System Validation

Shruti Chawane, Pietro Pennestrì

October 18, 2019

Contents

1 Part 1 - System modelling and analysis **1**
1.1 `NuSMV` model 1
1.2 `NuSMV` Model Validation 5
1.3 Winning strategies 6
1.4 State Space 9
1.5 Optimization of the `NuSMV` model 11
1.6 Gaps on the Board 11
1.6.1 Minimum holes to deadlock 12
1.6.2 Conditions to deadlock 12
1.6.3 Condition to ensure the victory 13
1.6.4 Many holes but no deadlock! 13
1.7 Long & Short matches 14

2 Part 2 - Software Analysis **15**
2.1 Runtime Monitor 15
2.2 Bounded Software Analysis 19
2.2.1 `GETDUPS.C` 19
2.2.2 `GETPRIMES.C` 22
2.3 Abstraction Refinement 24
2.3.1 Abstract model and predicates 24
2.3.2 CPAchecker vs CIVL 25

1 Part 1 - System modelling and analysis

1.1 `NuSMV` model

The following NuSMV module `game` models **parametrically** `Race to the Bottom`, according to the given game rules. The inputs of the module are the maximum abscissa and the maximum ordinate of the board[1].

The model considers *smart players*, this means that whenever a player is in position to win, then he will win the match. The figure 1 clarifies the meaning

[1] The minimum board size is 2×2 ($x_{\max} = 1,\ y_{\max} = 1$)

of the term *Smart Player*. The function `TRANS` generates randomly the token position, in accordance with the game rules. `FAIRNESS` conditions ensures that not only one move is used during the match. The Figure 2 depicts how the `NuSMV` model works.

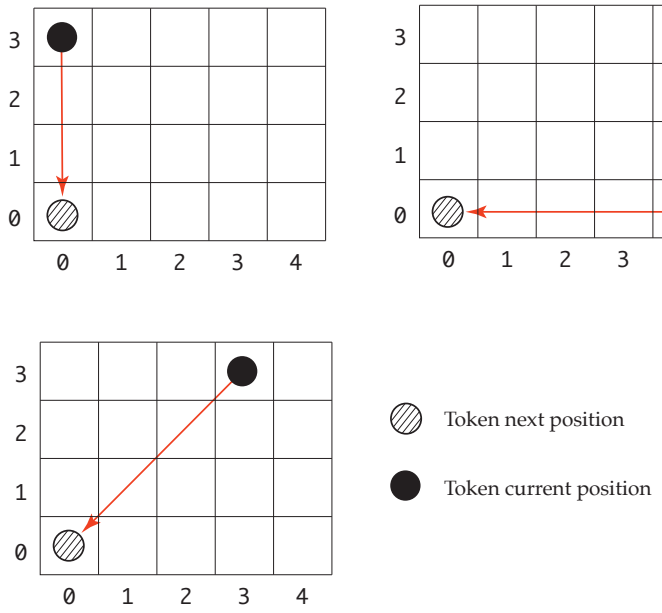

Figure 1: Smart Player

```
MODULE game(xmax, ymax)
    VAR
        x: 0..xmax;
        y: 0..ymax;

        move : {west,south,sw, none};
        player : {red, blue};
        state : {start,play,win};



    ASSIGN
        init(x):=xmax;
        init(y):=ymax;
        init(state):=start;
        init(move) := {west,south,sw};
```

