# Zusammenfassung: Programm- und Systemverifikation

---

**An dieser Zusammenfassung und der zugehörigen Formelsammlung kann gerne auf Github mitgewirkt werden!**

---

## Contents

- Fault: cause of an error
- Error: erroneous state, but not directly observable in behaviour → might lead to failure, but not necessarily
- Failure: deviation from expected behaviour

# Coverage

Coverage criteria state when enough testing has been done.

## Control Flow Based Coverage Criteria

### Path Coverage

Execute every path the program could take at least once.

Easy counter example to see that path coverage has not been reached are loops: every new loop iteration constitutes a new path and all paths have to be taken. Path coverage is generally not always reachable, e.g., it es not achievable for the following program:

```
1 while (1) {
2     if ( getchar () == EOF )
3     break ;
4 }
```

### Statement Coverage

Execute every statement of the program (merely syntactic) at least once.

Statement Coverage is implied by path coverage. Hence, if statement coverage can't be achieved, path coverage can't be achieved either. On the other hand, if path coverage can't be achieved for a given program, statement coverage still can be reached, as is the case in above program.

If for a given program statement coverage can't be achieved, it is said to contain unreachable code:

```
1 if (false){
2     ...
3 }
```

### Branch Coverage

Execute every branch at least once.

In literature, the definitions of branches are rather imprecise → what about unconditional jumps, `goto`, function calls or fall-throughs?

### Decision Coverage

Exercise every decision outcome at least once (one time `true`, one time `false`)

Again, definition of decisions is imprecise.

### Notes on Branch Coverage vs. Decision Coverage

Branch coverage implies decision coverage

- if "decision" means boolean expression at branching points only

Decision coverage implies branch coverage

- if "branch" doesn't include unconditional jumps
- if "decision" refers to *all* boolean expressions

### Condition Coverage

Exercise every boolean sub-expression/atom/condition outcome (but their values do not necessarily have to affect the overall outcome)

Condition coverage does not imply descision coverage, as can be seen by the following program, with the test cases $\{x = 5, y = -3\}$ and $\{x = -1, y = 2\}$

```
1 if ( ( x > 0) && ( y > 0) )
2     x++;
```

All outcomes of the sub-expressions are exercised once but the decision never evaluated to `true`.

### Condition/Decision Coverage

Combination of condition and decision coverage:

- cover all condition outcomes
- cover all decision outcomes

but not all branches of the 'decision tree' might be executed

### Modified Condition / Decision Coverage (MC/DC)

Every condition in a decision has to have taken affect on the outcome (independently) at least once. (remember the stuck-at error model in the lecture on digital design)

For example, see the exam of 2018, task on coverage, subtask D

MC/DC is defined in DO-178B (high relevance in industry)

### Multiple Condition Coverage

For $n$ sub-conditions in a decision, try all $2^n$ combinations.

## Data Flow Based Coverage Criteria

- Definitions: assignment of a value to a variable
- Use: statement where the value of a variable is read
  - C(omputation)-Use: defines/computes other variables
  - P(redicate)-Use: within conditional statements

Table 1: Data Flow Criteria

| Name | Criteria |
|---|---|
| all-defs | all definitions get used at some point |
| all-c-uses | one path from a definition to each c-use that is affected by that definition |
| all-p-uses | one path from a definition to each p-use that is affected by that definition |
| all-c-uses/some-p-uses | same as all-c-uses, but if there are no c-uses, than at least one affected p-use needs to be triggered |
| all-p-uses/some-c-uses | same as all-p-uses, but if there are no c-uses, than at least one affected c-use needs to be triggered |
| all-uses | all-c-uses and all-p-uses → all uses need to be executed |
| all-du-paths | same as all-uses, but all possible du paths have to be taken at least once, not just one path |

