GR(1) Verification

Luca Zaninotto – 2057074

General reactivity of rank 1

In temporal logic, general reactivity formulas of rank 1 are formulas in the shape

$$(\Box\Diamond f_1 \land \Box\Diamond f_2 \land \cdots \land \Box\Diamond f_n) \to (\Box\Diamond g_1 \land \Box\Diamond g_2 \land \cdots \land \Box\Diamond g_n)$$

In other words

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

To prove that a given model respects such specifications we can see it in another way, and verify weather the model does *not* satisfy

$$\left(\bigwedge_{i=1}^{n} \Box\Diamond f_i\right) \land \left(\bigvee_{i=1}^{n} \Diamond\Box(\neg g_i)\right)$$

By seeing the problem this way we have a clearer way to implement an algorithm to find weather a given specification is satisfied or not, following the general scheme:

- find the loops respecting $\Box\Diamond f_1$, from that, find the loops respecting $\Box\Diamond f_2$, and so on, until you either found a loop satisfying $\bigwedge_{i=1}^{n} \Box\Diamond f_i$ or an empty set.
  - If an empty set was found, the property is respected, the hypothesis is false.
- if not, for each $i \in \{1, \ldots, n\}$, starting from the loops respecting $\bigwedge_{i=1}^{n} \Box\Diamond f_i$ search for a loop respecting $\Diamond\Box\neg g_i$
  - if any of such loop is found, return a counterexample

Let's look at each individual step and its implementation to see how it works.

Loops respecting $\Box\Diamond f_n$

To find each the loop respecting $\Box\Diamond f_n$ we use the algorithm for repeatability check we saw in class:

```
def GF(spec, reach):
    """check weather the model, in the `reach` subset verifies
    G ( F (spec)).
    If it does, it returns the set of frontiers calculated
    """

    fsm = pynusmv.glob.prop_database().master.bddFsm
    if not reach:
        return None
    recur = reach * spec
    while (fsm.count_states(recur) != 0):
        reach = pynusmv.dd.BDD.false()
        new = fsm.pre(recur)
        news = [new, recur]
        while (fsm.count_states(new) != 0):
            reach = reach + new
            if recur.entailed(reach):
                return news
            new = (fsm.pre(new)) - reach
            news = [new] + news
        recur = recur * reach
    return None
```

The implementation works by constructing subsequent frontiers for which the pre-image eventually returns to some subset of the initial `recur` set. If at some point no such subset is found (the subset is the empty set) then there is no loop that satisfies $\Box\Diamond f_n$ starting from the `reach` subset, otherwise the explored frontiers are returns (from which we can build the set of loops). To find the intersection of the two we have to search for loops that both respect $\Box\Diamond f_n$ and $\Box\Diamond f_{n+1}$. In order to achieve this the simple intersection of the sets is not enough, we have to re-run the algorithm that checks $\Box\Diamond f_n$ on the loop set built starting from the frontiers of the last run, this way we find a new set of frontiers, contained in the first one that also respects the second one:

```
fsm = pynusmv.glob.prop_database().master.bddFsm

# find fs and gs from formula
fs, gs = parse_gr1(spec)

loop_set_f = reachability(fsm)
for f in fs:
    bdd = spec_to_bdd(fsm, f)
    fronts_f = GF(bdd, loop_set_f)
    loop_set_f = loops(fronts_f)
```

The first formula is checked in the reachable set (as usual), the second starting from the loop set of the first one, and so on, until all the formulas f are checked. Once this step is done either `loop_set_f` contains a valid loop set, which means that the hypothesis is `True`, therefore we have to check for the conclusion; or the set is empty, therefore the hypothesis is false and the formula is respected: we can simply return `(True, None)`.

In order to execute this step we have to define the function `loops`, building the set of all loops starting from the explored frontiers (removing all the unwanted paths in the computation of $\Box\Diamond f_n$):

```
def loops(frontiers):
    """Returns the /reach/ set build from the frontiers given as
    input, this is useful in the forward phase to restrict more and
    more the set /reach/ in the FG and GF algotihms

    """
    if not frontiers:
        return None
    fsm = pynusmv.glob.prop_database().master.bddFsm
    frontiers = list(reversed(frontiers))
    reach = frontiers[0]
    new = frontiers[0]
    for el in frontiers[1:]:
        new = fsm.post(new) * el
        new = new - reach
        reach = reach + new
    return reach
```

Essentially since the frontiers are computed using the pre-image of sets, the loop set is built "going forward", using the post-image.

