Formal Methods for Cyber-Physical Systems

Academic Year 2024/2025

Authors	Matteo Basso, 2075223
	Marco Positello, 2078261
Period	January 2025

Description
Report for Assignement 1.

Contents

List of Figures

Listings

1 Introduction

This report contains a description of the work we have done for this assignment.
The assignment consists of two parts:

1. Safety properties: implementation of the symbolic algorithm for invariant verification, using BDDs as data structure to represent and manipulate regions.

2. Reactivity properties: implementation of a symbolic algorithm to verify a special class of LTL formulas, using BDDs as data structure to represent and manipulate regions.
   The formulas considered by the algorithm are called **Reactivity formulas**, and have the special form:

$$\bigwedge_{i=1}^{n}(\Box\Diamond f_i \rightarrow \Box\Diamond g_i)$$

Figure 1.0.1: Reactivity formula

2 Safety properties

2.1 General description

The first part of the assignment consists of implementing the symbolic algorithm for invariant verification, using BDDs as data structure to represent and manipulate regions.

2.2 How the algorithm works

First of all we have implemented the following function:

```
 1  def witness_reachable (model : BddFsm, phi: BDD)
 2    -> Tuple[bool, Optional[List[dict]]]:
 3      reach = model.init
 4      new : list[BDD] = [model.init]
 5      k = 0
 6      while model.count_states(new[k]):
 7          if new[k].intersected(phi):
 8              s = model.pick_one_state_random(new[k] & phi)
 9              path = [s.get_str_values()]
10              for i in range(k-1, -1, -1):
11                  pred = (model.pre(s)) & new[i]
12                  pre_s = model.pick_one_state_random(pred)
13                  input = model.get_inputs_between_states(pre_s, s)
14                  path.insert(0,
15                      model.pick_one_inputs(input).get_str_values()
16                  )
17                  path.insert(0, pre_s.get_str_values())
18                  s = pre_s
19              return (True, path)
20          new.append(model.post(new[k]) - reach)
21          k = k+1
22          reach = new[k] + reach
23      return (False, None)
24
```

Listing 1: witness_reachable function

Given a region (composed by a set of states and a transition system) and a property (both as BDDs), this function, using a Symbolic Breadth-First-Search approach, checks if there exists at least one reachable state in which the given property is satisfied.

To do so, we use the variables *reach* which contains the set of actual reachable states and the list *new* that contains all new states founded on the *k-th* iteration.

In the row 6 the guard of the while cycle consists of checking if there are some new states founded during the previous iteration:

- If there are some new states then we firstly check if the property phi (φ) is satisfied for at least one of the new states (row 7); if so then we can pick one state that satisfies the property and start building a path from the picked state to an initial state, going backwards through the list. *new*, saving also one of the possible inputs between each state of the path. Finally, we return *True* and the path as witness of the satisfiability of the property (lines 8 to 19).
  If the property is not satisfied by any of the new states then we need to compute the set of

the new reachable states from the actual reachable states (lines 20 to 22, selecting only the states that we have never seen during the previous iterations), then the while cycle restart.

- If there are no new states it means that there are no reachable states that satisfies the property so we return *False*.

This algorithm always terminates because either we found a state in which the property is satisfied or we visit all the reachable states without finding any accepting state, taking into account that the number of states is finite.

We have also implemented the following function:

```python
def my_check_explain_inv_spec(model : BddFsm, spec : Spec)
  -> Tuple[bool, Optional[List[dict]]]:
    bdd_spec = spec_to_bdd(model, spec)
    bdd_notspec = bdd_spec.not_()
    res, trace = witness_reachable(model, bdd_notspec)
    return (not res, trace)

```

Listing 2: my_check_explain_inv_spec function

This is the function that, given a region (composed by a set of states and a transition system) and a property, checks if the specification is an invariant or not.

The specification is encoded as a BDD, then we use the function 1 with the negation of the specification to obtain the following result:

- If the function returns *True* it means that there exists a reachable state in which the negation of the specification is satisfied, hence the specification is not an invariant, so we return *False* and the execution provided by the previous function as witness.
- If the function returns *False* it means that there does not exist a state in which the negation of the specification is satisfied, hence the specification is an invariant and so we return *True*.

3 Reactivity properties

First, we present the general symbolic algorithm to verify the repeatability of some property, and then we show how we modified the general algorithm to verify a reactivity property.

3.1 Symbolic Repeatability Algorithm

3.1.1 General description

The goal of the algorithm is to find any possible **infinite execution** where the state F appears *repeatedly*, in other words, if in a region that contains a state F, this state is visited *repeatedly* infinitely many times.

To do so we use the *Symbolic Repeatability Algorithm*, where the key step is: given a region A, find a sub-region s where exists a state t that is reachable from s with ≥ 1 step. Then we have to find, if exists, an infinite execution that starts in s and lead back to itself, hence, satisfied *Repeatedly* F.

