 Winning strategies

In this section we use `arc` to determine if there exist winning strategies for both players of a Nim game. From now on, we assume the winner is the one that takes last matches; thus the loser is the one that has to play from a sink state. Applying the other rule is straightforward. Let call players A and B; we assume that A plays in first.

In a finite game with two players, a position i.e., a configuration of the model, is either *winning* or *losing* for the player that has to play from this position. Informally, a position is winning if there exists a move to a losing position (for the opponent). And, a position is losing if all possible moves lead into a winning position. A winning strategy proposes to a player facing a (winnin) position the set of possible moves that put the opponent in a *losing* configuration.

Definitions of winning and losing positions are mutually depend but, since we play in a finite arena, we can "easily" formalize these sets of configurations using fixpoint equations.

In the sequel we will use the following set of transitions that are actions of players on lines:

```
move := any_t - epsilon;
```

Let's start with trivially known winning and losing positions. The rule says that all `sink` states are losing position; we thus define the first set `Lose` that contains all trivially losing positions.

```
Lose := sink;
```

Then, all positions that precede `Lose` are obviously winning configurations; let call `Win` this new trivial set:

```
Win := any_s & src (move & rtgt (Lose));
```

These constant sets being defined we define following sets using least-fixpoint equations:

- `Winning` is the set of winning positions. A winning configuration is either in `Win` or is the origin of a winning move:

  ```
  Winning += Win or src (WinningMove);
  ```

- `WinningMove` is the set of transitions that are winning for the player that triggers them. This means that targets of such transitions are losing positions:

```
WinningMove += move & rtgt (Losing);
```

- `Losing` is the set of losing positions. A losing configuration is either a sink state in `Lose` set or all moves originating from these positions are losing moves:

```
Losing += Lose or (src(LosingMove) − src (WinningMove));
```

- Finally, `LosingMove` is the set of transitions that make the player lose the game by setting the opponent in a winning position:

```
LosingMove += move & rtgt (Winning);
```

Since `arc` does not support in `acheck` syntax the definition of fixpoints using several equations (it is however possible using Mec 5 syntax), we substitute definitions of `WinningMove`, `Losing` and `LosingMove` into `Winning`. We obtain the following `acheck` definitions:

```
Winning += Win |
           src (move & rtgt (Lose |
                             (src(move & rtgt (Winning)) −
                              src(move − rtgt (Winning)))));
LosingMove := move & rtgt (Winning);
Losing := Lose | (src (LosingMove) − src (move − LosingMove));
WinningMove := move & rtgt (Losing);
```

Now if we want to know if the first player will be the winner then we just have to check the emptyness of:

```
AWinner := initial & Winning;
```

Table **??** gathers computations made with `arc` for first values of N. When N equals 4 or 8, player A should loses the play if its opponent follows its winning strategy. These results just confirm what theory predicts. Actually it is known that a position is losing if the exclusive-or sum of remaining matches is 0. Table **??** gives this sum for each initial position i.e. the position from which A should start the play. For $N = 4$ and $N = 8$ this sum is 0.

Up to now we have show that first player has a strategy to win Nim game but we do not exhibit this strategy. `arc` permits to get it as a graph. We have to compute the subgraph of whole semantics that contains only winning plays. This subgraph is obtained from the semantics by removing moves of player A that do not belong to the strategy. Let define this set as:

```
BadMove := move & (rsrc (Winning) − WinningMove);
```

The set of configurations that belong to the strategy are accessible from the initial state without using wrong moves of A (any move of B should be losing):

```
plays := reach (initial, move−BadMove);
```

N	any_s	M	$\oplus$	AWinner
3	48	9	111	1
4	384	16	000	0
5	3840	25	1001	1
6	46080	36	0010	1
7	645120	49	1111	1
8	10321920	64	0000	0
9	185794560	81	10001	1

Table 1: Results computed with `arc` for 3 to 9 lines of matches. Column M gives the total number of matches. Column $\oplus$ gives the exclusive-or of number of matches in the lines. Column $AWinnner$ contains the cardinality of the property `AWinner`.

