NuSMV Game Grid Controller Synthesis

Leonardo Picchiami

SAPIENZA
UNIVERSITÀ DI ROMA

Formal Methods In Software Development - Project

Problem Definition

The game consists of a map (grid) has a goal cell to reach and obstacles and the purpose of the game is to reach the goal cell from a given initial cell.

The goal of the system is to generate a controller, fixed the grid and the goal cell, for that game.

Formal Definitions

- First of all it is necessary to present the mathematical definitions necessary for the generation of the controller.
- The controller that the system will synthesize will have to respect these formal definitions.

Controller

A *controller* for an LTS $\mathcal{S}$ is a function $K : S \times A \rightarrow \mathbb{B}$ such that $\forall s \in S$, $\forall a \in A$, if $K(s, a)$ then $a \in Adm(\mathcal{S}, s)$. We denote with $Dom(K)$ the set of states for which a control action is defined. Formally, $Dom(K) = \{s \in S | \exists a : K(s, a)\}$.

Formal Definitions

Control Law

A *control law* for a controller K is a (partial) function $k : S \rightarrow A$ such that for all $s \in Dom(\mathcal{S})$ we have $K(s, k(s))$.

LTS control problem

A *LTS control problem* is a triple $(\mathcal{S}, I, G)$ where:

- $\mathcal{S} = (S, A, T)$ is an LTS.
- I is the *initial* region.
- G is the *goal* region.

The controller is the solution to this type of problem.

Game Grid Control Problem

It is therefore possible to encode our game grid problem as a control problem.

To define a game grid control problem we must define:

- The Labeled Transition System (LTS) $\mathcal{S}_{game}$.
- The goal region G_{game}.
- The init region I_{game}.

Before providing these definitions, let's fix the concepts of legal action and correct action to define the specific LTS transition relationship of the problem and the other definitions.

Game Grid Control Problem (2)

An action is *legal* if it brings the current state into a successor state, so the cell in which we move, in the neighborhood of the current state and in the successor state there are no obstacles.

An action is *correct* if it is legal and if, from the current state, it is a right action that leads to the goal.

Game LTS Control Problem (3)

LTS game grid

The game grid LTS $\mathcal{S}_{game}$ in the following way

- S_{game} is the set of states where each state is a cell of the grid. Thus $|S_{game}| = N \times M$.
- A_{game} is the set of actions thus defined: {up, left-up, right-up, right, right-down, down, left-down, left}.
- T_{game} is the transition relationship defined as following: $T_{game} : S_{game} \times A_{game} \times S_{game} \rightarrow \mathbb{B}$ such that $\forall s, s' \in S_{game}$ $\forall a \in A_{game}$ $T_{game}(s, a, s')$ holds iff $Neigh(s, a, s') \wedge \neg Obs(s')$.

Game LTS Control Problem (4)

I and G game grid

In general, I_{game} and G_{game} are any subset of S_{game} states. In our case G_{game} corresponds to a fixed cell and it is not necessary to fix I_{game} for the synthesis of the controller.

- For LTS $\mathcal{S}_{game}$, from state s to s' through an action a, a transition is valid if and only if the action a is legal.
- The two following functions are used in the definition of transition relationship for this problem so that the actions for which the transition relationship is defined satisfy the requirement.

Neighborhood and Obstacle Functions

Neighborhood Function

We define the *Neigh* function which tells us if one state is in the neighborhood of the other state and can be reached through an action in the grid game LTS in the following way:

- $Neigh : S_{game} \times A_{game} \times S_{game} \rightarrow \mathbb{B}$ such that $\forall s, s' \in S_{game}$ $\forall a \in A_{game}$ $Neigh(s, a, s')$ holds iff s' is in the neighborhood of s and it is possible to reach s' from s through action a.

Obstacle Function

We define the *Obs* function which tells us if in a state is present an obstalce in the grid game LTS in the following way:

- $Obs : S_{game} \rightarrow \mathbb{B}$ such that $\forall s \in S_{game}$ $Obs(s)$ holds iff in s is present an obstalce.

