Formal Methods for Cyber-Physical Systems

Assignment 1 - Invariant Verification

Prof. Davide Bresolin

A.Y. 2022/2023

Federico Brian	1243422
Hou Cheng Lam	2072114
Kourosh Marjouei	2085109

Table of Contents

1 Introduction

The goal of this project is to replicate the `check_explain_ltl_spec` function in the `mc` module of `PyNuSMV`. `PyNuSMV` is a Python library, wrapper of `NuSMV`, which among other things also includes symbolic model checking algorithms. The `check_explain_ltl_spec` function takes a LTL given specification as an input, and then checks it against the SMV model loaded within the working environment. In our case, the specification is expected to be of type invariant. The function returns a tuple, with its first element returning a boolean, it is `TRUE` if all reachable states of the SMV model satisfies the specification, and `FALSE` otherwise. If the first element of the tuple is `TRUE`, then the second element of the tuple will be set to `None`. Otherwise, the function returns a counterexample of a path showing the SMV model violating the specification, starting from the initial states. Note that there can be more than 1 counterexample if the SMV model violates the specification. This explanation is a tuple of alternating states and inputs, starting and ending with a state.

In this report, we will discuss the methodology used in our implementation to replicate the function `check_explain_ltl_spec`. The correctness of our implementations will also be validated in the discussion section in this report.

2 Methodology

2.1 Model Preparation

To work with any SMV Models, the loaded .smv file needs to be converted into type `BddFsm`, which is a Python class for finite state machine-like (FSM-like) structures, encoded into BDDs. As the SMV model is loaded into the global environment, this can be done by calling `pynusmv.glob.prop_database().master.bddFsm`. The `prop` library provides a method `prop.expr` to extract each specification to check included in the .smv file. As `prop.expr` extracts specifications that are not limited for invariant checking, there is a constraint in place in the Python script to skip these specifications. For each specifications, the function `prop.not_` computes the negation. Finally, we can get the sets of states of `bddfsm` satisfying `nspec` as a BDD by calling the function `SPEC_TO_BDD`, as shown in the snippet below.

```
1: $nspec \leftarrow \text{prop.not\_}(spec)$
2: $bddspec \leftarrow \text{SPEC\_TO\_BDD}(bddfsm, nspec)$
```

By computing `bddspec`, we found the BDD of states that satisfy `nspec`. However, if `spec` is an invariant, then no state contained in `bddspec` should be reachable from the initial states. That is the reason why we chose to compute the reachability BDD (a.k.a. the BDD of reachable states starting from the initial states) and then intersect them with `bddspec`: if the result is an empty BDD, then states in which `nspec` is true are never reached. If that is the case, then `spec` is an invariant.

2.2 Invariant Check and Reachable States

By definition, a specification φ over state variables is an invariant of the transition system if every reachable state satisfies φ. In other words, every reachable state must be true according to the invariant specification.

Using the definition of an invariant specification, we can start building our algorithm by finding all reachable states for any given SMV models. Trivially, initial states are reachable states. By using the initial states specified in the SMV models, the `post` function of `fsm` module can be used to discover the states within the Post Image of the existing BDD structure, namely all states which can be reached from the initial states according to the SMV model, any states from this Post Image which are not members of the initial states are added to the set of reachable states. The remaining reachable states can then be found by recursively applying the `post` function to the current set of reachable states within the BDD structure. Simialrly as above, any newly discovered states are then added to this set/image of reachable states until no new states are found. The final image is the full set of reachable states within a SMV model.

```
1: **function** REACH($bddfsm, init$)
2:     $reach \leftarrow init$
3:     $new \leftarrow$ POST$(bddfsm, reach)$
4:     **while** $new \neq$ INTERSECTION$(reach, new)$ **do**
5:         $reach \leftarrow$ UNION(DIFF$(new, reach), reach$)
6:         $new \leftarrow$ POST$(bddfsm, reach)$
7:     **end while**
8:     **return** $reach$
9: **end function**

10: **PRE** := $bddfsm$ is the BDD of the system's FSM, $init$ is the BDD containing the initial
    states.
11: $reach \leftarrow$ REACH$(bddfsm, init)$
12: **POST** := $reach$ contains the BDD of all reachable states of the SMV Model.
```

2.3 Invariant Specifications

Recall that if a specification is an invariant for a SMV model, then all of its reachable states respect (hold true to) the specification. Therefore, no reachables states should be contained in the BDD returned by `SPEC_TO_BDD`. This is because that BDD represents the states which are not true to the specification. In other words, if the intersection between the set of reachable states and the BDD created by `prop.not_` is empty, then the specification can be concluded as an invariant. The function will return a tuple `(True, None)`, with `True` meaning that the specification of interest is an invariant.

```
1: **if** INTERSECTION$(bddspec, reach) = \emptyset$ **then**
2:     **return** $(True, None)$
3: **end if**

4: **PRE** := $init$ is the set of initial states of $bddfsm$, $bddspec$ is the BDD of states of $bddfsm$
    satisfying $nspec$, $reach$ contains the BDD of all reachable states of $bddfsm$.
5: **POST** := returns $True, None$ if no reachable state starting from $init$ is contained in
    $bddspec$, i.e. if $spec$ is an invariant.
```

2.4 Non Invariant Specifications

However, if the intersection is not empty, then there is at least one reachable state which does not hold true for the specification. In this case, a counterexample is needed as evidence. We can construct a counterexample as follows.

