VU Formale Methoden der Informatik

Block 4: Model Checking

Bernhard Urban

Matr.Nr.: 0725771

lewurm@gmail.com

May 21, 2011

Contents

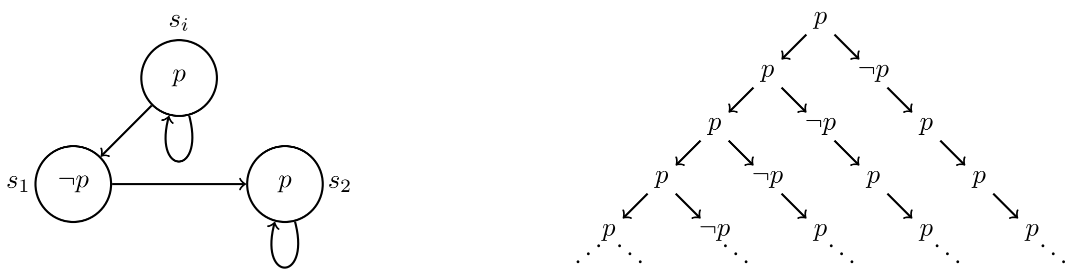


Figure 1: Kripke Model and Infinite Computation Tree

1 Exercise: CTL vs. LTL

Find a Kripke structure K with initial state s such that K has the property $\mathbf{AFG}\,p$ at state s, but not $\mathbf{AFAG}\,p$.

Solution

- $\mathbf{AFG}\,p$ (LTL): "On every path p will be eventually globally true".
- $\mathbf{AFAG}\,p$ (CTL): "On every path p will be eventually *on every path* globally true"

Beginning with s_i in each case. While the depicted kripke structure in Figure 1 is a suitable model for $\mathbf{AFG}\,p$, it isn't for $\mathbf{AFAG}\,p$, as the latter one is more restrictive, i.e. we'll never find a proper point on the left side of the infinite computation tree which can fulfill the expression $\mathbf{AG}\,p$.

2 Exercise: CTL

Show that the temporal operators **AX**, **AF**, **AG**, **AU**, and **EF** occuring in a CTL formula can be replaced by equivalent CTL formulas only using the operators **EX**, **EG**, and **EU**.

Solution

$$\mathbf{AX}\,p \equiv \neg\mathbf{EX}\,\neg p$$

In order to rewrite the expression $\mathbf{AX}\,p$–which means "p holds in the next state on all paths"–we use the negated exists operator and check for the opposite predicate, i.e. $\neg p$.

$$\mathbf{AF}\,p \equiv \neg\mathbf{EG}\,\neg p$$

$\mathbf{AF}\,p$ is equivalent to "p will be eventually true on all paths". We can rewrite this expression using **EG**, e.g. "there exists *no* path on which not p will be globally true".

$$\mathbf{EF}\,p \equiv \mathbf{E}[\mathit{true}\ \mathbf{U} p]$$

"There is a path on which p will be eventually true" can be rewritten as "There exists at least one path on which true holds until p holds".

$$\mathbf{AG}\, p \equiv \neg \mathbf{EF}\, \neg p$$

"p holds globally on all paths" can be rewritten with **EF** as follows "There exists no path on which not p eventually holds".

$$\mathbf{A}\,[p\ \mathbf{U} q] \equiv \mathbf{AF}\, q \wedge \neg \mathbf{E}\,[\neg q\ \mathbf{U} \neg p \wedge \neg q]$$

"p holds until q holds on all paths" can be expressed with a conjunction. The first part can be rewritten as $\neg\mathbf{EG}\, \neg q$ and means "on all paths eventually q holds". The second part means "there exists *no* path where not q holds until not p *and* not q hold".

3 Exercise: CTL Model Checking Algorithm

Give a graph-theoretic algorithm for CTL model checking, i.e., give an algorithm that traverses a Kripke structure $K = (S, T, L)$ until it has determined on which states in S a CTL formula φ holds.