Example: 3×3 game

Set of states S_{game} for this example:

- $\{(0,0),(0,1),(0,2),(1,0),$ $(1,1),(1,2),(2,0),(2,1),(2,2)\}$

$Dom(K) = \{(0,0),(1,1),$ $(1,2),(2,2)\}$

$K((0,0), right)$ is False.
$K((1,2), left)$ is False.
$K((1,2), down)$ is True because $(2,2)$ is the goal and therefore it is the correct action.

In the **grid**: 1 is an obstacle, 0 otherwise.

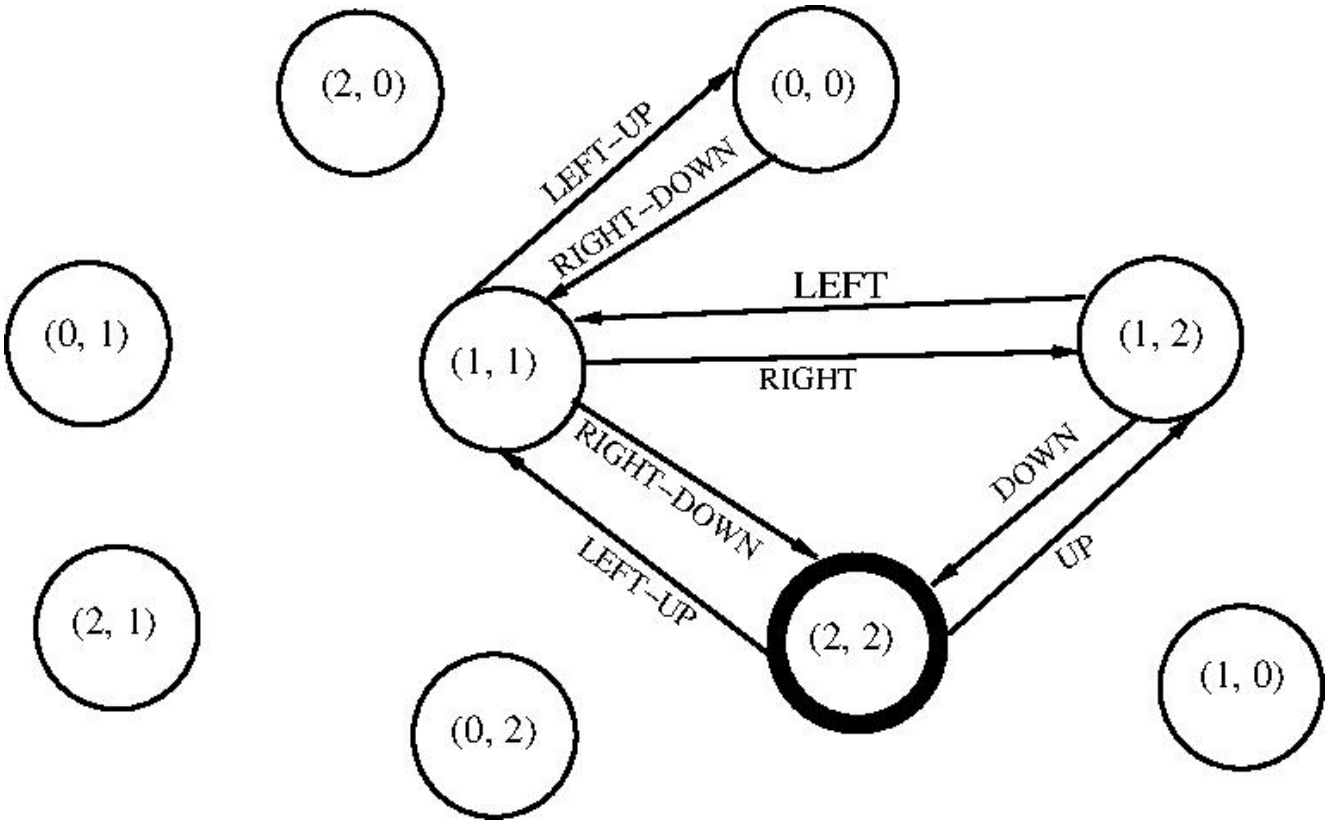
The **grid** of the game is:

0	1	1
1	0	0
1	1	0

The **goal** state is: (2, 2)

The solution to the grid game control problem, is a function set where each state-action pair (s, a) holds iff a is *correct* and s is among the set of states that have a path up to the goal.