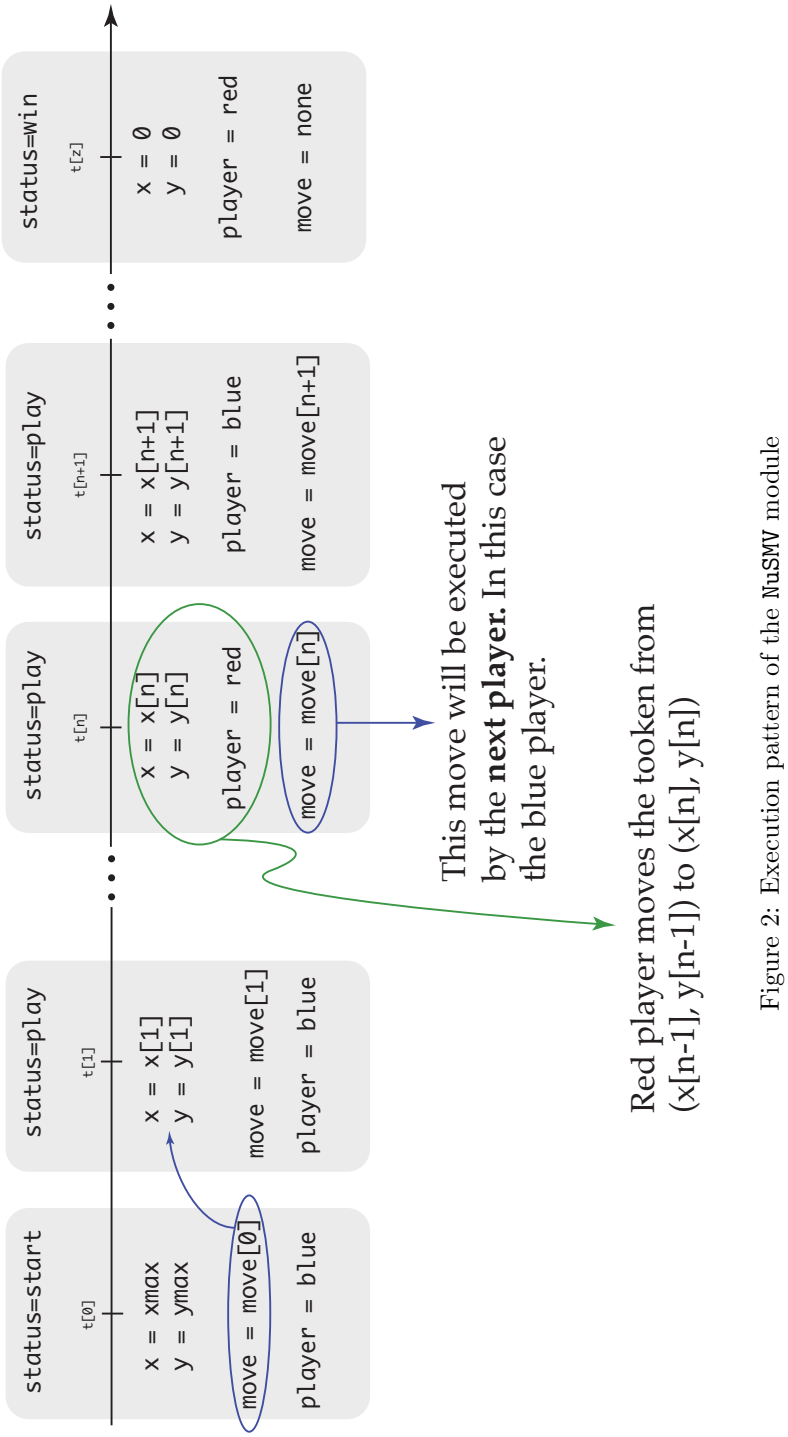

Figure 2: Execution pattern of the `NuSMV` module

```
    next(move):= case
        (next(x)=0 & next(y)=0): none;
        next(x)=0: south;
        next(y)=0: west;
        next(x)=next(y): sw;
        TRUE :{west, south, sw};
    esac;


    next(state):= case
        next(x)=0 & next(y)=0: win;
        state=start : play;
        state=play: play;
        state=win: start;
        TRUE: start;
    esac;


    next(player):= case
        state = start: player;
        state != start & player = blue: red;
        state != start & player = red: blue;
    esac;


TRANS
    ((move = west & y!=0 ) | (move = sw & x < y) ) -> next(x)
        < x;

TRANS
    ( (move = west & y = 0) | (move = sw & x = y) ) -> next(x)
         = 0;

TRANS
    move = south -> next(x) = x;

TRANS
    move = sw & x > y -> next(x) = x - y + next(y);

TRANS
    move = none -> next(x)=xmax;


TRANS
    ((move = south & x!=0 ) | (move = sw & y < x)) -> next(y)
```

```
            < y ;

    TRANS
        ( (move = south & x = 0) | (move = sw & x = y) ) -> next(y
            ) = 0;

    TRANS
        move = west -> next(y) = y;

    TRANS
        move = sw & y > x -> next(y) = y - x + next(x);

    TRANS
        move = none -> next(y) = ymax;


    FAIRNESS !(move=west);
    FAIRNESS !(move=sw);
    FAIRNESS !(move=south);

MODULE main()
        VAR
                match: game(2,5);
```

1.2 NuSMV Model Validation

In order to verify the consistency of the NuSMV model with the game rules, the following LTL specifications were checked and proved true:

1. South movement check
    - `G(match.move = south <-> next(match.x) = match.x)`
      `[Safety]` This property ensures globally that the abscissa of the token is **not modified** in the next move **if and only if** the player performs a south move.
    - `G((match.move = south | match.move = sw) <-> next(match.y) < match.y)`
      `[Safety]` This property ensures globally that the ordinate of the token is **decreased** in the next move **if and only if** the player performs a south or a south - west move.
    - `G((match.move = south) -> next(match.y) < match.y)`
      `[Safety]` This property ensures globally that if the player performs a south move **then** the ordinate of the token is **decreased** in the next move.
2. West movement check

- `G(match.move = west <-> next(match.y) = match.y)`
  `[Safety]` This property ensures globally that the ordinate of the token is **not modified** in the next move **if and only if** the player performs a west move.
- `G((match.move = west | match.move = sw) <-> next(match.x) < match.x)`
  `[Safety]` This property ensures globally that the abscissa of the token is **decreased** in the next move **if and only if** the player performs a west or a south - west move.
- `G((match.move = west) -> next(match.x) < match.x)`
  `[Safety]` This property ensures globally that if the player performs a south move **then** the abscissa of the token is **decreased** in the next move.

3. South - West movement check
   - `G(match.move = sw <-> match.x - next(match.x) = match.y - next(match.y));`
     `[Safety]` This property ensures globally that both abscissa and ordinate of the token are **varying** of the same quantity in the next move **if and only if** the player performs a south - west move.
   - `G(match.move = sw <-> match.x > next(match.x) & match.y > next(match.y));`
     `[Safety]` This property ensures globally that both abscissa and ordinate of the token are **decreasing** in the next move **if and only if** the player performs a south - west move.
4. A player should not play two times in a row.

```
G(match.state=play -> match.player!= next(match.player))
```

According to the game rules, every match must have a winner. This `Liveness` property was checked with the following `LTL Property`:

```
G(F match.state= win)
```

this ensures that in some future state there will be a winner. This last property, together with the `NuSMV` command `check_fsm`, ensures that the rules and the model for the game are **deadlock-free**.