Solution

Instead of writing pseudocode, I decided to develop an actual program with Haskell. The obvious advantage is we're able to *test* the developed code then, therefore the presented code is executable. You can obtain it at `http://wien.tomnetworks.com/fminf/` and try yourself with "The Glasgow Haskell Compiler" (GHC), e.g. `$ ghci 3_check.lhs` starts the interpreter. You can then execute `main`, for example, by just calling `main` in the interpreter.
First, the type definitions and functions are presented. Then, we'll apply some formulas on the kripke structure from Figure 2.

```
  -- © Manfred Schwarz & Bernhard Urban
import Data.List
import Text.Printf
```

3.1 `ctlchecker`: Types

```
data CTL =
  EX CTL | EG CTL | EU CTL CTL
   | AND CTL CTL | NOT CTL | TRUE | FALSE
   | Predicate String
    -- more features
   | AX CTL | AG CTL | AF CTL
```

```
    | EF CTL | AU CTL CTL
    | OR CTL CTL deriving (Show, Eq)
type State = String
type States = [String]
type Transitions = [(State, [State])]
  -- (for future work) TODO: should be replaced with some
  -- propositional logic datastructure!
type Formula = String
type Labels = [(State, Formula)]
type Kripke = (States, Transitions, Labels)
```

3.2 `ctlchecker`: Functions

```
  -- helper functions
appendState :: State → States → States
appendState _ [] = []
appendState s sts = s : sts
nextStep :: CTL → State → States → Kripke → States
nextStep ctl state successors k = concat
  [appendState state (ctlchecker ctl x k) | x ← successors]
  -- actual algorithm. it takes a CTL formula, a init state and
  -- a kripke structure. the function returns a trace of states
ctlchecker :: CTL → State → Kripke → States
ctlchecker (EX ctls) state (s, t, l) =
  case lookup state t of
      -- check all successors
    Just succ → nextStep ctls state succ (s, t, l)
      -- state has no successors, therefore EX can't be fulfilled.
    Nothing → []
ctlchecker (EG ctls) state (s, t, l) =
    -- check if state fulfills the formula
  let def = ctlchecker ctls state (s, t, l) in
  case lookup state t of
    Just succ →
        -- successors for state exists, therefore only
        -- continue when def isn't empty
      if def ≢ [] then
        nextStep (EG ctls) state succ (s, t, l)
      else []
    Nothing → def   -- no successor ⇒ return def
ctlchecker (EU ctla ctlb) state (s, t, l) =
```

```
  let defa = ctlchecker ctla state (s, t, l);
    defb = ctlchecker ctlb state (s, t, l) in
  case lookup state t of
    Just succ →    -- if ctlb is fulfilled, stop here
        if defb ≢ [] then defb else
          if defa ≢ [] then    -- otherwise ctla holds here
            nextStep (EU ctla ctlb) state succ (s, t, l)
          else []   -- else, we have to stop here
    Nothing → defb   -- if no succ. exists, ctlb must be fulfilled

  -- just take the intersection of both sets here
ctlchecker (AND ctla ctlb) state kr =
  ctlchecker ctla state kr `intersect` ctlchecker ctlb state kr

  -- only take the actual state if the formula wasn't fulfilled
ctlchecker (NOT ctls) state kr
  | ctlchecker ctls state kr ≡ [] = [state]
  | otherwise = []

  -- check if the predicate is equal to the actual state's prediacte
  -- (for future work) TODO: replace stringcompare with propositional logic
ctlchecker (Predicate p) state (s, t, l) =
  case lookup state l of
    Just x → if x ≡ p then [state] else []
    Nothing → []

ctlchecker (TRUE) state kr = [state]
ctlchecker (FALSE) state kr = []

  -- more features. replace CTL formulas according the rules in the 2. exercise
ctlchecker (AX ctls) state kr = ctlchecker (NOT (EX (NOT ctls))) state kr
ctlchecker (AG ctls) state kr = ctlchecker (NOT (EF (NOT ctls))) state kr
ctlchecker (AF ctls) state kr = ctlchecker (NOT (EG (NOT ctls))) state kr
ctlchecker (EF ctls) state kr = ctlchecker (EU (TRUE) ctls) state kr

ctlchecker (AU ctla ctlb) state kr =
  ctlchecker (AND
    (NOT (EU (NOT ctlb) (AND (NOT ctla) (NOT ctlb))))
    (NOT (EG (NOT ctlb)))
  ) state kr

ctlchecker (OR ctla ctlb) state kr =
  ctlchecker (NOT (AND (NOT ctla) (NOT ctlb))) state kr
```

