Università degli Studi di Padova
Dipartimento di Matematica

Corso di Laurea Magistrale in Computer Science

Laboratory Report

Verification of reactivity property

Work Group:
Davide Franzoso, davide.franzoso.2@studenti.unipd.it
Matricola 2022146

Matteo Mariani, matteo.mariani.1@studenti.unipd.it
Matricola 2029214

Alessandro Benetti, alessandro.benetti.1@studenti.unipd.it
Matricola 1210974

Instructor:
Prof. Davide Bresolin

Anno Accademico 2020-2021

Contents

1 Introduction

In this report we implement the symbolic repeatability check algorithm for the verification of reactivity property and we use BDDs as data structure to represent and manipulate states regions.

In this assignment our aim is to implement an algorithm to verify a special class of LTL formulas, called "reactivity" properties, that have the special form $\Box\Diamond f \rightarrow \Box\Diamond g$, and we want to check whether the transition system has an infinite execution that repeatedly visits a given property.

2 PyNuSMV

PyNuSMV is the python framework used in this project. It is a framework designed for prototyping and experimenting with BDD-based model checking algorithms based on NuSMV. It gives access to some main NuSMV functionalities, such as model and BDD manipulation. NuSMV models can be read, parsed and compiled, giving access to SMV's rich modeling language and vast collection of existing models.

In particular, in this project we have used the following modules:

- init: contains all the functions needed to initialize and close NuSMV.
- glob: provides functionalities to read and build a model from a SMV source file.
- fsm (expect for the method reacheable_state()): contains all the FSM-related structure like BDD represented FSM, BDD represented transition relation.
- prop: defines structures related to propositions of a model
- dd: provides BDD-related structures like generic BDD, lists of BDDs and BDD-represented states, input values and cubes.

3 Algorithms

In this section we are going to describe the 2 main algorithms that have been used in the project. They are actually part of the same algorithm, but we have decided to split their explanation for more clearance. The first algorithm is Symbolic Repeatability Check,(implemented in the function "check_formula()") it is used to understand if an invariant is repeatedly respected or not in a certain model. The second one represents the function that is called in the case where the invariant is not respected, its goal is to retrieve the cycle that violates the invariant repeatedly.

3.1 Symbolic Repeatability Check

Symbolic Repeatability Check is a nested search algorithm used to check whether the transition system has an infinite execution that repeatedly visits a given property. In the case where the invariant is respected, it returns true, otherwise, it returns false. The input of this algorithm is not the invariant itself but its negation. The core step of symbolic verification algorithms is image computation. That is the computation of regions that represent reachable states of the system achieved with the post operator. The dual operator, Pre(regA, Trans), corresponds to the pre-image computation, thath is the ragion that contains all the states from which some state in regA can be reached using one transition. The symbolic algorithm for checking repeatability uses both image computation and pre-image computation. The algorithm has two phases: in the first phase, the algorithm, computes the region Reach of all states reachable from the region Init of initial states by repeatedly applying the image computation operator Post. The second phase aims to find an infinite execution with repeating occurrences of the property φ. We will call this set of repeated occurrence, Recur. This region initially contains all the reachable states that satisfy the property φ and can be computed by intersecting the region Reach (computed in phase one) and the region Spec representing the property. In the first execution we call this region $Recur_0$. For each of the states s in this set, we want to determine if there exists an execution consisting of one or more transitions starting in the state s and ending in some state in $Recur_0$. To compute this, the inner loop repeatedly applies the pre-image computation to find those states from which states in $Recur_0$ can be reached in one or more transitions. In each iteration of the inner loop, the region Reach, that we will call PreReach, is updated by adding the unexplored states. The set of states to be newly explored is obtained by computing the pre-image of the current set New and removing the already explored states in PreReach using the set-difference operation. The inner loop terminates when there are no more new states to be examined. At this point, the region PreReach contains precisely those states that have a path to some state in $Recur_0$.