1.3 Winning strategies

Winning Strategy 1. *If the token is in positions where $x = 0$ or $y = 0$ or $x = y$ then the player, who moves the token from positions which respect one of the previously conditions, will always win the game.*

Proof. This statement is proven with the following CTL property.

```
AG(((match.x=match.y) | (match.x=0) | (match.y=0)) & (match.x!=0
          | match.y!=0) -> EX(match.x=0 & match.y=0) )
```

□

Winning Strategy 2. *If the token is in* $(1, 2)$ *or in* $(2, 1)$*, then there is no chance to win the game for the player who moves the token from this position.*

Proof. This statement can be proved with the following CTL property:

```
AG( (match.x=1 & match.y=2) | (match.x=2 & match.y=1) ->
     AX(match.x=0 | match.y=0 | match.x=match.y) )
```

It is always true that whenever the token is in $(1, 2)$ or in $(2, 1)$ then one of the following properties will be always true in the **next position** of the token:

- $x = 0$
- $y = 0$
- $x = y$.

As shown by the previously winning strategy from these positions our opponent will always win. □

Winning Strategy 3. *Except from positions* $(1, 2)$ *or* $(2, 1)$*, during the* `match.status = play`*, never leave the token where the following properties holds:*

$$|x - y| = 1 \tag{1}$$

Proof. The statement can be proven with the following CTL formula:

```
    ((match.x - match.y = 1) | (match.y - match.x = 1) ) &
   ((match.x!=1 & match.x!=2) | (match.x!=2 & match.x!=1)) &
(match.x>0 & match.y>0) & match.state!=start -> EX((match.x=1 &
            match.y=2) | (match.x=2 & match.y=1))
```

This formula shows that from points which respects condition 1, our opponent can lead us to $(1, 2)$ or $(2, 1)$. □

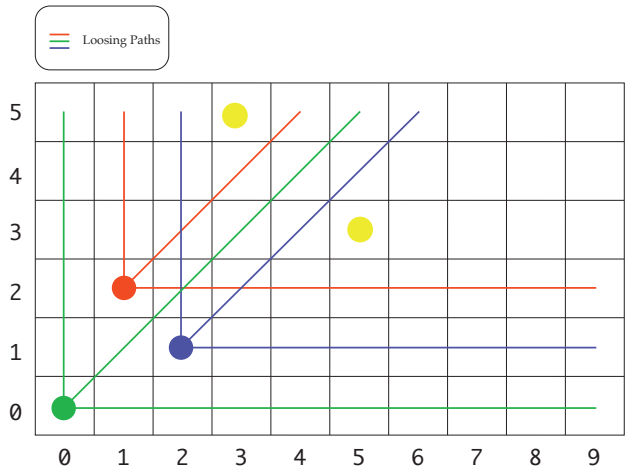

Figure 3: Winning Strategy

Winning Strategy 4. *As shown from Figure 3 if the player places the token in* $(3,5)$ *or in* $(5,3)$*, he will always win.*

Proof. We will only prove this winning strategy for the case of $(3,5)$. The proof for $(5,3)$ is similar.

The statement can be proven with the following `CTL` formula:

```
(match.x=3 & match.y= 5) -> AX( (match.x=2 & match.y > 1) |
(match.x=1 & match.y > 2 ) | (match.x=0) | (match.x - match.y =
1) | (match.y -match.x =1) | (match.x=match.y) | (match.y =2 &
match.x>1) | (match.y =1 & match.x > 2) | (match.y =0) )
```

If the token is in $(5,3)$ at time $t = k$, **all possible next locations** at time $t = k+1$ respect one of the following property:

- `match.x=2 & match.y > 1`. If this condition holds, at time $t = k+2$ the player can place the token in $(2,1)$.
- `match.x=1 & match.y > 2`. If this condition holds, at time $t = k+2$ the player can place the token in $(1,2)$.
- `match.x=0`. If this condition holds, at time $t = k+2$ the player can place the token in $(0,0)$.
- `match.x - match.y = 1`. If this condition holds, at time $t = k+2$ the player can place the token in $(2,1)$.
- `match.y -match.x =1`. If this condition holds, at time $t = k+2$, the player can place the token in $(1,2)$.

- `match.x=match.y`. If this condition holds, at time $t = k + 2$ the player can place the token in $(0, 0)$.
- `match.y =2 & match.x>1`. If this condition holds, at time $t = k+2$, the player can place the token in $(2, 1)$.
- `match.y =1 & match.x > 2`. If this condition holds, at time $t = k + 2$, the player can place the token in $(1, 2)$.
- `match.y =0`. If this condition holds, at time $t = k + 2$, the player can place the token in $(0, 0)$.

□

1.4 State Space

The total number of states can be computed with the following formula:

$$T(x_{\max}, y_{\max}) = n_{\text{moves}} n_{\text{players}} n_{\text{states}} (x_{\max} + 1)(y_{\max} + 1) \tag{2}$$

where:

- $n_{\text{moves}} = 4$ number of possible moves;
- $n_{\text{players}} = 2$ numper of possible players;
- $n_{\text{states}} = 3$ number of possible states.