3.3 `ctlchecker`: Mini-Testframework

The kripke model is depicted in Figure 2. You can safely skip to the output section now. Thanks for your attention so far! `:-)`

```haskell
s1 :: States
s1 = ["s1", "s2", "s3", "s4", "s5", "s6", "s7", "s8", "s9", "s10"]
t1 :: Transitions
t1 = [("s1", ["s2", "s4"]), ("s2", ["s3"]), ("s4", ["s5", "s6"]),
  ("s5", ["s7"]), ("s6", ["s8"]), ("s7", ["s10"]), ("s8", ["s9"])]
k1 :: Kripke
k1 = (s1, t1, zip s1 ["p1", "p2", "p4", "p2", "p4",  {-5- -} "p4", "p4", "p5", "p5", "p3"])
testfaelle =
    -- (CTL, initstate, expected result)
  [((AF (Predicate "p4")), "s1", ["s1"])
  , ((EF (EG (Predicate "p5"))), "s1", ["s1", "s4", "s6", "s8", "s9"])
  , ((EF (EG (Predicate "p2"))), "s1", [])
  , ((AX (Predicate "p2")), "s1", ["s1"])
  , ((EU (Predicate "p4") (Predicate "p3")), "s5", ["s5", "s7", "s10"])
  , ((AF (EX (Predicate "p4"))), "s1", ["s1"])
  ]
main :: IO ()
main = do
  putStrLn $ "Kripke Structure:"
  putStrLn $ "States:        " ++ (show states)
  putStrLn $ "Transitions:  " ++ (show $ take 4 trans)
  putStrLn $ "               " ++ (show $ drop 4 trans)
  putStrLn $ "Labels:        " ++ (show $ take 5 labels)
  putStrLn $ "               " ++ (show $ drop 5 labels)
  putStrLn $ "Some testcases:"
  sequence_ [
    printTestcase is tc (ctlchecker tc is k1) eres
     | (tc, is, eres) <- testfaelle
    ]
  where (states, trans, labels) = k1
printTestcase initstate tc result expected =
  printf "init: %2s, %36s: %s %s\n"
    initstate (show tc) (show result) check
  where
    check =
      if result == expected then "(OK)"
      else "  (FAIL: " ++ (show result) ++
        ", expected: " ++ (show expected) ++ ")"
```