Example: 3×3 game (LTS)



Model Checking: Planning

- The generation of the controller was carried out using a Model Checking approach.
- Specifically, the NuSMV model checker was used exploiting the planning technique: planning consists in the negation of a true specification in order to obtain the (minimum) solution as a counterexample.
- Fixed goal and grid, it is possible to establish if a state is controllable by verifying that there is at least one path from the init cell to the goal where the state (the grid cell) is the init cell.
- In our case, the counterexample gives us the states of the Game Grid LTS that are observed along the path from the init to the goal. (In the NuSMV model the init cell must also be fixed).

System Overview

The system of controller synthesis therefore consists of the following basic components:

1. The synthesis algorithm of the controller.
2. The NuSMV model that the system must generate to verify a given cell and to know, by counterexample, if there is at least one path from that cell to the goal using NuSMV as a black box to get the solution to the given problem.
3. The generation of the code of the K function that represents the synthesized controller (and of the Legal function for comparison).
4. The input file for the system, i.e. how the grid and the goal cell are specified to the system.

Two type of controller synthesis algorithm will now be presented: a standard version that processes each cell and check if exists a counterexample. We will then present a version of this optimized algorithm which is based on some crucial observations of the problem.

Useful Functions for Synthesis

The following functions are used in both controller synthesis algorithms:

- *generateNuSMVModel*(S_{game}, *initState*, *goalState*): given the set of states S_{game}, then the grid, the state taken into consideration *initState* as the initial state and *goalState* as the goal state, it generates the NuSMV model for verifying the existence of a path from *initState* to *goalState*. Returns a NuSMV model.
- *NuSMVSubroutine*(*NuSMVModel*): given a NuSMV model as input, this is a blackbox subroutine that invokes NuSMV by passing the *NuSMVModel* as input. Returns the result of NuSMV execution.

Useful Functions for Synthesis (2)

- *existsCounterexample*(*NuSMVResult*): given a NuSMV excecution as input, this function returns true if the NuSMV model checker returned a counterexample. Otherwise returns false.
- *correctAction*(*NuSMVCounterexample*, (*state*, *action*)): given a NuSMV counterexample and a state-action pair as input, the function exploits the counterexample to establish on a given action is *correct* for that state. The function returns true if the action is *correct*, false otherwise.

Controller Synthesis Algorithm (Standard)

Algorithm: Controller Synthesis Standard

Input: Game Grid control problem $(\mathcal{S},\ G_{game})$, $\mathcal{S} = (S_{game},\ A_{game},\ T_{game})$,
function: $ctrSynthStandard(\mathcal{S},\ G_{game})$

1. $K(s,\ a) \leftarrow 0$
2. **forall** $s \in S_{game}$ **do**:
3. $\quad model \leftarrow generateNuSMVModel(S,\ s,\ G_{game})$
4. $\quad resultRoutine \leftarrow NuSMVSubroutine(model)$
5. $\quad$ **if** $existsCounterexample$(resultRoutine) **then**:
6. $\quad\quad K(s,\ a) \leftarrow \exists a, s'\ [T_{game}(s,\ a,\ s') \wedge$
7. $\quad\quad\quad\quad\quad\quad\quad\quad \wedge\ correctAction(resultRoutine,\ (s,\ a))]$
8. **return** $K(s,\ a)$

Useful Functions for Optimized Synthesis

The optimized version uses a very simple assumption: in the counterexample returned by NuSMV we have a sequence of crossed (observed) states to reach the goal state. This means that each intermediate crossed state will also have at least one path leading to the goal state. In this version they are extracted from the counterexample. This will reduce the number of states processed.

To do this we define the following function:

- *exstractsCrossedStates(NuSMVResults)*: extracts from the counterexample returned by NuSMV the crossed states during the verification of the existence of at least one path from the init state to the goal state. Returns the set of crossed states.