3.1.2 How the algorithm works

3.1.2.1 First Phase (Reachable states)

The first phase is about **computing** the set of **reachable states**. To archive this goal we will iterate over all the states that belongs to the region S with its transitions system *Trans*. Inside the loop we use the functions *Post* and *Diff* where:

- **Post(A,Trans)**: computes the post-image (i.e. the set of successors) of state variables S in a region A:
    - Take the intersection of the region A and the transition system *Trans* ($S \cup S$').
    - With the existential quantification, project out the variables in S.
    - Rename the state variables S' to obtain a region over S.
- **Diff(A, B)**: Returns a region containing all the states in A and not in B.

At the end of the iteration we have a set *Reach* that represents all the reachable states.

The pseudocode of this phase is the following:

Input: region *Init* over S and region *Trans* over S U S'
Output: the region representing all reachable states

```
Reach = Init
New = Init
while not IsEmpty(New) do
    New = Diff(Post(New, Trans), Reach)
    Reach = Union(Reach, New)
end while
```

Listing 3: Pseudocode for computing the set of reachable states

3.1.2.2 Second Phase (Symbolic Repetability Check)

The second phase involves the **Symbolic Repeatability Check** in order to find if there exists an execution where the formula F is *Repeatedly* satisfied.

The crucial step is to find the sub-region s of A where exists $t \in A$ that is reachable from s with ≥ 1 transitions.

As for computing the set of reachable states, we need to repeatedly apply the **post-image operator** to compute the set of states which can reach A, we repeatedly apply the **pre-image operator** and **intersect** the result with the the region A (later in chapter 3.2 and also in the code we call this region *New*).

The invariant of the algorithm is that the number of reachable states will be reduced for every iteration until nothing left. This means that no execution is found where formula F is repeated. Otherwise, if the number of reachable states is not reduced compared to the previous iteration, it means that we have found an *infinite execution* that satisfying *Repeatedly F*.

The pseudocode of this phase is the following:

Input: regions *Init* and F over S and region *Trans* over *S U S'*
Output: returns *True* if F is repeatable, *False* otherwise

```
Reach = phase1_algorithm()
Recur = Intersect(Reach, F)
while not IsEmpty(Recur) do
    PreReach = EmptySet()
    New = Pre(Recur, Trans)
    while not IsEmpty(New) do
        PreReach = Union(PreReach, New)
        if IsSubset(Recur, PreReach) then
            return True
        end if
        New = Diff(Pre(New, Trans), PreReach)
    end while
    Recur = Intersect(Recur, PreReach)
end while
return False

```

Listing 4: Pseudocode for checking the repeatability of a property

3.1.2.3 Third Phase (Finding witnesses)

The third phase focuses on finding **witnesses** of *Repeatability* of F, i.e. a **lasso-shaped** execution that starts from an initial state $s \in F$ and returns to itself.

To correctly compute the witness, we need to follow these steps:

1. Build the **Prefix** (i.e. the path from an initial state to the first state of the **cycle** s):

   We need to find the region that contains s, and then compose the path from an initial state to the state s.

   Let's call:

- *New*: set initial states.
- *Reach*: set of states that can be reached from *New* and initialized with only the initial states.
- *Trace*: list of regions that can be reached during execution and leads to the region where resides s.

Now while $s \notin New$:

(a) Compute the region *New'* with the **post-image operation** and **Diff** with the set *Reach*. By doing this, we ensure that *New'* contains only the states that can be reach from *New* and are effectively new states and not already contained in *New*.

(b) Add to *Reach* the set of states *New'*.

(c) Add to *Trace* the region *New'* and assign *New'* to *New*.

At this point, we can build the *prefix* path, iterating over *Trace* (if *Trace* is empty, it means that, for sure a cycle exists but it starts from an initial states, and so a prefix is not needed):

(a) Pick a state from the result of the **pre-image operation** between s and the i-Th element of *Trace* and prepend to *prefix*.

The pseudocode for building the prefix is the following:

```
New = Init
Reach = Init
Trace = [New]
while not New.Contains(s) do
    New = Diff(Post(New, Trans), Reach)
    Reach = Union(Reach, New)
    Trace.append(New)
end while
if trace.length < 1 then
    return []
end if
Prefix = []
current_state = s
for i = (Trace.length - 1) to 1 do
    current_state =
        Pick(Intersect(
            Pre(current_state, Trans), Trace[i])
        )
end for
return Prefix
```

Listing 5: Pseudocode for computing the set of reachable states

2. Find the **Cycle** (i.e. the actual **cycle** path from s to itself):

(a) Pick a state s from the set of *reachable states*.

(b) Compute the set of states R s.t. $t \in R$ and t is reachable from s with ≥ 1 transitions. All of the transitions must be contained in the region *PreReach* (i.e. the region containing all the reachable states from the previous iteration).

(c) If $s \in R$ then we can build the *loop* from s to itself, otherwise we need to choose another state $s' \in R$ and restart from step 2b.