In the following the formula $R(x_{max}, y_{max})$ to compute the number of reachable states will be deduced.
In Figure 4 considers the case where $x_{\max} \leq y_{\max}$:

- The number p_0 of reachable states where `match.status=play` and 3 movements { `south, west, sw` } are allowed is:

$$p_0 = ((x_{\max} + 1)(y_{\max} + 1) - y_{\max} - x_{\max} - 1 - 1 - x_{\max})\, n_{\text{players}} (n_{\text{moves}} - 1) \tag{3}$$

- The number p_1 of reachable states where `match.status=play` and only one movement is allowed is:

$$p_1 = (x_{\max} + x_{\max} + y_{\max})\, n_{\text{players}} \tag{4}$$

- In the $(x_{\max}, y_{\max})$ position the total numbers of reachable states is $n_{\text{players}}(n_{\text{moves}} - 1)$.
- In the $(0, 0)$ position the total number of reachable positions is n_{players}

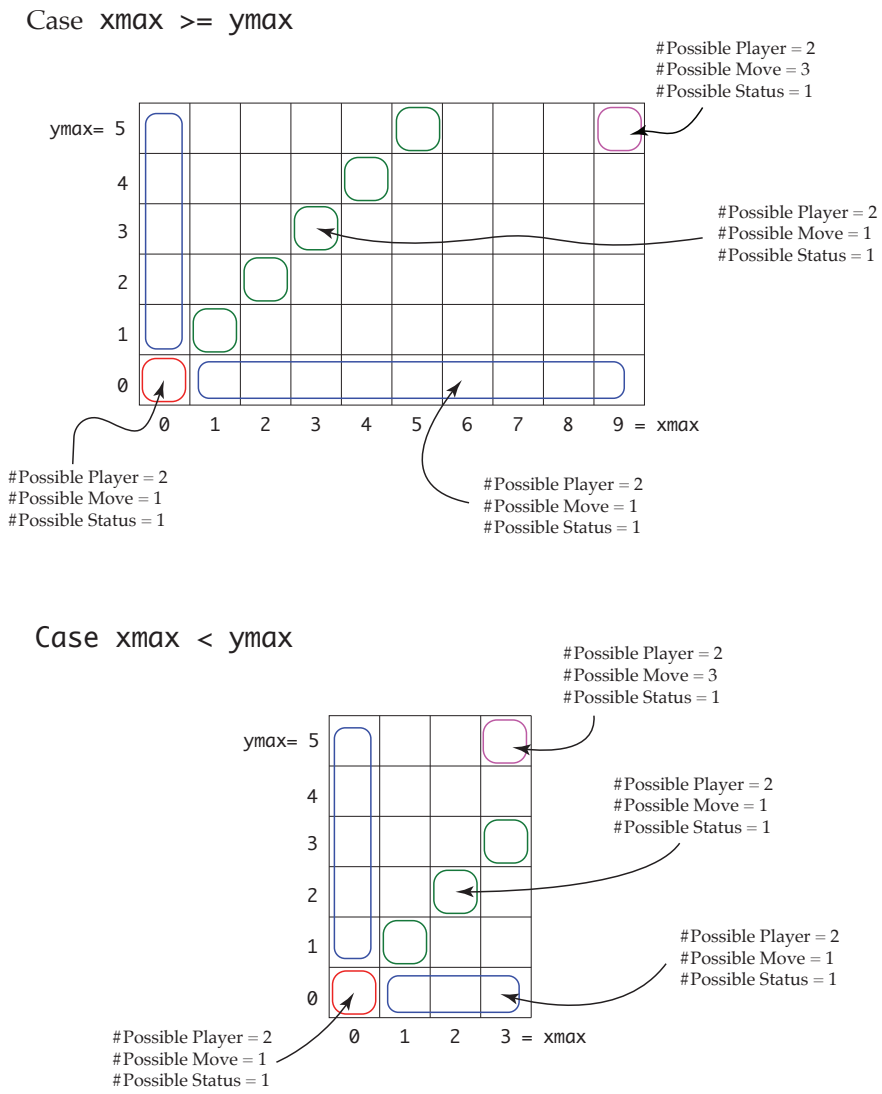

Figure 4: Count of reachable states

The total number of reachable states when $x_{\max} \leq y_{\max}$ is given by the following formula:

$$\begin{aligned} &R_{x_{\max} \leq y_{\max}}(x_{\max}, y_{\max}) = n_{\text{players}}(n_{\text{moves}} - 1) + n_{\text{players}} \\ &+ ((x_{\max}+1)(y_{\max}+1) - y_{\max} - x_{\max} - 1 - 1 - x_{\max})\, n_{\text{players}}(n_{\text{moves}} - 1) \\ &+ (x_{\max} + x_{\max} + y_{\max})\, n_{\text{players}} \ . \end{aligned} \tag{5}$$

Similarly with the case $R_{x_{\max} \leq y_{\max}}(x_{\max}, y_{\max})$, we can deduce $R_{x_{\max} > y_{\max}}(x_{\max}, y_{\max})$:

$$\begin{aligned} &R_{x_{\max} > y_{\max}}(x_{\max}, y_{\max}) = n_{\text{players}}(n_{\text{moves}} - 1) + n_{\text{players}} \\ &+ ((x_{\max}+1)(y_{\max}+1) - y_{\max} - x_{\max} - 1 - 1 - y_{\max})\, n_{\text{players}}(n_{\text{moves}} - 1) \\ &+ (x_{\max} + y_{\max} + y_{\max})\, n_{\text{players}} \ . \end{aligned} \tag{6}$$

By substituting in equations (5) and (6) the value of n_{players}, n_{states} and n_{moves}, we obtain the expression for $R(x_{max}, y_{max})$:

$$\boxed{R(x_{max}, y_{max}) = 8 + ((x_{\max}+1)(y_{\max}+1) - x_{\max} - y_{\max} - 2)6 + 2\,|x_{\max} - y_{\max}|} \tag{7}$$

The equation (7) has been implemented with the following `python` script:

```
def rstates(xmax,ymax):
       if (xmax<=ymax):
              return 8 + ((xmax+1)*(ymax+1) - xmax - ymax -1 -1 -
                  xmax)*6
               + (xmax+xmax+ymax)*2
       else:
              return 8 + ((xmax+1)*(ymax+1) - xmax - ymax -1 -1 -
                  ymax)*6
              + (ymax+xmax+ymax)*2
```

1.5 Optimization of the NuSMV model

From equation (2) it is clear that the total number of states is dependent only on the size of the board, n_{players}, n_{states} and n_{moves}. Unfortunately this parameter cannot be changed without changing the rules of the game.