Controller Synthesis Algorithm (Optimized)

Algorithm: Controller Synthesis Optimized

Input: Game Grid control problem $(\mathcal{S},\ G_{game})$, $\mathcal{S} = (S_{game},\ A_{game},\ T_{game})$,
function: $ctrSynthOptimized(\mathcal{S},\ G_{game})$

1. $K(s,\ a) \leftarrow 0$
2. **forall** $s \in S_{game}$ **do**:
3. **if** $\forall a \neg K(s,\ a) \wedge \neg Obs(s)$ **then**:
4. $model \leftarrow generateNuSMVModel(S,\ s,\ G_{game})$
5. $resultRoutine \leftarrow NuSMVSubroutine(model)$
6. **if** $existsCounterexample(resultRoutine)$ **then**:
7. $K(s,\ a) \leftarrow \exists a, s'\ [T_{game}(s,\ a,\ s') \wedge$
8. $\wedge\ correctAction(resultRoutine,\ (s,\ a))]$
9. $S_{cross} \leftarrow exstractsCrossedStates(resultRoutine)$
10. **forall** $s_{cross} \in S_{cross}$ **do**:
11. $K(s_{cross},\ a) \leftarrow \exists a, s'\ [T_{game}(s_{cross},\ a,\ s') \wedge$
12. $\wedge\ correctAction(resultRoutine,(s_{cross},\ a))]$
13. **return** $K(s,\ a)$

Algorithms Python Implementation (Standard)

```
def controller_synthesis(self):
    '''
    Generate the controller for the grid, i.e. generate the list of all coordinates that have at least one path leading to the target and the Python code
    of the K function (and the Legal function for comparison) so that you can use it if a pair action-state is controllable.
    This version is the basic algorithm that checks all cells and looks if there is a counterexample returned by NuSMV,
    if there is a path leading to the target and therefore the cell will be added to the set of controlled states and will be enabled to the controller
    the correct action leading to the goal.
    '''

    print("ciao")
    N = len(self.__grid)
    M = len(self.__grid[0])

    states_actions = set()
    for i in range(N):
        for j in range(M):
            self.__checked_cells += 1
            model_path = self.__model_path
            model_path += "model_{0}x{1}-{2}_{3}.smv".format(N, M, i, j)
            ut.nusmv_model_synthesis(N, M, self.__grid, (i, j), self.__goal, model_path)

            checker_res = ut.nusmv_subroutine(self.__nusmv_path, model_path)

            nusmv_parser = nusmv_pars.ModelCheckerParsing(checker_res)
            if nusmv_parser.exists_path_init_to_goal(self.__goal):
                self.__controlled_states.append((i, j))
                nusmv_parser.parse_state_action_conterexample()

                s_a = nusmv_parser.get_correct_state_action()
                if s_a:
                    states_actions.add(s_a)


    if len(self.__controlled_states) > 0:
        self.code_generation(states_actions)

    else:
        print("\n\nThe controller cannot be generated, there have been no states for which no action can be enabled.\n\n")
```

Algorithms Python Implementation (Optimized)

```
def optimized_controller_synthesis(self):
    '''
    Generate the controller for the grid, i.e. generate the list of all coordinates that have at least one path leading to the target and the Python code
    of the K function (and the Legal function for comparison) so that you can use it if a pair action-state is controllable.
    This version is optimized: it only considers the cells that do not have an obstacle and parses the cells crossed
    to reach the goal indicated by the counterexample. The assumption is used that if a cell is crossed to reach the goal,
    it itself has a path that leads to the goal. This reduces the cells in which the path is explicitly checked.
    '''

    N = len(self.__grid)
    M = len(self.__grid[0])

    states_actions = set()
    for i in range(N):
        for j in range(M):
            if (i, j) not in self.__controlled_states and self.__grid[i][j] == 0:
                self.__checked_cells += 1
                model_path = str(self.__model_path)
                model_path += "model_{0}x{1}-{2}_{3}.smv".format(N, M, i, j)
                ut.nusmv_model_synthesis(N, M, self.__grid, (i, j), self.__goal, model_path)

                checker_res = ut.nusmv_subroutine(self.__nusmv_path, model_path)

                nusmv_parser = nusmv_pars.ModelCheckerParsing(checker_res)

                if nusmv_parser.exists_path_init_to_goal(self.__goal):
                    nusmv_parser.parse_state_action_conterexample()
                    states_actions |= set(nusmv_parser.get_postprocessed_state_action())

                    for cell in nusmv_parser.get_crossed_states():
                        if not cell in self.__controlled_states:
                            self.__controlled_states.append(cell)


    if len(self.__controlled_states) > 0:
        self.code_generation(states_actions)

    else:
        print("\n\nThe controller cannot be generated, there have been no states for which no action can be enabled.\n\n")
```