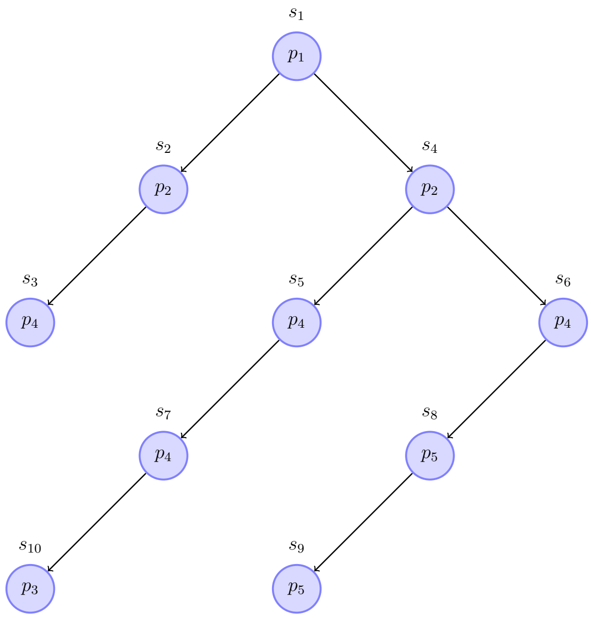


Figure 2: Example Kripke Model

3.4 Output of the Programm

```
Kripke Structure:
States:      ["s1","s2","s3","s4","s5","s6","s7","s8","s9","s10"]
Transitions: [("s1",["s2","s4"]),("s2",["s3"]),("s4",["s5","s6"]),("s5",["s7"])]
             [("s6",["s8"]),("s7",["s10"]),("s8",["s9"])]
Labels:      [("s1","p1"),("s2","p2"),("s3","p4"),("s4","p2"),("s5","p4")]
             [("s6","p4"),("s7","p4"),("s8","p5"),("s9","p5"),("s10","p3")]
Some testcases:
init: s1,                    AF (Predicate "p4"): ["s1"] (OK)
init: s1,               EF (EG (Predicate "p5")): ["s1","s4","s6","s8","s9"] (OK)
init: s1,               EF (EG (Predicate "p2")): [] (OK)
init: s1,                    AX (Predicate "p2"): ["s1"] (OK)
init: s5, EU (Predicate "p4") (Predicate "p3"): ["s5","s7","s10"] (OK)
init: s1,               AF (EX (Predicate "p4")): ["s1"] (OK)
```

4 Exercise: Simulation

Given two models $M_1 = (S_1, I_1, R_1, L_1)$ and $M_2 = (S_2, I_2, R_2, L_2)$, give an algorithm that determines whether M_2 simulates M_1, i.e., whether $M_1 \leqslant M_2$ holds.

Solution

```
-- © Manfred Schwarz & Bernhard Urban
import Data.List
import Data.Maybe
```

4.1 `sim`: Types

Again, we need some representation for kripke structures, this time with initiale states. Therefore we call it `Model`.

```
type Model = (States, Init, Transitions, Labels)
type State = String
type States = [State]
type Init = States
type Transitions = [(State, [State])]
type Labels = [(State, String)]
type Relation = (State, State)
```

4.2 `sim`: Functions

This function takes M_1 and M_2. It returns a tuple, where the boolean tells us if M_2 simulates M_1. the relation represents the H:

```
sim :: Model → Model → (Bool, [Relation])
sim m1@(_, i1, _, _) m2@(_, i2, _, _) = (if res ≡ [] then False else and res, h)
  where
    h = sim_genH m1 m2   -- calculate H
```

check if: "for every $s_1 \in I_1$ there is $s_2 \in I_2$ s.t. $(s_1, s_2) \in H$"

```
    res = [s2 ∈ i2 | s1 ← i1, s2 ← f s1 h]
    f :: State → [Relation] → [State]   -- helper function. doesn't deserve a name
    f _ [] = []
    f s ((s1, s2) : rs) = if s ≡ s1 then s2 : (f s rs) else f s rs
sim_genH :: Model → Model → [Relation]
sim_genH m1@(ss1, _, _, l1) m2@(ss2, _, _, l2) =
  catMaybes [sim_check m1 m2 (h : hs) [] x | x ← (h : hs)]
  where   -- generate a cross product of all states of both models with the same predicates
    (h : hs) = [(a1, b2) | a1 ← ss1, b2 ← ss2
      , let ap1 = fromJust $ lookup a1 l1
      , let bp2 = fromJust $ lookup b2 l2
```

```
        , ap1 ≡ bp2]
sim_check :: Model → Model → [Relation] → [Relation] → (State, State) → Maybe Relation
sim_check m1@(ss1, i1, r1, l1) m2@(ss2, i2, r2, l2) hr visited (s1, s2) =
  case lookup s1 r1 of
    Just t1s → if for_each_t1_a_t2_exists t1s then Just (s1, s2) else Nothing
    Nothing → Nothing
  where
    for_each_t1_a_t2_exists :: States → Bool
    for_each_t1_a_t2_exists t1s = and
      [or
        [case lookup t1 r1 of    -- t1 and t2 have no succ.
           Just _ → False
           Nothing → case lookup t2 r2 of
             Just _ → False; Nothing → True
         ∨ (t1, t2) ∈ visited   -- or: already visited?
      -- Otherwise, check if for (t1, t2) also hold (attention, variable renaming...):
      --    ∀o1[(t1, o1) ∈ R1 ⇒ ∃o2[(t2, o2) ∈ R2 ∧ (o1, o2) ∈ H]]
         ∨ case sim_check m1 m2 hr ((t1, t2) : visited) (t1, t2) of
           Just x → (t1, t2) ≡ x; Nothing → False
         | t2 ← case lookup s2 r2 of Just x → x; Nothing → []
        , (t1, t2) ∈ hr]   -- do this only for tuples, which are
       | t1 ← t1s]
```