1.6 Gaps on the Board

In this section a board of 10×6 will be considered.

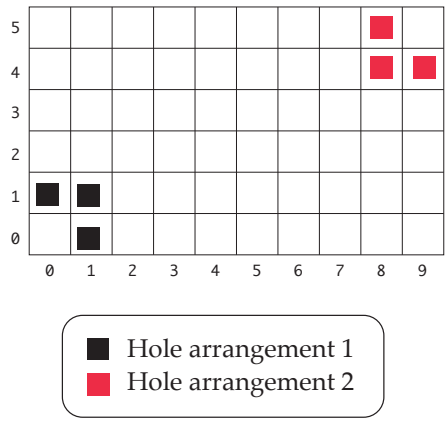

Figure 5: Minimum numbers of holes to deadlock

1.6.1 Minimum holes to deadlock

The minimum number of holes to deadlock the game is 3. These holes should be placed as shown in Figure 5.

To prove the deadlock for hole arrangement 1, the following `CTL` property is used:

```
AX(match.x>1 & match.y>1) -> !EX(match.state=win)
```

1.6.2 Conditions to deadlock

Given the set $H := \{(x^H, y^H)\}$ of holes, one of the following conditions should hold to deadlock the game:

$$\begin{aligned} &\forall x \in \left[0, x_M^H\right] : \exists \left(x, y_M^H\right) \in H \\ &\forall y \in \left[0, y_M^H\right] : \exists \left(x_M^H, y\right) \in H \end{aligned} \tag{8}$$

$$\begin{aligned} &\forall x \in \left[x_m^H, x_M^H\right] : \exists \left(x, y_m^H\right) \in H \\ &\forall y \in \left[y_m^H, y_M^H\right] : \exists \left(x_m^H, y\right) \in H \end{aligned} \tag{9}$$

where, x_m^H is the minimum among the abscissa of the holes, x_M^H is the maximum among the abscissa of the holes, y_m^H is the minimum among the ordinate of the holes and x_M^H is the maximum among the ordinate of the holes.

1.6.3 Condition to ensure the victory

If we want to ensure the victory of the first player, there is no need to place any hole. As discussed on the winning strategy section, on its first move the player places the token in $(5, 3)$. This will ensure him the victory. If we want to ensure the victory of the second player holes should placed as shown in Figure 6:

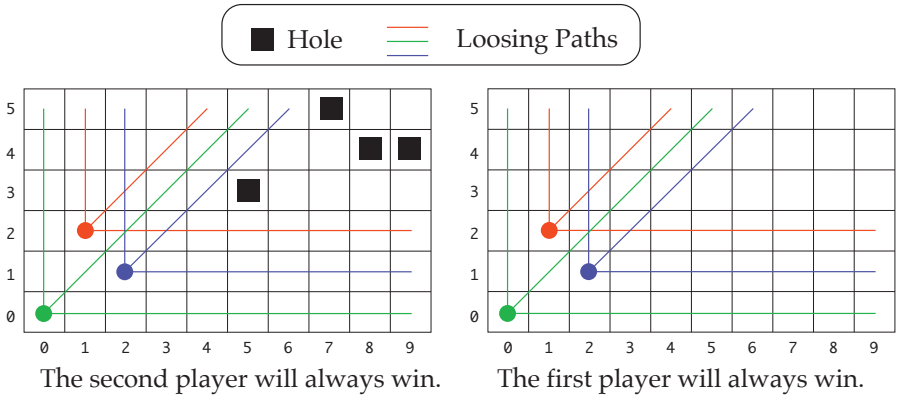

Figure 6: Conditions to ensure the victory

1.6.4 Many holes but no deadlock!

The Figure 7 depicts how to place as many holes as possible without having deadlock.