NuSMV Generated Model

The NuSMV model that is generated has the following features:

- The grid is encoded as a global macro array of arrays, along with the dimensions of the grid N and M. This information is therefore not placed in the state space. The grid keeps information about the cells that have an obstacle (with value 1) and the cells without obstacle (with value 0).
- The state space is encoded by the two variables i and j which represent the coordinates of the cell and by the variable *action* which represents the action that is performed by a given cell.
- The state space is therefore given by $D_i \times D_j \times D_{action}$ where D_i is $0..N-1$ and D_j is $0..M-1$ while action has the 8 actions plus the *no-action* action which encodes the possibility of not being able to perform any action from that cell.

NuSMV Generated Model (2)

- The set of initial states (one only initial state) is defined with the values obtained by the system: The values of *i* and *j* are relative to the coordinates of the cell of the processed grid, while *action* has as initial value *no-action*.
- About *action*, the transition relation is defined as follows: for each possible value pair of *i* e *j* that corresponds to the coordinates of a cell in the grid, all possible actions (legal actions) from that cell are assigned non-deterministically so that NuSMV can check each possible action from that cell.
- For *i* and *j* the transition relation is defined on the basis of the next value of *action*, so that the two variables can be increased, decreased or none of the two depending on which action is chosen and in order to pass from a pair of coordinates of a cell to another.

NuSMV Generated Model (3)

- The specification is expressed in the LTL temporal logic which indicates that the goal cell provided by the system will not be reached in the future. So i and j will never take the coordinate values of the goal cell. If exists at least one path, the specification will be violated and NuSMV will return the counterexample.

NuSMV Generated Model (4)

```
DEFINE

    grid := [[0, 1, 1], [1, 0, 0], [1, 1, 0]];
    N := 3;
    M := 3;
```

```
VAR

    i : 0..2;
    j : 0..2;
    action : {no-action, up, right-up, left-up, down, right-down, left-down, left, right};
```

```
ASSIGN

    init(i) := 0;
    init(j) := 0;
    init(action) := no-action;
```

```
next(action) := case
                grid[i][j] = 1 : no-action;
                (i = 0) & (j = 0) : {right-down};
                (i = 1) & (j = 1) : {right-down, right, left-up};
                (i = 1) & (j = 2) : {left, down};
                TRUE : no-action;
                esac;

next(i) := case
        ((next(action) = down) | (next(action) = right-down) | (next(action) = left-down)) & (i + 1 < N) : i + 1;
        ((next(action) = up) | (next(action) = right-up) | (next(action) = left-up)) & (i - 1 >= 0): i - 1;
        TRUE : i;
        esac;

next(j) := case
        ((next(action) = right) | (next(action) = right-up) | (next(action) = right-down)) & (j + 1 < M): j + 1;
        ((next(action) = left) | (next(action) = left-up) | (next(action) = left-down)) & (j - 1 >= 0) : j - 1;
        TRUE : j;
        esac;
```

```
LTLSPEC

    !(F ((i = 2) & (j = 2)))
```

K Function

- The implementation of the K function satisfies the definition given at the beginning: it takes as input a state and an action and returns a boolean value.
- In our case, an output file is generated with the controller's Python code. The K function in the code returns true if the action for that was allows to go the right way for the goal, false otherwise.
- In addition, the Python code of a function (called Legal) is generated, always in the output file, which, given a grid cell as a state and an action, remains true if the action is legal for the state, false otherwise.
- In this way it is easy to see that the controller, for all the controllable states, has not enabled all the legal actions but only a subset of them.