Loops respecting $\Diamond\Box g_n$

If we arrived in this step, it means there are loops respecting $\bigwedge_{i=1}^{n}\Box\Diamond f_i$. Starting from these loops, we have to find some loop for which one of the g_i holds. Note that the starting point is *always* the loops respecting $\bigwedge_{i=1}^{n}\Box\Diamond f_i$, if some loop respecting $\diamond\Box g_i$ for some $i \in \{0, \dots, n\}$, this means that the formula is not respected: returning the intermediate exploration frontiers helps us build the counterexample.

```
def FG(spec, reach, recur):
    """Check weather the model, in the `reach` subset verifies
    F ( G(spec))
    If it does, it returns the set of frontiers calculated
    """

    fsm = pynusmv.glob.prop_database().master.bddFsm
    # reach = reachability(fsm)
    if not reach:
        return None
    recur = recur * spec
    while (fsm.count_states(recur) != 0):
        reach = pynusmv.dd.BDD.false()
        new = fsm.pre(recur) * spec
        news = [new, recur]
        while (fsm.count_states(new) != 0):
            reach = reach + new
            if recur.entailed(reach): # recur == reach
                return news
            new = (fsm.pre(new) * spec) - reach
            news = [new] + news
        recur = recur * reach
    return None
```

We can notice it is essentially the same algorithm to check $\Box\Diamond\varphi$, but in this case each frontier is calculated also intersecting with the property we're checking.

Building the counterexample

If the latter step results in something different from an empty set, it means that exists some loop that satisfies

$$\exists j \mid \bigwedge_{i=1}^{n} f_i \wedge \neg g_j$$

This means we can build a witness for the falsehood of the initial implication

$$\bigwedge_{i=1}^{n} f_i \rightarrow \bigwedge_{i=1}^{n} g_i$$

Starting from the frontiers returned by the FG algorithm we can output a loop with a variation of the algorithm seen in class:

1. compute recur and reach from the frontiers set, pick one state in recur, will be our first guess for the initial loop state s.
2. compute subsequent frontiers of new states based on the post-image of the latter one (starting from s), until there are no states in the last frontier, keeping track of all the union of all new frontiers r.
    - If s is not inside r pick another state in the intersection between r and reach, repeat step 2.
3. build the loop based on the frontiers, starting from s:
    - compute the pre-image of the currently considered node, intersect it with the frontier built on the post-image (call pred the intersection)
    - select one node in the pred set
    - expand the loop with the new node
    - repeat for all frontiers

```
def counterexample(frontiers):
    """Given a list of frontiers of the FG algorithm, build
    a lasso-shaped execution that starts from the model initial states
    and loops over the states given by the frontiers

    """
    fsm = pynusmv.glob.prop_database().master.bddFsm
    recur = frontiers[-1]
    frontiers = frontiers[:-1]
    reach = reduce(lambda x,acc: x+acc, frontiers)
    s = fsm.pick_one_state(recur)
    while True:
        r = pynusmv.dd.BDD.false()
        new = fsm.post(s) * reach
        r_front = [new]
        while not fsm.count_states(new) == 0:
            r = r + new
            new = (fsm.post(new) * reach) - r
            r_front = r_front + [new]
        r = r * reach
        if s.entailed(r):
            i = 0
            for front in r_front:
                if s.entailed(front):
                    break
                i += 1
            head = head_to(s)[:-1] # forward [init ... s]
            # build the loop, in a /reverse/ order (with the pre-image)
            loop = [s]
            curr = s
            for new in reversed(range(i)):
                pred = fsm.pre(curr) * r_front[new]
                curr = fsm.pick_one_state(pred)
                loop = loop + [curr]
            loop = loop + [s]
            # head = [init ... ], loop = [s ... s] // reversed
            path = head + list(reversed(loop))
            return path
        else:
            s = fsm.pick_one_state(r)
```

Essentially working like the algorithm shown in class, with the only exception that it is initially reconstructing the recur and reach sets from the frontiers. It also returns the whole lasso-shaped execution, starting from an initial node and leading to the loop. We can do this because the frontiers built by the `FG` algorithm ensure that all the loops inside the `reach` set respect $\Box\neg g_i$ for some $i \in \{i, \ldots, n\}$.

On the implementation

The implementation uses in some parts of the code the `functools` package. It should be included with the latest releases of python, but in case it is not, the user can install it with:

```
pip install functools
```