Formal Methods for Cyber-Physcial Systems - Invariant Verification Assignment

Federico Brian, Hou Cheng Lam, Kourosh Marjouei

December 2022

Contents

1 Introduction

The goal of this project is to replicate the `check_explain_ltl_spec` function in the `mc` module of the `PyNuSMV` Python library, is a Python wrapper to `NuSMV` symbolic model checking algorithms. The `check_xplain_ltl_spec` function takes a LTL given specification as an input, it is then checked against the SMV model which is loaded within the working environment. The specification is expected to be of type invariant, however this is not verified by the built in function. 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 FSM structure, 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 the 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_` returns any states which do not satisfy the specification, i.e. states where the specifications are false. Finally, we can convert the sets of states from `prop.not_` into a BDD.

$nspec \leftarrow$ prop.not_($spec$)
$bddspec \leftarrow$ SPEC_TO_BDD($bddfsm, nspec$)

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 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.

function REACH($bddfsm, init$)
    $reach \leftarrow init$
    $new \leftarrow$ POST($bddfsm, reach$)
    while $new \neq$ INTERSECTION($reach, new$) **do**
        $reach \leftarrow$ UNION(DIFF($new, reach$), $reach$)
        $new \leftarrow$ POST($bddfsm, reach$)
    end while
    return $reach$
end function

PRE := $bddfsm$ is the BDD of the system's FSM, $init$ is the BDD containing the initial states.
$reach \leftarrow$ REACH($bddfsm, init$)
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 exist in the BDD created by `prop.not_`, as this 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.

if INTERSECTION($bddspec, reach$) $= \emptyset$ **then**
    return $(True, None)$
end if

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, record this state.
2. Find the preimage of this state, that is, the set of states that could lead to the selected reachable state in 1.
3. Record the preimage found.
4. Find the preimage of the current preimage, that is, the set of states that could lead to the selected imagine recorded in 3.
5. Repeat step 3 and 4, until we find an initial state.

```
counter_examples ← INTERSECTION(bddspec, reach)
function BACKWARD_IMAGE_COMP(bddfsm, init, counter_examples)
    images ← []
    pre_counter_example ← PICK_ONE_STATE_RANDOM(counter_examples)
    APPEND(images, pre_counter_example)
    while INTERSECTION(init, pre_counter_example) ≠ ∅ do
        counter_example ← pre_counter_example
        pre_counter_example ← PRE(bddfsm, counter_example)
        INSERT(images, pre_counter_example)
    end while
    return images, counter_example_original
end function

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.
images ← BACKWARD_IMAGE_COMP(bddfsm, init, counter_examples)
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 alogrithm.
```

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 `POST()`.
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. Record this state.
3. Find an input required to go from the initial state to this intersection by applying the functions `GET_INPUTS_BETWEEN_STATES` and `PICK\_ONE\_INPUTS`. Record this input set.

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

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

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

```
function FIND_TRACE(bddfsm, init, images, counter_example_original)
    trace ← []
    start ← init
    for i ← 1 to n do                                    ▷ LENGTH(images - 1)
        start ← INTERSECTION(start, images[i])
        next_state ← PICK_ONE_STATE(start)
        APPEND(trace, next_state)
        post ← INTERSECTION(POST(start), images[i + 1])
        inputs ← GET_INPUTS_BETWEEN_STATES(start, post)
        APPEND(trace, PICK_ONE_INPUTS(inputs))
        start ← post
    end for
    APPEND(trace, counter_example_original)
    return trace
end function

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 se-
    lected by the function BACKWARD_IMAGE_COMP, which is equivalent
    to images[n], where n is the index of the last member of images.
trace ← FIND_TRACE(bddfsm, init, images, counter_example_original)
POST := trace contains the states and inputs which is the path to get from
    the initial state to the 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.

```
if INTERSECTION(bddspec, reach) ≠ ∅ then
    counter_examples ← INTERSECTION(bddspec, reach)
    images, counter_example_original ←
            BACKWARD_IMAGE_COMP(bddfsm, init, counter_examples)
    trace ←
            FIND_TRACE(bddfsm, init, images, counter_example_original)
    return (False, trace)
end if
```

3 Discussion

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.

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.

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.