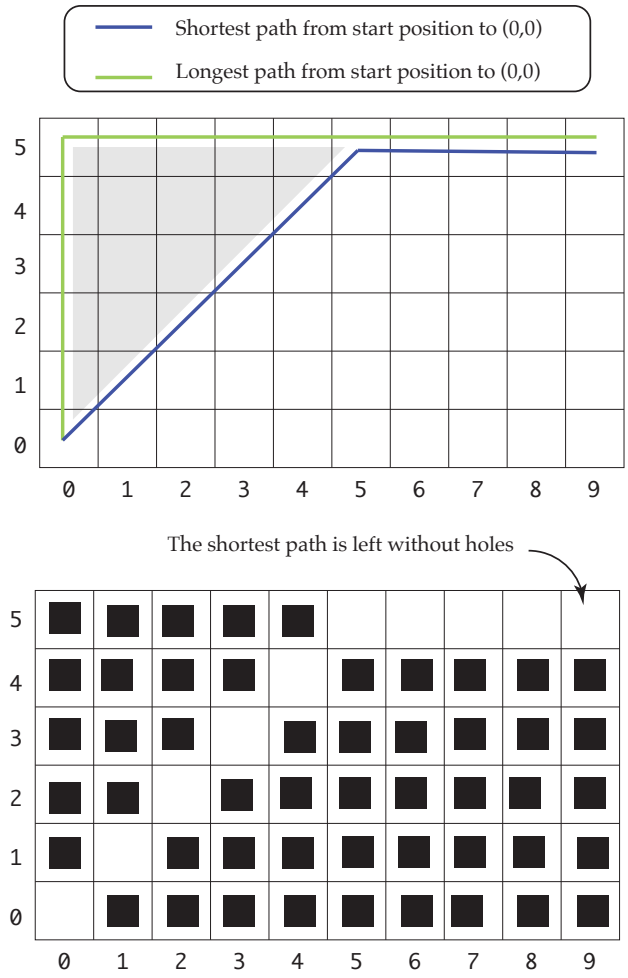

Figure 7: How to place the maximum number of holes without having deadlock

1.7 Long & Short matches

As shown in the victory strategy section, the shortest match is with a square board. In this case the first player wins in only one move. The longest possible games happen when each player moves one step each time. In this case there are two possible trajectories for the token, as shown in Figure 8. The moves needed for the longest possible games are $x_{\max} + y_{\max}$.

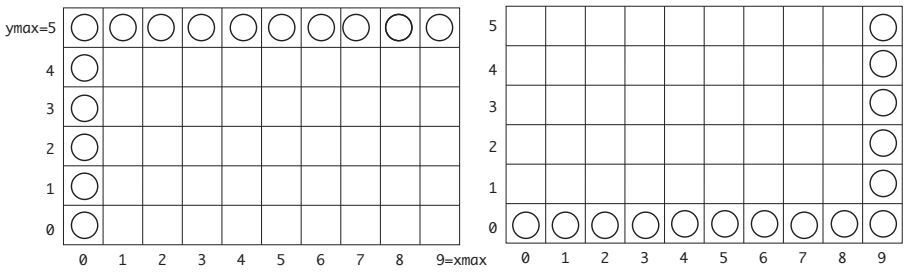

Figure 8: Possible trajectories for the longest match

2 Part 2 - Software Analysis

2.1 Runtime Monitor

The following run time monitor has been constructed to capture the behavior of the game.

```
IMPORTS {
import main.Board;
}

GLOBAL{

VARIABLES{

int numPlayers = 0;

int ox;
int oy;

int xmax;
int ymax;

}

FOREACH (Board b) {

EVENTS{

createPlayer() = {Board b1.createPlayer()} where {b = b1;}
doMove() = {Board b1.doMove()} where {b = b1;}

}
```

```
PROPERTY k{

STATES{
ACCEPTING{win {System.out.println("WIN!");} }
BAD{badmove {System.out.println("Bad Move!");} }
NORMAL{play {System.out.println("Play");} }
STARTING{starting {System.out.println("Start");}}
}

TRANSITIONS{
starting -> play [doMove \\ ox = b.getTokenX(); oy = b.getTokenY
    (); System.out.println("Game start at: (" +ox + "," + oy + ")
    " );]

play -> play [doMove \ ( ( (ox - b.getTokenX() == oy - b.
    getTokenY()) && ( ox > b.getTokenX() && oy > b.getTokenY() )
    ) || (oy > b.getTokenY() && ox==b.getTokenX()) || (ox > b.
    getTokenX() && oy==b.getTokenY() ) ) && (b.getTokenY() != 0
    || b.getTokenX() != 0) \ System.out.println( "(" +ox + "," +
    oy + ") --> (" + b.getTokenX() + "," + b.getTokenY() + ")" );
     ox = b.getTokenX(); oy = b.getTokenY();]


play -> badmove [doMove \ !( ( (ox - b.getTokenX() == oy - b.
    getTokenY()) && ( ox > b.getTokenX() && oy > b.getTokenY() )
    ) || (oy > b.getTokenY() && ox==b.getTokenX()) || (ox > b.
    getTokenX() && oy==b.getTokenY() ) ) \ System.out.println(
    "(" +ox + "," + oy + ") --> (" + b.getTokenX() + "," + b.
    getTokenY() + ")" ); ]


play -> win [doMove \ ( ( (ox - b.getTokenX() == oy - b.getTokenY
    ()) && ( ox > b.getTokenX() && oy > b.getTokenY() ) ) || (oy
    > b.getTokenY() && ox==b.getTokenX()) || (ox > b.getTokenX()
    && oy==b.getTokenY() ) ) && (b.getTokenY() == 0 && b.
    getTokenX() == 0) \ System.out.println("ok");]



}

}

}

}
```

The LARVA runtime monitor highlighted the following bug in the code:

> while playing a match between a human and a computer, the human player could pass his turn without moving the token.