K Function Implementation (3x3 Game Example)

```
def K(state, action):

    if (state[0] < 0 or state[0] >= 3) or (state[1] < 0 or state[1] >= 3):
        raise Exception("The input state is not among the grid states.")

    if action.lower() not in ["up", "left-up", "right-up", "right", "right-down", "down", "left-down", "left"]:
        raise Exception("The input action is not among the actions that can be performed.")

    if state == (1, 1) and action.lower() == "right-down":
        return True

    if state == (1, 2) and action.lower() == "down":
        return True

    if state == (0, 0) and action.lower() == "right-down":
        return True

    return False
```

Legal Function Implementation (3x3 Game Example)

```python
def Legal(state, action):

    if (state[0] < 0 or state[0] >= 3) or (state[1] < 0 or state[1] >= 3):
        raise Exception("The input state is not among the grid states.")

    if action.lower() not in ["up", "left-up", "right-up", "right", "right-down", "down", "left-down", "left"]:
        raise Exception("The input action is not among the actions that can be performed.")

    if state == (0, 0) and action.lower() == "right-down":
        return True

    if state == (1, 1) and action.lower() == "left-up":
        return True

    if state == (1, 1) and action.lower() == "right":
        return True

    if state == (1, 1) and action.lower() == "right-down":
        return True

    if state == (1, 2) and action.lower() == "down":
        return True

    if state == (1, 2) and action.lower() == "left":
        return True

    if state == (2, 2) and action.lower() == "left-up":
        return True

    if state == (2, 2) and action.lower() == "up":
        return True

    return False
```

Legal Function and K Function Interrogation (3x3 Game Example)

```
import synthesis as syn


def game_test():

    print("\n\n\n***** TEST CONTROLLER INTERROGATION *****\n")

    grid = [[0, 1, 1], [1, 0, 'I'], [1, 1, 'G']]

    for i in range(len(grid)):
        for j in range(len(grid[0])):
            print("{0}  ".format(grid[i][j]), end = '')
        print()

    print("\n\n")



    print("Legal(({0}, {1}), {2}) = {3}".format(1, 2, "down", syn.Legal((1, 2), "down")))
    print("K(({0}, {1}), {2}) = {3}".format(1, 2, "down", syn.K((1, 2), "down")))

    print("\n")

    print("Legal(({0}, {1}), {2}) = {3}".format(1, 2, "left", syn.Legal((1, 2), "left")))
    print("K(({0}, {1}), {2}) = {3}".format(1, 2, "left", syn.K((1, 2), "left")))

    print("\n")

    print("Legal(({0}, {1}), {2}) = {3}".format(1, 1, "left-up", syn.Legal((1, 1), "left-up")))
    print("K(({0}, {1}), {2}) = {3}".format(1, 1, "left-up", syn.K((1, 1), "left-up")))


    print("\n")

    print("Legal(({0}, {1}), {2}) = {3}".format(1, 1, "right", syn.Legal((1, 1), "right")))
    print("K(({0}, {1}), {2}) = {3}".format(1, 1, "right", syn.K((1, 1), "right")))

    print("\n")

    print("Legal(({0}, {1}), {2}) = {3}".format(1, 1, "right-down", syn.Legal((1, 1), "right-down")))
    print("K(({0}, {1}), {2}) = {3}".format(1, 1, "right-down", syn.K((1, 1), "right-down")))



if __name__ == "__main__":
    game_test()
```

```
***** TEST CONTROLLER INTERROGATION *****

0  1  1
1  0  I
1  1  G



Legal((1, 2), down) = True
K((1, 2), down) = True


Legal((1, 2), left) = True
K((1, 2), left) = False


Legal((1, 1), left-up) = True
K((1, 1), left-up) = False


Legal((1, 1), right) = True
K((1, 1), right) = False


Legal((1, 1), right-down) = True
K((1, 1), right-down) = True
```

Input File Specifications

The specifications are passed through a file which can be expressed in the following two ways:

Mode 1:

```
Goal: (2, 2)
M: 3
N: 3
Grid:
      0 1 1
      1 0 0
      1 1 0
```

Mode 2:

```
Goal: (2, 2)
M: 3
N: 3
```

The first mode specifies the grid, its size and the goal cell. In the second mode the grid is not specified; in that case the grid is generated randomly using a suitable option. If the option is specified in the first mode, the grid is not parsed.