By intersecting this region with Recur, we obtain the set $Recur_1$, a subset of $Recur_0$. The outer loop is now repeated again with this revised value of Recur. The algorithm can terminate in two ways:

- $Recur_{i+1}$ is empty. In this case there isn't a reachable state that satisfy the property within a loop
- $Recur_{i+1} = Recur_i$. This fact means that there is at least a state that satisfy the property in a loop. If this is the case then the state can be reached repeatability.

Our implementation In this particular assignment we required to verify a reactive formula, a formula that is in the form:

$$\Box\Diamond f \rightarrow \Box\Diamond g \tag{1}$$

So we need to find a loop such that all states respect the negation of the property g and at least one state respects the property f. To achieve this task we decided to give as input to the algorithm both f and the negation of g and, set as $Recur_0$ the intersection between the reachable states and f (in this way we are sure that if there is a cycle with at least one state that has f true we can find it). In order to be sure to find a cycle that has all the states that don't respect g, every time we compute the pre image pre the algorithm interesecates pre with the negation of g

3.2 Retrieve the trace

As already said, the symbolic repeatedly check algorithms can stops in 2 different manner. If it stops in the case represented by the first point, then there's nothing to do, the LTL property is respected and the algorithm returns True. If the algorithm stops in the case represented by the second point, then the liveness property is not respected, this means that there exists a state that violate the invariant repeatedly. If this is the case then we call the function $get\_trace()$, in order to retrieve a "proof" for the fact that the system doesn't respect the invariant, precisely we need a path that, starting from a state in the last set $recur$ (so a state that belongs to the set F) , it forms a loop (so it ends with the starting state) and all the states satisfy the properties already described. So Symbolic repeatedly check doesn't return just a value but a couple of values:

- The boolean value that indicates if the system respects the liveness property or not;
- The trace that represent the loop that contains the states that violate the invariant(in the case where the invariant is respected the path assumes value equal to none)

get_trace() Since we don't know which state in recur appears in a loop we decided to implement the following algorithm:

- for each state in the last $recur$ set found we have checked if it appears repeatedly in a loop. To do that we have performed the following steps:
  - The system starts from the selected recur state and it performs a sequence of $post$ operation until 2 things may happen:
    * at some point the post set (computed by performing the $post()$ operation on the last computed post set, the difference with the last $preReach$ (computed in $check\_formula()$) retrieved and the difference with the states already explored) is empty, this means that we can't build a loop with the considered recur state.
    * the last post set computed contains the same recur state from which we have started the search. This means that we can create a loop with that state. To do so we call the function $rewind()$ that takes in input the selected recur state and the set of post computed in $get\_trace()$

rewind() $rewind()$ is the algorithm in charge to retrieve the trace that goes from an initial state to the state that violates the invariant. To perform this operation we keep in memory each "post" set discovered by get_trace (set of new states to explore) separately, we call the set of these new sets set_of_post. In order to retrieve the path we have decided to start from the end, so from the selected recur state s. Then the algorithm computes its pre-image to get all the states w such that exist a transition from w to s, we call this set a. Now, when $get\_trace()$ algorithm computed the post image of $set\_of\_post_{len(set\_of\_post)-1}$ it got $set\_of\_post_{len(set\_of\_post)}$, this means that there exists at least one state s' in $set\_of\_post_{len(set\_of\_post)-1}$ that is pre-image of the error state s, so it must be contained in a. To retrieve s' it sufficient to perform the intersection between a and $set\_of\_post_{len(set\_of\_post)-1}$. In order to get the input used to go from s' to s we used the function $get\_inputs\_between\_states(s', s)$ that gives the BDD representing the possible inputs between s' and s. The path built up to now is $\{s,\ s'\}$. To get all the other states of the path we can use the procedure just explained iteratively until we reach the starting state,that is, the selected recur state. So, at iteration i:

1. we retrieve the state s' by computing the intersecation between $set\_of\_post_{len(set\_of\_post)-i}$ and the the set a_{i-1} that is the pre-image set computed at iteration $i-1$
2. we get the input using the function $get\_inputs\_between\_states(s', s)$
3. we compute the pre-image of s' a_i

This procedure continues until we compute the intersection between new_1 and the the set $a_{len(new)-1}$ to get the recur state. And it is for sure the recur state because in $get_trace()$ $set\_of\_post_1$ is equal to the recur state. So with this procedure we retrieve a path that, starting from a recur state, it ends up to the same recur state passing through reachable state where the g sentence of the formula $\Box\Diamond f \rightarrow \Box\Diamond g$ is not respected. A prove that this is true is described in the section "Correctness".

4 Correctness

In this section we are going to discuss the correctness of the algorithms implemented in this project. We are going to discuss separately the correctness of symbolic repeatability and the correctness of the algorithm that retrieves the cycle.

4.1 Correctness of symbolic repeatablility check

According to us, a proof of formal correctness is not needed, since we have only adapted the original implementation of the algorithm to verify the reactive formula. To verify that our implementation is correct we have compared the results given by our implementation with the results given by the function check_explain_ltl_spec() that is the function implemented by pyNuSMV, and it outputs a boolean value that represent if an invariant is respected or not, and, if it is not respected, then it outputs a trace.

4.2 Correctness of `get_path()` algorithm

The problem with the verification of the correctness of the trace is that the trace is not unique. This means that we cannot compare our result with the output given by the function check_explain_ltl_spec() in fact, the results can be different, but both corrects. To overcome this problem instead of compare our solution with another solution we decided to verify that the loop we find is a valid one.

4.3 Correctness by Code

Complementary to the implementation of an algorithm are the correctness tests that are provided to sanction its correctness. In the case of the `get_trace()` algorithm, we found it appropriate to test the following assumptions:

- that all the states of the cycle, computed by `get_path()`, respect the negation of the property g and at least one state respects the property f of the formula $\Box\Diamond f \rightarrow \Box\Diamond g$;
- that the starting state of the loop appears twice in the trace set.
- the state that appears twice is a state that belongs to the recur set.

4.4 Results Table

In the following we present a table with some results. The first column represent a certain invariant (grouped by file). The second columns represents our result for each invariant, if the result is false then the respective invariant is not respected otherwise it is respected. The third column represents the result given by the function check_explain_ltl_spec(). In the fourth column there are the values that represent if our result is equal to the result given by the function check_explain_ltl_spec(), for each row. In the last column each value is the output of the function verify_correctness() to verify the correctness of the trace

filename_invariant	**our_results**	**instructor_results**	**matching**	**trace_correctness**
3bit_counter.smv_inv_1	True	True	True	
3bit_counter.smv_inv_2	True	True	True	

filename_invariant	our_results	instructor_results	matching	trace_correctness
3bit_counter.smv_inv_3	False	False	True	True
3bit_counter.smv_inv_4	True	True	True	
gigamax.smv_inv_1	False	False	True	True
gigamax.smv_inv_2	False	False	True	True
syncarb5.smv_inv_1	True	True	True	
syncarb5.smv_inv_2	False	False	True	True
delay_inverter.smv_inv_1	True	True	True	
delay_inverter.smv_inv_2	True	True	True	
mutex.smv_inv_1	True	True	True	
mutex.smv_inv_1	True	True	True	
switch.smv_inv_1	True	True	True	
switch.smv_inv_2	True	True	True	
switch.smv_inv_3	False	False	True	True
railroad.smv_inv_1	False	False	True	True
railroad.smv_inv_2	True	True	True	
railroad.smv_inv_3	True	True	True	

Table 1: summury table of correctness.

As we can see from the table our results are equal to the results given by the function check_explain_ltl_spec(), and all the traces that we have computed pass the test described in the section 4.3 (note that if the invariant is true then the cell in the last column is empty since we don't output the trace in that case).