Now the expected subgraph is obtained using the following `dot` command. The result is displayed on figure **??** (page **??**).

Graphical description of the winning strategy makes sense only for small graphs. Another way to proceed is to request `arc` to generate a controller that embeds the strategy. Controllers is a basic feature of AltaRica language. Actually each intermediate node of the hierarchy is a controller that can constrain behaviours of its sub-nodes. Usually these constraints are simply assertions to wire flows or synchronization vector to enforce synchronism of events. These basic constraints can be coupled with complex behaviours described by transitions of the intermediate node.

In the current model of the game, the top-level node has no actual behaviours; the `main` node just declares its sub-nodes. We request `arc` to generate behaviours that *implements* the strategy for player A. To generate a usefull controller, this latter must be able to control and to take decisions according to current position of the game; this is why state variable and events of lines have been set `public`. This attribute makes state variables accessible to the parent node (but he could use a flow variable) and it creates implicitly a copy of events in the controller.

To generate the controller we use the following `arc` command (for $N = 3$). The two first parameters are behaviours allowed by the controller; the third parameter is the name of the generated node and the last one requests `arc` to simplify guard of transitions (using BDDs).

```
project (plays, move - BadMove, '$NODENAME3_R1', true)
```

The controller generated for $N = 3$ is given below. In produced node, `arc` has made synchronization of public events explicit.

```
/*
 * This node is the result of the projection of the node 'Nim'
 * on its subnode 'Nim'.
 */
node Nim3_R1
 event
```

```
  'L[0].take_1' : public;
  'L[0].take_2' : public;
  'L[0].take_3' : public;
  'L[0].take_4' : public;
  'L[0].take_5' : public;
  'L[1].take_1' : public;
  'L[1].take_2' : public;
  'L[1].take_3' : public;
  'L[1].take_4' : public;
  'L[1].take_5' : public;
  'L[2].take_1' : public;
  'L[2].take_2' : public;
  'L[2].take_3' : public;
  'L[2].take_4' : public;
  'L[2].take_5' : public;
 sub
  L : Line[3];
 sync
  <'L[2].take_5', L[2].take_5>;
  <'L[2].take_4', L[2].take_4>;
  <'L[2].take_3', L[2].take_3>;
  <'L[2].take_2', L[2].take_2>;
  <'L[2].take_1', L[2].take_1>;
  <'L[1].take_5', L[1].take_5>;
  <'L[1].take_4', L[1].take_4>;
  <'L[1].take_3', L[1].take_3>;
  <'L[1].take_2', L[1].take_2>;
  <'L[1].take_1', L[1].take_1>;
  <'L[0].take_5', L[0].take_5>;
  <'L[0].take_4', L[0].take_4>;
  <'L[0].take_3', L[0].take_3>;
  <'L[0].take_2', L[0].take_2>;
  <'L[0].take_1', L[0].take_1>;
 init
  L[0].n := 1,   L[1].n := 3,   L[2].n := 5;
/* Existential transitions */
 trans
  false |- 'L[2].take_5' -> ;
  false |- 'L[2].take_4' -> ;
  L[2].n=5 and L[1].n=3 and L[0].n=1 |- 'L[2].take_3' -> ;
  (L[2].n=2 and L[1].n=0 or L[2].n=2 and L[1].n=2) and L[0].n=0 or
      (L[2].n=2 and L[1].n=1 or L[2].n=2 and L[1].n=3) and L[0].n=1
      |- 'L[2].take_2' -> ;
  (L[2].n=1 and L[1].n=0 or 1<=L[2].n and L[2].n<=2 and L[1].n=1 or
      L[2].n=2 and L[1].n=2) and L[0].n=0 or (1<=L[2].n and
      L[2].n<=2 and L[1].n=0 or L[2].n=2 and L[1].n=3) and L[0].n=1
      |- 'L[2].take_1' -> ;
  false |- 'L[1].take_5' -> ;
  false |- 'L[1].take_4' -> ;
  1<=L[2].n and L[2].n<=2 and L[1].n=3 and L[0].n=1 |-
      'L[1].take_3' -> ;
  (L[2].n=0 or L[2].n=2) and L[1].n=2 and L[0].n=0 or (L[2].n=0 or
      L[2].n=2) and L[1].n=3 and L[0].n=1 |- 'L[1].take_2' -> ;
  (0<=L[2].n and L[2].n<=1 and L[1].n=1 or 1<=L[2].n and L[2].n<=2
      and L[1].n=2 or L[2].n=2 and L[1].n=3) and L[0].n=0 or
      (L[2].n=0 and L[1].n=1 or L[2].n=2 and L[1].n=3) and L[0].n=1
      |- 'L[1].take_1' -> ;
  false |- 'L[0].take_5' -> ;
  false |- 'L[0].take_4' -> ;
  false |- 'L[0].take_3' -> ;
  false |- 'L[0].take_2' -> ;
  (0<=L[2].n and L[2].n<=1 and L[1].n=0 or L[2].n=0 and L[1].n=1 or
      L[2].n=2 and 2<=L[1].n and L[1].n<=3) and L[0].n=1 |-
      'L[0].take_1' -> ;
edon
```