4.3 sim: Mini-Testframework

Again, you can skip this part.

```
main = do
  putStrLn "see page 4 on the slides \"Abstraction\", I <= S:"
  printTestCase (sim m1 m2)
  putStrLn "see page 4 on the slides \"Abstraction\", I >= S:"
  printTestCase (sim m2 m1)
  putStrLn ""
  putStrLn "M1 <= M2 from exercise 5:"
  printTestCase (sim m51 m52)
  putStrLn "M1 >= M2 from exercise 5:"
  printTestCase (sim m52 m51)
printTestCase (res, h) = putStr $
  "\tit is" ++ ismodel ++ " a model. " ++
  "the relation H is\n\t" ++
  (show $ take 5 h) ++ "\n" ++ reminder
  where
    ismodel = if res then "" else " NOT"
```

```
    reminder = if (length h) > 5 then
      "\t" ++ (show $ drop 5 h) ++ "\n" else ""

  -- testcases
states1, states2 :: States
states1 = ["s1", "s2", "s3", "s4", "s5"]
states2 = ["s1'", "s2'", "s3'", "s4'"]

m1, m2 :: Model
m1 = (states1, ["s1"],
  [("s1", ["s2"]), ("s2", ["s3"]), ("s1", ["s4"]), ("s4", ["s5"]),
    ("s3", ["s3"]), ("s5", ["s5"])],
  zip states1 ["r", "g", "b", "g", "o"])

m2 = (states2, ["s1'"],
  [("s1'", ["s2'"]), ("s2'", ["s3'", "s4'"]), ("s3'", ["s3'"]), ("s4'", ["s4'"])],
  zip states2 ["r", "g", "b", "o"])

m5s1, m5s2 :: States
m5s1 = ["s1", "s2", "s3", "s4", "s5"]
m5s2 = ["s1'", "s2'", "s3'", "s4'", "s5'", "s6'", "s7'"]

m51, m52 :: Model
m51 = (m5s1, ["s1"],
  [("s1", ["s2", "s3"]), ("s2", ["s2"]), ("s3", ["s1", "s5", "s4"]),
    ("s4", ["s4"]), ("s5", ["s4"])],
  zip m5s1 ["a", "d", "b", "d", "c"])

m52 = (m5s2, ["s1'"],
  [("s1'", ["s4'", "s2'", "s3'"]),
    ("s2'", ["s1'", "s3'", "s5'"]),
    ("s3'", ["s6'"]),
    ("s4'", ["s1'", "s6'", "s7'"]),
    ("s5'", ["s6'"]),
    ("s6'", ["s6'"]),
    ("s7'", ["s6'"])],
  zip m5s2 ["a", "b", "d", "b", "c", "d", "c"])
```