Output Log

```
********************** NUSMV GAME GRID CONTROLLER SYNTHESIS ***********************


Number of line N: 3
Number of column M: 3
Goal of the grid: (2, 2)


Grid value 0: grid cell without obstacle (traversable cell).
Grid value 1: grid cell with obstacle.
Grid value 2: controlled grid cell.
Grid value G: goal cell.


Game grid:

0 1 1
1 0 0
1 1 G


Controller synthesis algorithm: optimized.


***** Controlled Grid *****


2 1 1
1 2 2
1 1 G
```

```
Controlled states of generated controller:

[(0, 0), (1, 1), (2, 2), (1, 2)]



***** STATISTICS *****

Empty cells: 4.
Cells with obstacles: 5
Controllable cells: 4.
Processed cells: 2.

Execution time: 0.022809982299804688 sec.
Python process RAM usage: 15.4140625 MB
```

Experimental Settings and Tests

The system was developed only on Linux systems. It has been specifically developed and tested on Linux Mint 19 and Ubuntu 16.04 using Python 3.6 and 3.7.

The following experimental tests were carried out on an Intel 1.8GHz i7-4500U processor with 8GB of RAM.

Experimental Settings and Tests (2)

- The following table compares, for each test carried out, the number of cells processed by the standard synthesis algorithm and the number of cells processed by the optimized synthesis algorithm by also observing the number of controlled cells.

N	M	Goal Cell	#CtrlCell	#CellAlgoStand	#CellAlgoOpt
3	3	(2, 2)	4	9	2
3	3	(2, 2)	7	9	4
4	7	(2, 5)	6	28	13
8	6	(1, 2)	8	48	21
10	10	(2, 8)	61	100	48
42	25	(21, 13)	310	1050	349
38	49	(36, 5)	878	1862	473
49	52	(47, 48)	1249	2548	615

Experimental Settings and Tests (3)

- The following table compares, for each test carried out, the execution times of the standard synthesis algorithm and the executions time of the optimized synthesis algorithm.

N	M	#CtrlCell	TimeAlgoStand	TimeAlgoOpt
3	3	4	0.09535s	0.02281s
3	3	7	0.10051s	0.04691s
4	7	6	0.34609s	0.165s
8	6	8	0.64811s	0.31551s
10	10	61	2.29837s	1.19456s
42	25	310	5m 18s	2m 16s
38	49	878	1h 6m 49s	30m 41s
49	52	1249	2h 58m 52s	1h 20m 40s

Experimental Settings and Tests (4)

- The following table compares, for each test carried out, the memory used by Python process with standard synthesis, with optimized synthesis and the maximum memory used by NuSMV in the invocations (managed by a subprocess).

N	M	#CtrlCell	PyMemStand	PyMemOpt	NuSMVMem
3	3	4	15.168 MB	15.414 MB	13.47 MB
3	3	7	15.160 MB	15.027 MB	13.527 MB
4	7	6	15.074 MB	15.121 MB	13.754 MB
8	6	8	15.004 MB	15.145 MB	14.012 MB
10	10	61	15.125 MB	15.301 MB	15.324 MB
42	25	310	15.664 MB	15.564 MB	52.188 MB
38	49	878	15.688 MB	15.652 MB	97.883 MB
49	52	1249	16.480 MB	15.754 MB	119.309 MB

Add-ons

In addition to the generation of the controller for the game grid, there is also a mode that does not provide for the generation of the controller: given a grid, the goal cell and an init cell is computed if there is at least one path from the init cell to the goal cell.

This mode is very similar to the generation of the controller, it actually performs the same operations; only that instead of being carried out on the whole grid, it is done on a single cell.

Then Planning is used in the same way by generating the NuSMV model and using NuSMV as a suboutine to know if exists at least one path from the init cell to the goal cell.

Add-ons (2)

The algorithm used is practically identical to the algorithm of the Standard Controller Synthesis so it is not reported neither pseudocode nor Python implementation. The only difference is that in this mode it does not iterate over the whole grid. So it is not reported on the presentation.

The input file has the same modalities and the same shape as the one for the synthesis of the controller, but the init cell must also be specified in a single line of the file in the following way: Init: (x, y).

Some tests have been carried out where the system says if there is a path from the init cell to the goal cell. But by not generating a controller there are no statistical observations, so they were not reported on the presentation.