1. Randomly select a reachable state which violates the specification, store this state.

2. Find the pre-image of this state, that is, the set of states that could lead to the selected reachable state in 1 step, using the built-in `pre` function.

3. Store the pre-image found.

4. Find the pre-image of the current pre-image, that is, the set of states that could lead to the selected imagine stored in 3.

5. Repeat step 3 and 4, until we find an initial state.

```
1: $counter\_examples \leftarrow$ INTERSECTION$(bddspec, reach)$
2: **function** BACKWARD_IMAGE_COMP$(bddfsm, init, counter\_examples)$
3:     $images \leftarrow \emptyset$
4:     $pre\_counter\_example \leftarrow$ PICK_ONE_STATE_RANDOM$(counter\_examples)$
5:     APPEND$(images, pre\_counter\_example)$
6:     **while** INTERSECTION$(init, pre\_counter\_example) \neq \emptyset$ **do**
7:         $counter\_example \leftarrow pre\_counter\_example$
8:         $pre\_counter\_example \leftarrow$ PRE$(bddfsm, counter\_example)$
9:         INSERT$(images, pre\_counter\_example)$
10:    **end while**
11:    **return** $images, counter\_example\_original$
12: **end function**

13: **PRE** := $bddfsm$ is the BDD of the system's FSM, $init$ is the BDD containing the initial
    states, $counter\_examples$ contains the set of all states in the SMV Model which do not
    respect the LTL specification.
14: $images \leftarrow$ BACKWARD_IMAGE_COMP$(bddfsm, init, counter\_examples)$
15: **POST** := $images$ contains a list of images where each image contains a set of states which
    are the preiamges of the next image. $counter\_example\_original$ is the random counter
    example selected by this algorithm.
```

$images$ contains the path of states from the initial state of the SMV model, to the randomly selected reachable state which violates the specification for variant checking. The final step of our algorithm is to find the inputs between each interim state in this path, we can construct this as follows:

1. Start from the initial state, we can compute the post image of the state by using the built-in `post` function.

2. Find the intersection between this post image and the second image of $images$, as this is the "next" state which will lead to our counterexample state. Store this state.

3. Find an input required to go from the initial state to this intersection by utilising the functions `GET_INPUTS_BETWEEN_STATES` and `PICK_ONE_INPUTS`. Store this input set.

4. Similar to step 2, find intersection between the post image of the current state and the next image of $images$. Store this state.

5. Similar to step 3, find a possible input required to go from the current state to this intersection. Store this input set.

6. Repeat step 4 and 5, until we reach to the counterexample.

```
1:  function FIND_TRACE($bddfsm, init, images, counter\_example\_original$)
2:      $trace \leftarrow \emptyset$
3:      $start \leftarrow init$
4:      for $i \leftarrow 1$ to $n$ do                                  ▷ LENGTH(images - 1)
5:          $start \leftarrow$ INTERSECTION($start, images[i]$)
6:          $next\_state \leftarrow$ PICK_ONE_STATE($start$)
7:          APPEND($trace, next\_state$)
8:          $post \leftarrow$ INTERSECTION(POST($start$), $images[i+1]$)
9:          $inputs \leftarrow$ GET_INPUTS_BETWEEN_STATES($start, post$)
10:         APPEND($trace$, PICK_ONE_INPUTS($inputs$))
11:         $start \leftarrow post$
12:     end for
13:     APPEND($trace, counter\_example\_original$)
14:     return $trace$
15: end function

16: PRE := $bddfsm$ is the BDD of the system's FSM, $init$ is the BDD containing the initial
    states, $images$ is the output from BACKWARD_IMAGE_COMP function,
    $counter\_example\_original$ is the random counter example selected by the function
    BACKWARD_IMAGE_COMP, which is equivalent to $images[n]$, where $n$ is the index of
    the last member of $images$.
17: $trace \leftarrow$ FIND_TRACE($bddfsm, init, images, counter\_example\_original$)
18: POST := $trace$ contains the states and the transitions (inputs) which constitutes the path to
    get from an initial state to the randomly chosen counterexample.
```