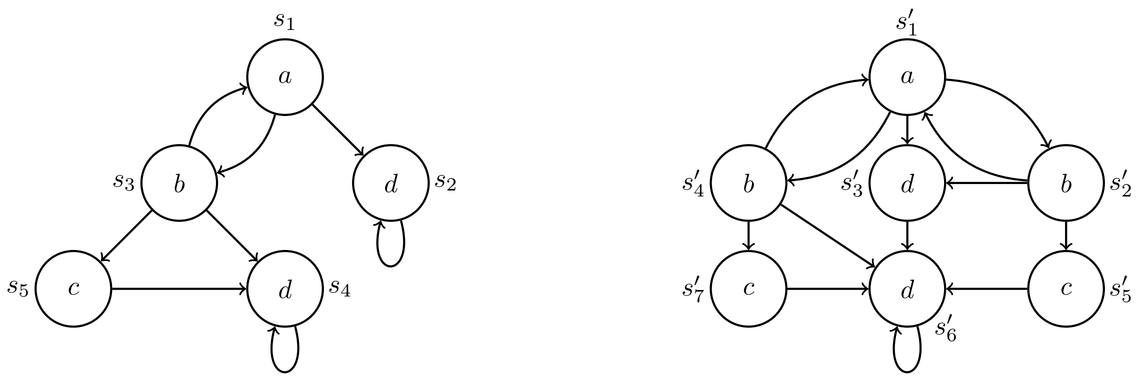
4.4 Output of the Programm

```
see page 4 on the slides "Abstraction", I <= S:
it is a model. the relation H is
[("s1","s1'"),("s2","s2'"),("s3","s3'"),("s4","s2'"),("s5","s4'")]
see page 4 on the slides "Abstraction", I >= S:
it is NOT a model. the relation H is
[("s3'","s3"),("s4'","s5")]
```

```
M1 <= M2 from exercise 5:
it is a model. the relation H is
[("s1","s1’"),("s2","s3’"),("s2","s6’"),("s3","s2’"),("s3","s4’")]
[("s4","s3’"),("s4","s6’"),("s5","s5’"),("s5","s7’")]
M1 >= M2 from exercise 5:
it is a model. the relation H is
[("s1’","s1"),("s2’","s3"),("s3’","s2"),("s3’","s4"),("s4’","s3")]
[("s5’","s5"),("s6’","s2"),("s6’","s4"),("s7’","s5")]
```

5 Exercise: Bisimulation

Give a bisimulation relation for the following two Kripke structures:



Solution

See Section 4.4. Although it’s the correct result, we didn’t gained it 100% correctly, since the algorithm from the section above just checks for simulation and not for bisimultion. However, as the algorithm doesn’t determine the minimal set of H, it produces the right result *for this example.*

6 Exercise: Abstraction

Given the following program:

```
int p, q, x, y;
void foo() {
  p = 0; q = 0;
  while (x > 0) {
    y = x;
    if (y == 0) {
      p = 1;
    }
```

```
    x = x - 1;
  }
  if (p != 0) {
    assert(0); // ERROR
  }
}
```

a) Provide a labeled transition system for the given program.

b) Provide an abstraction for the labeled transition system that uses the predicates $(p = 0)$, $(q = 0)$, and $(x > 0)$.

c) Show manually, that the error state is reachable in the abstraction.

d) Refine the abstraction with a suitable predicate to get rid of the error state.

Solution

(a) See Figure 3.

(b) $p1 = p == 0$, $p2 = q == 0$ and $p3 = x > 0$. The abstraction is depicted in Figure 4. Note that I mirrored the third state, in order to provide a better overview.

(c) The red arrows in Figure 4 are one example for a spurious trace, since this state isn't reachable in the original program.

(d) $y > 0$ would be a suitable predicate. Although the "evil states" (6 and 7) still exist then, the won't be reachable anymore (cf. state 5 and 6).

7 Exercise: CBMC

Use CBMC to solve the Hamilton path decision problem for a given graph, i.e., write a C program that

1. initializes the representation of the graph, e.g., a two-dimensional array that encodes the transition matrix of the graph,
2. guesses a path through the graph,
3. and checks whether the path is a Hamilton path.

Note, you can implement the guessing step by initializing the elements of the path with nondeterministic values, CBMC will then derive suitable values in case a Hamilton path exists. Write your program such that CBMC reports an assertion error in case a Hamilton path exists.

