Stefano Panozzo - 1211143

RELAZIONE PROGETTO SINGOLO - ALWAYS AND EVENTUALLY

Per l'implementazione dell'algoritmo per la verifica delle formule di tipo *always* ed *eventually*, si sono prima definiti due metodi di supporto: `isEmpty (BddFsm, BDD)` e `reachable_states(BddFsm)`. Il primo ritorna `True` se il BDD è vuoto, il secondo ritorna la regione di stati raggiungibili del modello in input.

```
def isEmpty(model, x):
    """
    Return True if the BDD is empty
    x : BDD
    model : finite-state machine of the model
    """
    if model.count_states(x) == 0:
        return True
    return False
```

```
def reachable_states(model):
    """
    Returns the region representing all reachable states, given the initial
states of the model
    model : finite-state machine of the model
    """
    init = model.init
    reach = init
    new = init
    while not(isEmpty(model, new)):
        new = model.post(new).diff(reach)
        reach = new.union(reach)
    return reach
```

Always

Per capire se una proprietà è sempre soddisfatta bisogna mostrare che ogni esecuzione del sistema non presenti alcuna occorrenza della proprietà negata.

Per verificare che quindi una formula soddisfi *always* bisogna prima trovare gli stati raggiungibili che non facciano parte del set trovato dal metodo `spec_to_bdd(BddFsm, Spec)`, ovvero trovare l'insieme di stati che non soddisfano la proprietà. Una volta trovati questi stati che invalidano la formula, è sufficiente sceglierne uno (chiamato `bad_state`) e costruire un controesempio risulta semplice, in quanto basta trovare un'esecuzione del sistema che contenga lo stato trovato precedentemente senza preoccuparsi del fatto che gli altri stati dell'esecuzione soddisfino o meno la proprietà. Per fare ciò, si può suddividere la ricerca del controesempio in due parti: la prima trova la traccia dallo stato `bad_state` ad un qualsiasi stato iniziale del sistema, la seconda trova la traccia dallo stato `bad_state` ad uno stato già presente nella traccia, così da terminare l'esecuzione in caso di ciclo.

```
def check_explain_always(spec):
    ltlspec = pynusmv.prop.g(spec)
    print("always ltlspec: ", ltlspec)
    model = pynusmv.glob.prop_database().master.bddFsm
    bddspec = spec_to_bdd(model, spec)
    trace = []
    res = True
    reachable = reachable_states(model)
    not_property = reachable.diff(bddspec)
    # check the formula and build the counterexample trace
    if not(isEmpty(model, not_property)):
        res = False
        bad_state = model.pick_one_state(not_property)
        trace.append(bad_state.get_str_values())
        state = bad_state
        reached = pynusmv.dd.BDD.false()
        # add in 'trace' the elements - states and inputs - from init state to
'bad_state'
        while not(isEmpty(model, state.diff(model.init))):
            np_state = model.pre(state)
            np_state = np_state.intersection(reachable)
            inp = model.get_inputs_between_states(np_state, state)
            inp_i = model.pick_one_inputs(inp)
            state = model.pre(state, inp_i)
            state_s = model.pick_one_state(state)
            trace.insert(0, inp_i.get_str_values())
            trace.insert(0, state_s.get_str_values())
            reached = reached.union(state)
        state = bad_state
        # add in 'trace' the elements - states and inputs -  from 'bad_state'
to "final" state (e.g. repeated element in a cycle)
        while not(isEmpty(model, state.diff(reached))):
            reached = reached.union(state)
            np_state = model.post(state)
            np_state = np_state.intersection(reachable)
            inp = model.get_inputs_between_states(state, np_state)
            inp_i = model.pick_one_inputs(inp)
            state = model.post(state, inp_i)
            state_s = model.pick_one_state(state)
            trace.append(inp_i.get_str_values())
            trace.append(state_s.get_str_values())
    return res, trace
```

Eventually

Per capire se una proprietà è eventualmente soddisfatta bisogna mostrare che ogni esecuzione del sistema presenta almeno un'occorrenza che verifica la proprietà. Non affermare la formula però risulta più complesso, infatti bisogna dimostrare che nessuna esecuzione contenga almeno un'occorrenza che verifica la proprietà, visto che $\neg \diamond \varphi = \Box \neg \varphi$.

Nell'implementazione sono stati utilizzati tre *bdd* di supporto: `reachable` che contiene gli stati raggiungibili dal sistema, `not_property` che contiene gli stati raggiungibili che non soddisfano la proprietà e `reached` che, man mano che si eseguono le transizioni, contiene gli stati già incontrati, così da terminare la funzione una volta trovato un ciclo nel sistema.

La funzione inizializza una variabile `init` che contiene uno stato iniziale che non soddisfa la proprietà - se questo esiste - e da questo cerca un cammino che contenga solamente stati che non soddisfano la proprietà, così da ottenere un controesempio per $\neg \diamond \varphi$. Nel caso l'intersezione fra l'insieme di stati di una transizione e gli stati di `not_property` sia vuota, la formula di $\diamond\varphi$ risulta soddisfatta e quindi ritorna `True` e una traccia vuota.

```
def check_explain_eventually(spec):
    ltlspec = pynusmv.prop.f(spec)
    print("eventually ltlspec: ", ltlspec)
    model = pynusmv.glob.prop_database().master.bddFsm
    bddspec = spec_to_bdd(model, spec)
    trace = []
    res = True
    reachable = reachable_states(model)
    not_property = reachable.diff(bddspec)
    init = not_property.intersection(model.init)
    if(isEmpty(model, init)):
        return True, []
    res = False
    init = model.pick_one_state(init)
    trace.append(init.get_str_values())
    state = init
    reached = pynusmv.dd.BDD.false()
    # add in 'trace' the elements - states and inputs -  from 'init' to "final"
state (e.g. repeated element in a cycle)
    # return True if a state that satisfy the property is found
    while not(isEmpty(model, state.diff(reached))):
        reached = reached.union(state)
        np_state = model.post(state).intersection(not_property)
        if (isEmpty(model, np_state)):
            return True, []
        inp = model.get_inputs_between_states(state, np_state)
        inp_i = model.pick_one_inputs(inp)
        state = model.post(state, inp_i)
        state_s = model.pick_one_state(state)
        trace.append(inp_i.get_str_values())
        trace.append(state_s.get_str_values())
    return res, trace
```