Finally, the set of states and inputs can be returned by the function `check_explain_inv_spec` which shows a counterexample of how a reachable state of the model invalidates the specification. Starting from the initial state, then the first set of inputs, then to the next state, second set of inputs, and repeat until the counter example of reachable state is listed.

```
1:  if INTERSECTION($bddspec, reach$) $\neq \emptyset$ then
2:      $counter\_examples \leftarrow$ INTERSECTION($bddspec, reach$)
3:      $images, counter\_example\_original \leftarrow$
                BACKWARD_IMAGE_COMP($bddfsm, init, counter\_examples$)
4:      $trace \leftarrow$
                FIND_TRACE($bddfsm, init, images, counter\_example\_original$)
5:      return ($False, trace$)
6:  end if

7:  PRE := $bddfsm$ is the FSM of the system represented as a BDD, $init$ is the set of initial
    states of $bddfsm$, $bddspec$ is the BDD of states of $bddfsm$ satisfying $nspec$, $reach$
    contains the BDD of all reachable states of $bddfsm$
8:  POST := returns the couple $False, trace$ where $trace$ contains the states and the
    transitions (inputs) which constitutes the path to get from an initial state to the randomly
    chosen counterexample.
```

3 Discussion

3.1 Basic correctness

Is the True/False answer correct for all cases?

The True/False answer correctly for all cases in our custom function `check_explain_inv_spec`. This is ensured by accepting the invariant only when all reachable states of any SMV models respect the LTL condition. In other words, the set of reachable states is a subset of the set of states which respect the LTL condition, which is equivalent in saying that the intersection between the set of reachable states and the set of states which DO NOT respect the LTL condition, which is the solution of our implementation.

3.2 Symbolic implementation

Is the basic reachability algorithm implemented with a symbolic approach?

As depicted in 2.2, the set of reachable states are computed using BDD operations such as `POST`, `INTERSECTION`, `UNION` and `DIFF`. They are then stored in a BDD-like structure called `bddfsm` of module `fsm` of `PyNuSMV`.

3.3 Correctness of the counterexamples

Are the counterexample real executions of the system? Are they returned in the correct form?

Counterexamples provided by our `check_explain_inv_spec` function are ensured to represent real executions of the system by construction. This is possible because every state of the counterexample is computed twice. Firstly by computing pre-images with the `pre` function, and secondly by computing and crossing post-images with the previously found pre-images with `post` and `intersection` functions. The transition that needs to happen in order for a *pre-state* to reach a *post-state* is computed by the `get_inputs_between_states` function. This machinery allows us to store states and their transitions in a coherent manner. We then return the result as a Python `tuple`, the same type as `pynusmv.mc.check_explain_ltl_spec` function.

3.4 Symbolic counterexample search

Is the search for counterexample implemented with a symbolic approach?

Like the computation of reachable states, the search for counterexamples were implemented by us by exploiting a symbolic approach, namely, by using BDDs and their operations. We built a trace from an initial state to a counterexample by forward computing post-images and intersecting them with states already discovered by the pre-image computation. In this way it is possible to build a trace starting from an initial state and ending to a state in which the invariant property does not hold.

3.5 Justification of correctness

Does the report prove that the algorithm and the counterexample search are correct?

During the process of implementing a solution to this problem, the correctness of the algorithm and the counterexample search were ensured. This is done by using `While` loops in the functions `Reach` and `BACKWARD_IMAGE_COMP`. In both cases, the algorithms would only start

and end under specific conditions, for example, in the `While` loop within `Reach`, the algorithm would only start looking for new reachable under the condition that there are new states in `POST`$(init)$ where their reachable states have not been found, and the algorithm will stop once it does not find any new reachable states. Thus preventing us from looking for states which have already been discovered. This and other main topics are discussed in detail in sections 2.1 and 2.2. The invariant case is described in 2.3, while the non-invariant case in 2.4
The search for counterexamples was implemented with a symbolic approach in our implementation, as it relies solely on using functions provided by the `PyNuSMV` and that the whole function is fulfilled by BDDs. These counterexamples are found by first backward searching preimages from the counterexample until an initial state. By forward finding a valid input with the build in function `get_inputs_between_states` between each interim state from the initial state to the counterexample, this proves that these counterexamples are real executions of the system. The outputs are construct in the same presentation as the built in function `check_explain_ltl_spec`, which are in the correct form as expected.

4 Conclusion

In this report, we have showcased an implementation to replicate the `check_explain_ltl_spec` function in the `mc` module of the `PyNuSMV` Python library. We have explained and reasoned the methodology used in our solution and through the Discussion section, we have ensured that our implementation is correct, has the right (symbolic) approach and that the results from the algorithm matches what is required.

References

[1] Nusmv: A new symbolic model checker. https://nusmv.fbk.eu/.

[2] Pynusmv 1.0rc8 documentation. https://pynusmv.readthedocs.io/.

[3] Rajeev Alur. *Principles of Cyber-Physical Systems*. MIT Press, 2015.

[4] Simon Busard and Charles Pecheur. Pynusmv: Nusmv as a python library. volume 7871 of *LNCS*, pages 453–458. Springer-Verlag, 2013.