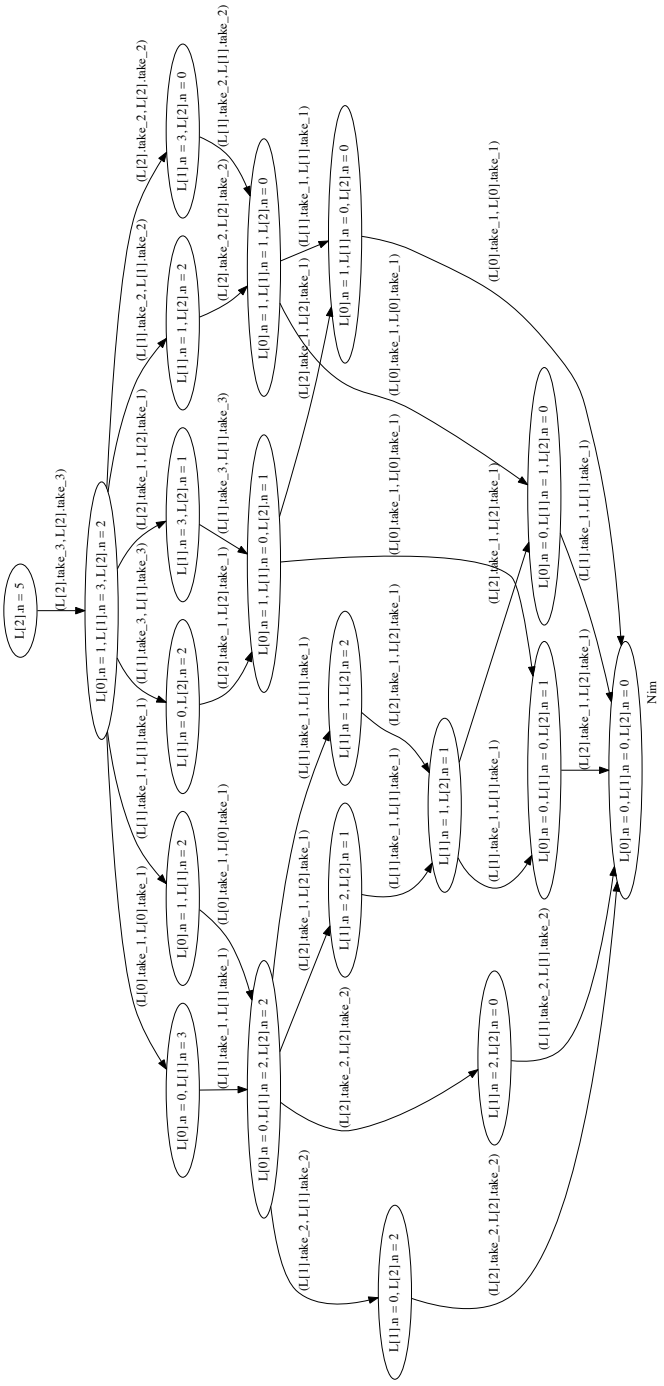

Figure 1: Winning strategy for player A for Nim game with 3 lines