This event produced the following trace:

```
Start
[k]AUTOMATON::> k(main.Board@475b7792 ) STATE::>starting
Game start at: (4,5)
Play
[k]MOVED ON METHODCALL: void main.Board.doMove(int, int) TO STATE
    ::> play
[k]AUTOMATON::> k(main.Board@475b7792 ) STATE::>play
(4,5) --> (4,5)
Bad Move!
[k]MOVED ON METHODCALL: void main.Board.doMove(int, int) TO STATE
    ::> !!!SYSTEM REACHED BAD STATE!!! badmove
aspects._asp_ToTheBottom1.
    ajc$before$aspects__asp_ToTheBottom1$3$d4d6ddd3(
    _asp_ToTheBottom1.aj:38)
main.Board.main(Board.java:77)
[k]AUTOMATON::> k(main.Board@475b7792 ) STATE::>badmove
[k]AUTOMATON::> k(main.Board@475b7792 ) STATE::>badmove
[k]AUTOMATON::> k(main.Board@475b7792 ) STATE::>badmove
[k]AUTOMATON::> k(main.Board@475b7792 ) STATE::>badmove
[k]AUTOMATON::> k(main.Board@475b7792 ) STATE::>badmove
[k]AUTOMATON::> k(main.Board@475b7792 ) STATE::>badmove
[k]AUTOMATON::> k(main.Board@475b7792 ) STATE::>badmove
[k]AUTOMATON::> k(main.Board@475b7792 ) STATE::>badmove
```

The following is a copy of the played match:

```
       running the application...
       Enter the width of the board:
       5
       Enter the height of the board:
       6
       Is the first player a human player (true) or s computer
           player (false)?
       true
       What is the name of the first player?
       pietro
       Is the second player a human player (true) or a computer
           player (false)?
       false
```

```
What is the name of the second player?
mypc
The current position is (4, 5)
pietro may play!
Would you like to move south(0), southwest(1) or west(2)?
0
How many steps would you like to move?
0
The current position is (4, 5)
mypc may play!
The current position is (1, 5)
pietro may play!
Would you like to move south(0), southwest(1) or west(2)?
0
How many steps would you like to move?
0
The current position is (1, 5)
mypc may play!
The current position is (1, 4)
pietro may play!
Would you like to move south(0), southwest(1) or west(2)?
0
How many steps would you like to move?
0
The current position is (1, 4)
mypc may play!
The current position is (1, 1)
pietro may play!
Would you like to move south(0), southwest(1) or west(2)?
0
How many steps would you like to move?
0
The current position is (1, 1)
mypc may play!
The current position is (1, 0)
pietro may play!
Would you like to move south(0), southwest(1) or west(2)?
0
How many steps would you like to move?
0
The current position is (1, 0)
mypc may play!
GAMEOVER!
mypc has won!
```

The given code has also another bug:

If the user desires to play with a board of $D_x \times D_y$, the script will consider a board of $(D_x + 1) \times (D_y + 1)$.

This bug was detected while constructing the `LARVA` runtime monitor. The bug was fixed by changing the following lines of code in `Board.java` (around line 20):

```
tokenX = x-1;
tokenY = y-1;
```

2.2 Bounded Software Analysis

2.2.1 `GETDUPS.C`

The code provided has the following bugs inside the `getDup()` function:

- The size of the `uniques[]` array is larger then needed.
- The `uniques[]` array is not initialized.

The program `getDups.c` was fixed and modified to work with `CIVL`. In the following listing of the new code version is attached.

```
#include <civlc.cvh>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#include <string.h>
#include <time.h>


void printArray(int array[], int arraySize){
       printf("[");
       for(int loop = 0; loop < arraySize; loop++){
     printf("%d␣", array[loop]);
       }
       printf("]␣\n");
}

/*
* Looks for $elem$ in the given array.
* Returns the index where this element can be found or -1 if it
    cannot be found
*/
int contains(int arr[], int bound, int elem) {
       for (int i = 0; i < bound; i++) {
              if (arr[i] == elem) {
```

```
                        return i;
                }
        }
        return -1;
}

/*
* Checks whether an array contains any duplicates
* It returns the duplicate or -1 if there is no duplicate
*
* The uniques array had an incorrect size.
* The uniques array was not initialized properly!
*
*/
int getDup(int arr[], int size) {
        int uniques[size];
        for (int j = 0; j < size; j++) {
                uniques[j]=-1;
        }


    for (int j = 0; j < size; j++) {
                int loc = contains(uniques, size, arr[j]);
        if (loc == -1) {
                uniques[j] = arr[j];
        } else {
                return arr[j];
        }
    }
        return -1;
}

$input int arraySize = 15;


int main() {
        /* int a[] = {5, 9, 1, 8, 2, 7, 3, 6, 0, 21321};
        int dup = getDup(a, 7);
        printf("%d", dup);
        */


    /*
    Populate array with random variables. Array a should not have
        duplicates!
    Duplicates in array a will be introduced later.
```

```
    */

    // Initialize array a
    int a[arraySize];
    for (int i=0; i<arraySize; i++){
          a[i] = i+1;
    }


    /*
        Debug the function getDup()
    During this debug the array a will be populate with a couple
        of
    duplicates a time. For each control there is only one
        duplicate pair inside
    the array.
    */

    for (int i = 0; i < arraySize; i++)
    {
        for (int j = 0; j < arraySize; j++)
        {
               if(j != i){
                      int oldValue = a[j];
                      a[j]=a[i];
                      $assert( getDup(a, arraySize) == a[i]);
                      a[j]= oldValue;
               }
        }
    }


}
```

The following output was collected from `CIVL`:

```
       === Stats ===
       time (s) : 48.76
       memory (bytes) : 157515776
       max process count : 1
       states : 124128
       states saved : 123187
       state matches : 0
       transitions : 124127
       trace steps : 113353
       valid calls : 723880
```

```
provers : cvc4, z3
prover calls : 0

=== Result ===
The standard properties hold for all executions.
```

2.2.2 GETPRIMES.C

We assume that the algorithm implemented for `isPrime()` function is correct.
The following modifications on `C` code were implemented:

- `[OLD] i<=limit ==> [NEW] i<limit` This out of index referencing was highlighted by CIVL.
- The `printPrimes()` function was a `void` type, we turned it to `int`, because to validate the code we needed that the function returned the array with primes numbers.