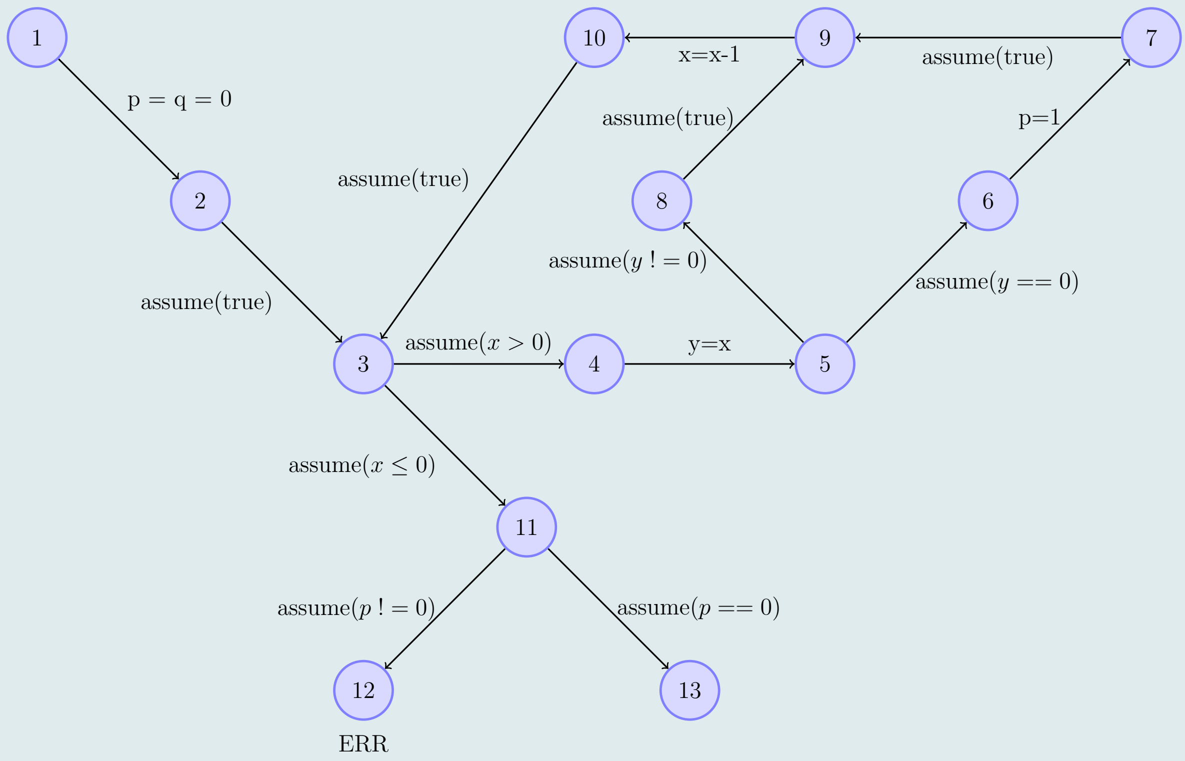


Figure 3: Labeld Transition System of the given Programm

Solution

```c
#define N 12

int nondet_int();

void f() {
    // adjacency matrix
    int graph[N][N];
    // hamilton path (one cell for each node)
    int path[N];
    int i, j, t, valid;

    // initialize graph. connect all nodes to each other,
    // so a hamilton cycle must exists by construction
    for(i = 0; i < N; i++) {
        for(j = 0; j < N; j++) {
            graph[i][j] = 1;
        }
    }

