Formal Methods for Cyber-Physical Systems

Assignment 2 - Verification of reactivity properties

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 implement a symbolic algorithm for the verification of a special class of LTL formulas, using BDDs as data structure to represent and manipulate regions. The class of formulas considered by the algorithm is called "reactivity" properties and have the special form

$$\Box\Diamond f \rightarrow \Box\Diamond g$$

where f and g are Boolean combination of base formulas with no temporal operators.

To do this, we will use the `PyNuSMV` Python library, which is a Python wrapper to `NuSMV` symbolic model checking algorithms. We will reimplement the given function `check_react_spec`($spec$), respecting the following specifications:

1. The function checks if the reactivity formula spec is satisfied by the loaded SMV model or not, that is, whether all the executions of the model satisfy spec or not.
2. The return value of `check_react_spec`($spec$) is a tuple where the first element is `True` and the second element is `None` if the formula is true. When the formula is not verified, the first element is `False` and the second element is an execution of the SMV model that violates spec. The function returns `None` if spec is not a reactivity formula.
3. The execution is a tuple of alternating states and inputs, starting and ending with a state. States and inputs are represented by dictionaries where keys are state and inputs variable of the loaded SMV model, and values are their value.
4. The execution is looping: the last state should be somewhere else in the sequence to indicate the starting point of the loop.

In this report, we will discuss the methodology used in our implementation to replicate the function `check_react_spec`($spec$). 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 reactivity checking, there is already a constraint in place in the Python script to skip these specifications. This is done by first checking whether the spec is of type LTL, then the LTL spec is split into 2 parts: the left and the right. The algorithm then confirms that each of these formulas are of type 'GF', which is a LTL specification representing 'always eventually', or equivalently, 'repeatedly'. Finally, the two specifications are checked to be boolean formulas before the program proceeds with the rest of `check_react_spec`($spec$), which is described below.

As the `check_react_spec`$(spec)$ function does not pass the global FSM environment, we need to first retrieve and store it. The left side and right side of the specification to check for reactivity is then seperated into 2 specifications by `parse_react`$(spec)$, namely f and g.

2.2 Reachable States

Before checking whether a specification satifies the reactivity requirement, we need to create a BDD which contains all reachable states in a FSM structure. Reachable states are states can be visited under a SMV model. 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 Reactivity Check

Recall that the reactivity specification of interest in this report has the special form

$$\Box\Diamond f \rightarrow \Box\Diamond g$$

where f and g are Boolean combination of base formulas with no temporal operators. In other words, a SMV model satisfies a reactive formula whenever a state satisfying f is visited, another state satisfying g will also be visited eventually (with certainty), and this happens infinitely often.

The first thing we have to do in order to solve the requested task, is to negate the reactivity formula, yielding:

$$\neg(\Box\Diamond f \rightarrow \Box\Diamond g) = \Box\Diamond f \rightarrow \Diamond\Box\neg g$$

Therefore, to check if the loaded SMV model respects a reactivity formula, it suffices to check wheter there is a cycle that satisfies $f \ \wedge \ \neg g$, *i.e.*, if $f \ \wedge \ \neg g$ is satisfied repeatedly.

'Repeatably' formulas are a specific type of formulas of the shape:

$$\Box\Diamond f$$

also known as 'always eventually' or 'global future'. Since we already have correct algorithms for computing the satisfiability of a formula in 'Repeatedly' form, then it suffices to implement coherently those algorithm in Python language using `PyNuSMV`'s functions and BDDs as data structures. In such a manner, we ensure overall correctness with a symbolic approach, as requested by this assignment.

In order to proceed in such fashion, we need to compute the negation of g and then retrieve the BDD representing it. In the `PyNuSMV` library, the function `prop.not_`$(spec)$ converts a specification $spec$ to its negative counterpart, *i.e.* where the specification is false. We can use this, along with the `parse_react` function, which retrieves base formulas (f, g) from formula $spec$ if it is of the reactive type, `None` otherwise. Then we make use of the built-in `spec_to_bdd` function to build 2 BDD objects, one containing states that satisfise f and the other containing states that satisfies $\neg g$ in the SMV model. Observe that if the provided LTL formula is not a reactive one, then the main algorithm takes no further actions in checking its satisfiability by terminating immediately.

```
if PARSE_REACT(spec) is None then
    return None
end if
f, g ← PARSE_REACT(spec)
ng ← prop.not_(g)
bddspec_f ← SPEC_TO_BDD(bddfsm, f)
bddspec_ng ← SPEC_TO_BDD(bddfsm, ng)
```

With the BDDs in place, we can start the algorithm with the new BDD $(recur)$ with states that are reachable and satisfying repeatedly both f and $\neg g$. The repeatability algorithm works as follows:

```
reach ← REACH(init)
recur ←INTERSECTION(INTERSECTION(reach, bddspec_f), bddspec_ng)
is_repeatable ← False
pre_reach ← reach
while INTERSECTION(pre_reach, recur) ∧ ¬(is_repeatable) do
    pre_reach ← INTERSECTION(PRE(recur), ng)
    new ← pre_reach
    while new ≠ ∅ do
        pre_reach ← UNION(pre_reach, new)
        if recur ⊆ pre_reach then
            is_repeatable ← True
            break
        end if
        new ← INTERSECTION(DIFF(PRE(new), pre_reach), bddspec_ng)
    end while
    recur ← INTERSECTION(pre_reach, recur)
end while
```

2.4 Reactivity Specifications

If our repeatability algorithm above has failed to find a cycle, *i.e.* `recur` is not entirely contained in pre_reach, then it means that when a state satisfying f is visited, it is not guaranteed that its post states will stay respecting $\neg g$. Then eventually, a post state will satisfy g. In this case, the `break` command in the repeatability algorithm will not be called and the flag $is\_repeatable$ will remain as `FALSE`. The `check_react_spec`($spec$) function will then return a tuple `(True, None)`, with `True` meaning that the reactivity specification of interest is respected.