The pseudocode for finding the cycle is the following:

```
s = PickState(Recur)
while True do
    R = EmptySet()
    New = Intersect(Post(s,Trans), PreReach)
    while not IsEmpty(New) do
        R = Union(R, New)
        New = Intersect(Post(New,Trans), PreReach)
        New = Diff(New, R)
    end while
    R = Intersect(R, Recur)
    if s      R then
        return s
    else
        s = PickState(R)
    end if
end while

```

Listing 6: Pseudocode for finding the cycle

3. Build the **loop**:
   (a) While computing R we need to keep track of the sequence of **frontiers**, called New_1, ..., New_n.
   (b) Find the index k s.t. $s \in New_k$.
   (c) If $k > 1$, we need to find the set predecessors of s, intersect it with New_{k-1}, and then choose an arbitrary element from the resulting set.
   (d) Keep doing until $k = 1$, so we obtain a path from s going back to itself.

The pseudocode for building the loop is the following:

```
for i = 1 to k do
    if New[k].Contains(s) then
        break
    end if
end for
path = [s]
curr = s
for i = k      1 to 1 do
    Pred = Intersect(Pre(curr,Trans), New[i])
    curr = PickState(Pred)
    path = [curr].append(path)
end for
path = [s].append(path)
return path

```

Listing 7: Pseudocode for building the loop

3.2 Adaption to Reactive Formula

To find a witness of an infinite loop, we need to negate the reactive formula:

$$\bigwedge_{i=1}^{n} (\Box\Diamond f_{\mathrm{i}} \rightarrow \Box\Diamond g_{\mathrm{i}})$$

Obtaining:

$$\bigwedge_{i=1}^{n} (\Box\Diamond f_{\mathrm{i}} \wedge \Diamond\Box\neg g_{\mathrm{i}})$$

Now we need to check the *Repeatability* of the formula F_i while formula $\neg G_i$ *persistently* holds. In other words, we need to check that $\neg G_{\mathrm{i}}$ also holds while computing the set *New* in the *second phase* by intersecting the region that represents $\neg G_{\mathrm{i}}$ with the region *New*.

The following function is the implementation in Python of the symbolic repeatability algorithm adapted to reactive formulas:

```
        def reactiveness_symbolic_repeatability(model : BddFsm,
                                                 f : BDD, g : BDD)
          -> Tuple[bool, Optional[List[State]], Optional[int]]:
            reach : BDD = get_reachable_states(model)
            recur : BDD = reach & f & ~g
            while model.count_states(recur) > 0:
                new = model.pre(recur) & ~g
                pre_reach : BDD = BDD.false(model)
                while model.count_states(new) > 0:
                    pre_reach = pre_reach | new
                    if recur.entailed(pre_reach):
                        s, cycle = get_cycle(model, recur, pre_reach)
                        prefix = get_prefix(model, s)
                        return True, (prefix + cycle), len(prefix)
                    new = (model.pre(new) & ~g) - pre_reach
                recur = recur & pre_reach
            return False, None, None

```

Listing 8: reactiveness_symbolic_repeatability function

The functions *get_ reachable_ states*, *get_ cycle* and *get_ prefix* are simply the implementation of the functions described in section 3.1.2.

3.3 Conclusion

The function that solves the problem of the second part, using all the function described in the sections above is:

```
1  def my_check_explain_react_spec(model : BddFsm, spec : Spec)
2    -> Optional[Tuple[bool, Optional[List[dict]]]]:
3      if not parse_react(spec):
4          return None
5      reactivity_formulas = build_implies_formula_list(spec)
6      for f in reactivity_formulas:
7          f_spec, g_spec = get_components_of_implies_spec(f)
8          f_bdd = spec_to_bdd(model, f_spec)
9          g_bdd = spec_to_bdd(model, g_spec)
10         result, states_list, prefix_length =
11             reactiveness_symbolic_repeatability(model, f_bdd, g_bdd)
12         if result:
13             counterexample : List[dict] =
14                 [states_list[0].get_str_values()]
15             for i in range(0, len(states_list) - 1):
16                 curr = states_list[i]
17                 next = states_list[i+1]
18                 input = model.pick_one_inputs_random(
19                     model.get_inputs_between_states(curr, next)
20                 )
21                 counterexample.append(input.get_str_values())
22                 counterexample.append(next.get_str_values())
23             return False, counterexample, (prefix_length * 2)
24     return True, None
25
```

Listing 9: my_check_explain_react_spec function

First of all we must be sure that the given specification is a reactive formula (line 3).

Then, using a function that splits the components of a reactive formulas in a list of formulas with the form:

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

and, for each component of the list, we use the adapted function 8 to check if the formula is satisfied or not.

If all the formulas that compose the list are satisfied then we return *True*, otherwise if the function 8 return *False* for one component of the list, it means that there are at least one component that is not satisfied by the algorithm.

So, in order to build the execution and return it as a proof, we use the list of states returned by the function 8, then extract the information about the states and the inputs between states, and finally compose a list of states and inputs that represent the execution (lines 12-23).