```

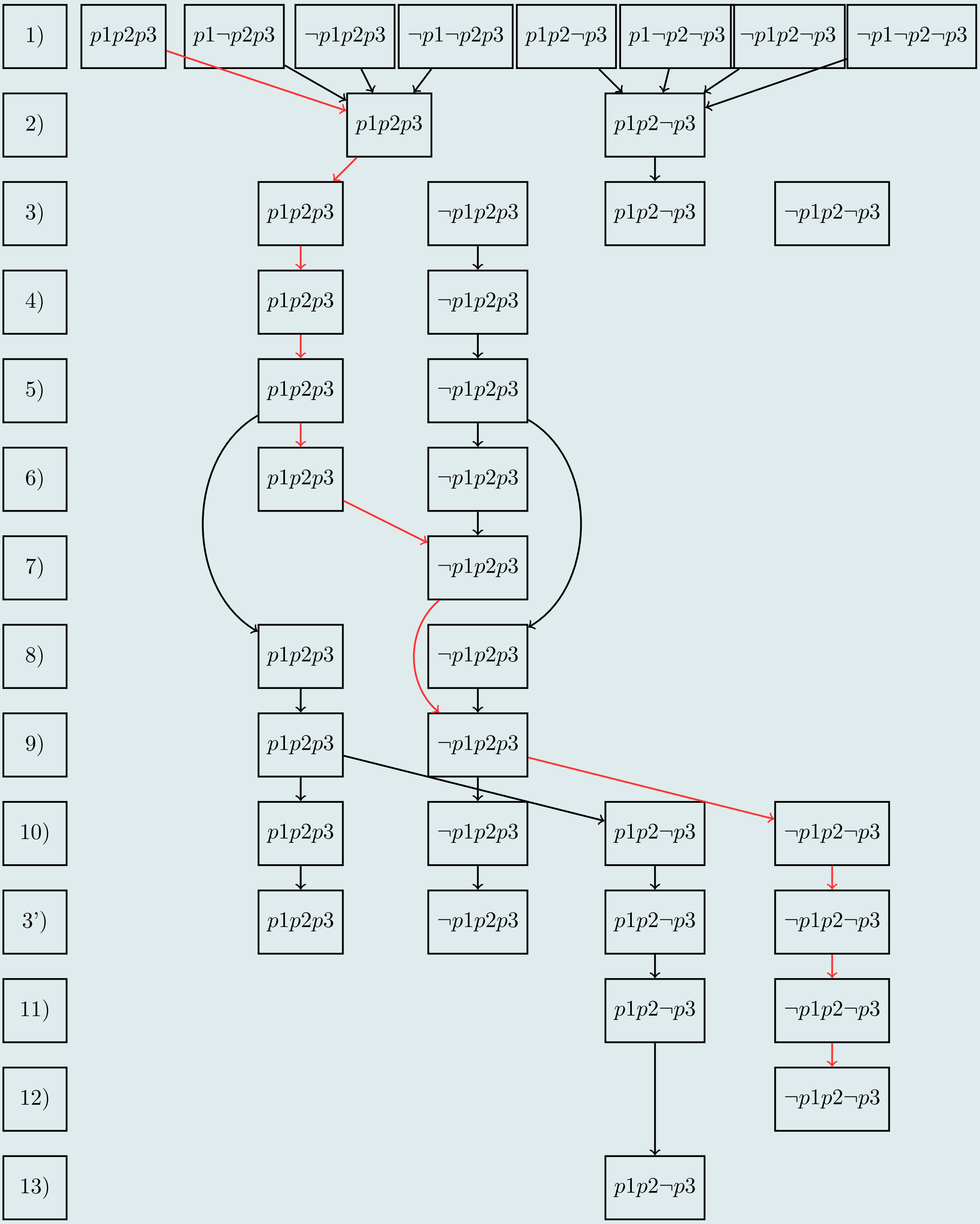


Figure 4: Abstraction for the given Program

```
    // guess some path, with proper assumptions
    for(i = 0; i < N; i++) {
        path[i] = nondet_int();
        __CPROVER_assume(path[i] >= 0 && path[i] < N);
    }

    // check if the choosen path is really a hamilton one.
    // simply check if a node occurs more than once
    valid = 1;
    for(i = 0; i < N; i++) {
        t = 0;
        for(j = 0; j < N; j++) {
            if(i == path[j])
                t++;
        }
        if(t != 1) {
            valid = 0;
        }
    }

    // check if there exists an edge for each step in the path
    for(i = 0; i < N-1; i++) {
        if(graph[path[i]][path[i+1]] == 0) {
            valid = 0;
        }
    }

    // check if "not valid" is correct. counterexample plzkkthx
    __CPROVER_assert(!valid, "w00t. found hamilton cycle");
}

/* Output by cbmc:
[...]
 7_ham::f::1::path={ 2, 3, 7, 5, 1, 6, 10, 9, 4, 8, 0, 11 }
[...]
Violated property:
  file 7_ham.c line 48 function f
  w00t. found hamilton cycle
  !(_Bool)valid

VERIFICATION FAILED
 */
```