if NOT $is\_repeatable$ **then**
  return $(True, None)$
end if

2.5 Non Reactivity Specifications

If the specification is not respected by the SMV model, then the above repeatability algorithm would have stopped and have the flag $is\_repeatable$ set as as `True`. In this case, our function `check_react_spec`($spec$) should return a tuple with the first element is `False` and the second element is an execution of the SMV model that violates spec. As our repeatability algorithm above does not look for a specific path recording a path with a state appearing twice, we need to first find the cycle in the SMV model, build the loop and finally, find a path to connect a state in this loop from an initial state.
Recalling the algorithms discussed in lectures, the pseudocodes for finding a cycle is the following:

$recur\_states \leftarrow recur$
$found\_cycle \leftarrow False$
$frontiers \leftarrow \emptyset$
$s \leftarrow$ PICK_STATE($recur\_states$)
while $\neg found\_cycle$ **do**
  $R \leftarrow \emptyset$
  $frontiers \leftarrow \emptyset$
  $new \leftarrow$ INTERSECTION(POST(s), pre_reach)
  while $new \neq \emptyset$ **do**
    $R \leftarrow$ UNION(R, new)
    APPEND($frontiers, R$)
    $New \leftarrow$ INTERSECTION(POST(new), pre_reach)
    $new \leftarrow$ DIFF(new, R)
  end while
  $R \leftarrow$ INTERSECTION($R, recur$)
  if $s \subseteq R$ **then**
    $found\_cycle \leftarrow True$
  else
    $s \leftarrow$ PICK_STATE($recur\_states$)
  end if
end while

Moreover, the algorithm to build a cycle is shown below:

$k \leftarrow 0$
while $s \nsubseteq frontiers[k]$ **do**
    $k \leftarrow k + 1$
end while

$path = [s]$
$curr \leftarrow s$
for $i \leftarrow k - 1$ **downto** 0 **do**
    INTERSECTION(PRE($curr$), $frontiers[i]$)
    $curr \leftarrow$ PICK_STATE($pred$)
    $path \leftarrow$ CONCAT(LIST($path$), $curr$)
end for
$path \leftarrow$ CONCAT(LIST(s), $path$)

Now that we have a path of a loop of states and inputs with f occuring repeatedly without g being true, we can start searching for the states' preimages and repeat until we find an initial state. This initial state would have the quickiest path to this loop, the path is also recorded.

function BACKWARD_IMAGE_COMP
    $images \leftarrow \emptyset$
    $counterex \leftarrow s$
    $pre\_counterex \leftarrow s$
    while INTERSECTION($pre\_counterex, init$) $= \emptyset$ **do**
        $counterex \leftarrow pre\_counterex$
        $pre\_counterex \leftarrow$ PRE($counterex$)
        CONCAT(LIST($pre\_counterex$), $images$)
    end while
    return $images$
end function

$images$ contains the path of states from the initial state of the SMV model, to one of the states in the cycle with f occuring repeatedly without g being true. 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 a state in the counterexample cycle. 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 cycle with f occuring repeatedly without g being true.

function FIND_TRACE($bddfsm, init, images, counter\_example\_original$)
  $trace \leftarrow \emptyset$
  $start \leftarrow init$
  for $i \leftarrow 1$ **to** n **do** ▷ LENGTH(images - 1)
    $start \leftarrow$ INTERSECTION($start, images[i]$)
    $next\_state \leftarrow$ PICK_ONE_STATE($start$)
    APPEND($trace, next\_state$)
    $post \leftarrow$ INTERSECTION(POST($start$), $images[i+1]$)
    $inputs \leftarrow$ GET_INPUTS_BETWEEN_STATES($start, post$)
    APPEND($trace$, PICK_ONE_INPUTS($inputs$))
    $start \leftarrow post$
  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 a state in the cycle which is a counterexample selected by the function BACKWARD_IMAGE_COMP, which is equivalent to $images[n]$, where n is the index of the last member of $images$.
$trace \leftarrow$ 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 repeating cycle as a counterexample.

Finally, the set of states and inputs can be returned by the function `check_react_spec`($spec$) which shows a counterexample of how a repeating cycle of states in 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.

3 Discussion

During the process of implementing a solution to this problem, the correctness of the algorithm and the search for a counterexample were ensured. This is done by using `While` loops in the search for repeated states and counterexamples throughout the `check_react_spec`($spec$) function. For example, the symbolic algorithm of Repeatability would only start and end under specific conditions. In its most outer loop, the algorithm would only start looking for repeated states under the condition that there is a least one state which satisfies the specification f but not g and is reachable in the SMV model. The algorithm will stop once it finds a loop of states which satisfy the specification f but not g, or until all possible states are checked, thus finishing the loop.

The True/False answer correctly for all cases in our custom function `check_react_spec`($spec$), this is ensured by only modifying the flag $is\_repeatable$ when a path of loop is found between states which satisfy the specification f but not g in a SMV model. In another word, there is a possibility that when specification f is satisfied, g may not be satisfied afterwards. In a case where the specification g is respected whenever f is respected, the flag `check_react_spec`($spec$) will not be changed.

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. The counterexamples are found by first finding the repeated loop with states respecting "f but not g" by searching pre- and post-images of states, with functions such as `PRE()`, `POST()` and `INTERSECTION()`. Then a path to the initial states is also found with the `PRE()` function. The inputs between states are found with the function `get_inputs_between_states()` and `pick_one_inputs()`. By working within the FSM environment, 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 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.