The program `primes.c` was fixed and modified to work with `CIVL`. In the following a listing of the new version of the code is attached.

```
#include <civlc.cvh>
#include <stdio.h>
#include <stdlib.h>


$input int limit = 100;

int isPrime(int num)
{
    if (num <= 1) return 0;
    if (num % 2 == 0 && num > 2) return 0;
    for(int i = 3; i < num / 2; i+= 2)
    {
        if (num % i == 0)
            return 0;
    }
    return 1;
}


// print all primes number < limit
int printPrimes(int *primes, int z) {
   unsigned long long int i,j;

   for (i=2;i<limit;i++)
      primes[i]=1;
```

```
    printf("primes[0]=%d␣\n",primes[0]);
    printf("primes[1]=%d␣\n",primes[1]);

    for (i=2; i<limit;i++)
        if (primes[i])
            for (j=i;i*j<limit;j++)
                primes[i*j]=0;
}

int main(){

    /*
        We assume that the algorithm implemented for isPrime()
            function
        is correct.

        NOTE: the following modification on c code were done:

        1) [OLD] i<=limit --------> [NEW] i<limit

        This out of index referencing was highlighted by CIVL.

        2) The printPrimes() function was void, we turned it to
            int,
            because to validate the code we needed that the
                function
            returned the array primes.

    */

    unsigned long long int i;

    int *primes;
    primes = malloc(sizeof(int)*limit);
    int z = 1;

    printPrimes(primes, z);

    int error=0;
    for (i=2;i<limit;i++)
        if (primes[i]){
            printf("%dth␣prime␣=␣%lld\n␣",z++,i);
            if(isPrime(i)==0){
                error=error+1;
            }
```

```
        }


    $assert(error==0);
    printf("Error␣c␣:␣%d␣␣\n␣",error);
    free(primes);
}
```

2.3 Abstraction Refinement

2.3.1 Abstract model and predicates

The Figure 9 shows the abstract model for the program `predabs.c`

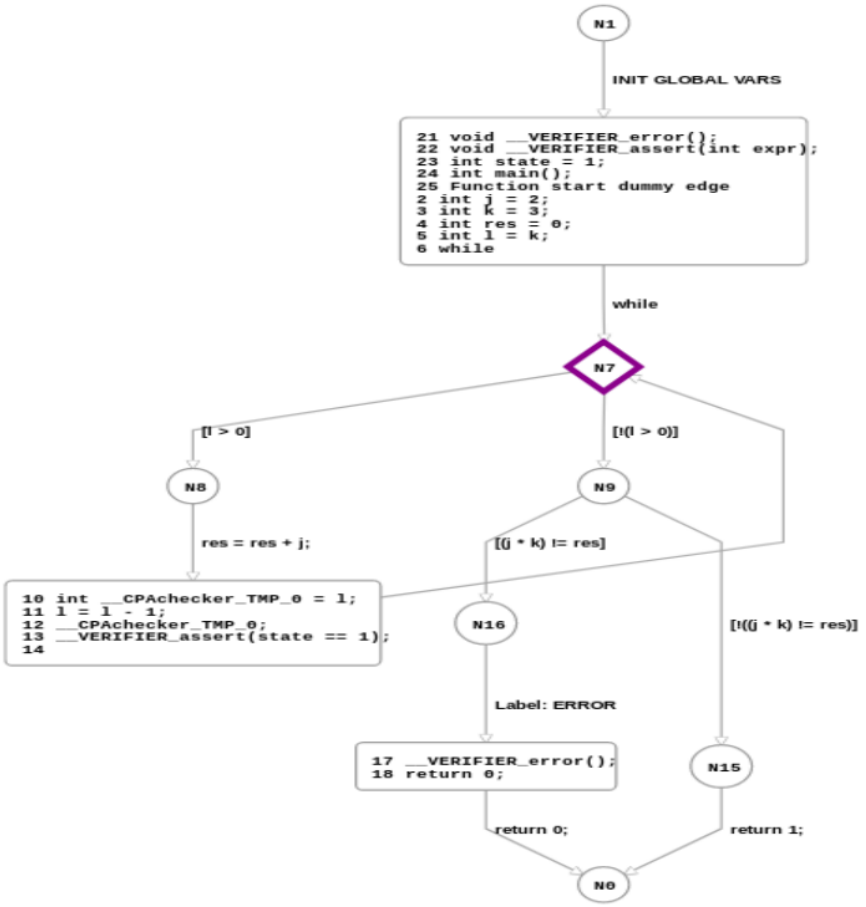

Figure 9: Abstract model

The predicates used to generate the abstract model of the program are:

- $l > 0$
- $jk! = res$

2.3.2 CPAchecker vs CIVL

The code `IsPRIME_CPA.c` was checked in CPA-checker and `ISPRIME_CIVL.c` was evaluated in `CIVL` tool. Both the codes check if the number is prime or not. `CIVL` and `CPAchecker` both were checked the safety property. `CIVL` checks all the possible paths that the program can take and checks if the path holds safety property. The result of the `CIVL` tool indicated that the safety property holds for all the possible paths that the program can take. On the other hand, the `CPAchecker` defines an abstract model and checks if the safety property holds over the abstract model. If the safety property holds over the abstract model, it holds over the concrete model as well. The result of the `CPAchecker` indicated that the safety property holds for the abstract model and hence for the concrete model as well.