## Mutation Testing

Aim: test how well a test suite is capable of finding bugs in software

Idea: create mutation of that software by deliberately injecting a bug → check

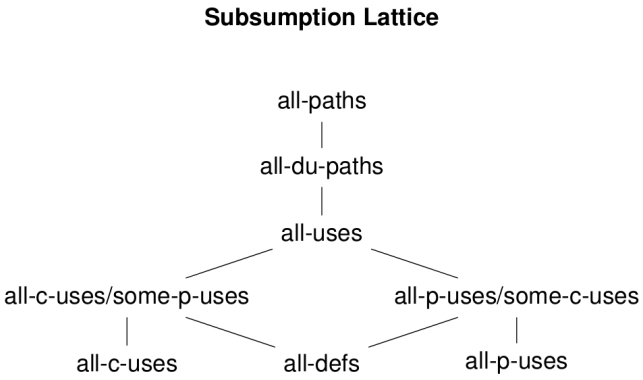


Figure 1: subsumption lattice

whether test suite "kills" mutant

or the other way around: create mutant $P_2$ of program $P_1$, encode behaviour of $P_1$ and $P_2$ in formula and use SAT solver to see whether $P_1 \oplus P_2$ is satisfiably $\rightarrow$ if yes, then satisfying assignment represents a testcase that kills that mutant; if no, then the bug can't be found

# Automated Test Case Generation

## Model based test case generation

General scheme:

1. develop an (abstract) model of the system.
2. automatically derive abstract test cases from the abstract model
3. map the abstract test cases to concrete ones
4. apply the concrete test cases to the implementation

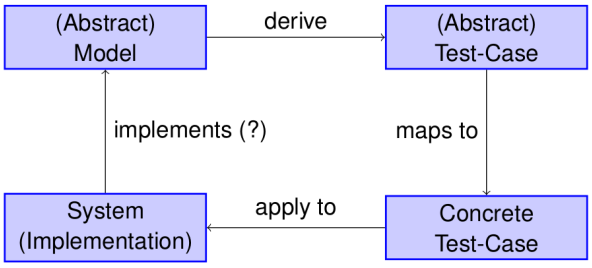


Figure 2: procedure for model based test case generation

Don't:

- extract test-cases from implementation

- apply test-cases extracted from model to generated code
- let coverage criteria drive your test-case generation

### Assertion Violations

Idea: try to find inputs that crash the system → no need to check output for correctness

Assertions can be used to express partial specifications:

- buffer overflow: `assert (i < len); ... a[i]`
- division by zero: `assert (y != 0); ... x/y`
- invalid pointer: `assert (p != NULL); ... *p`
- assertions for further specification

utilize SMT solvers to find such crashing inputs:

1. perform *symbolic* execution of a path → derive SMT formula representing current program state
2. at any assertion, ask an SMT solver for input values that lead to assertion failure

Naive exploration of paths quickly becomes a problem (search space explosion) → heuristics:

- breadt-first search
- depth-first search
- coverage-optimized (take that path that increases coverage the most)
- random selection

If we can check for assertion violations, we can check contracts.

### Oracle

Alternatively, we can ask an oracle for correct output.

The oracle could be:

- a less efficient (but correct) implementation
- an executable specification
- ...

## Model Checking

### Bounded Model Checking

We wish, we could automate Haore reasoning, but finding loop invariants can't be generated automatically. Work-around: we restrict ourselves to a finite amount of loop iteration → bounded model checking

"Forwards with Hoare": calculate stronges post-condition to a given pre-condition and a statement $sp(stmt, P)$

Table 2: Rules for "Forwards with Hoare" (stronges post-condition)

| stmt | $sp(stmt, P)$ |
|---|---|
| `x := e;` | $\exists x' : x = e[x/x'] \wedge P[x/x']$ |
| `assert R;` | $P \wedge R$ |
| `stmt1; stmt2;` | $sp(stmt_2, sp(stmt_1, P))$ |
| `ite(B, C1, C2);` | $sp(C_1, B \wedge P) \vee sp(C_2, \neg B \wedge P)$ |

The last rule in the table is interesting: it merges two paths → useful to avoid exponential blow up through loop unwinding.

**Loop unwinding**: Unwindig a loop for a given amount of iterations. In last iteration, insert so called unwinding assertion, which lets us know if more iterations were possible.

The program fragment `while(B){ BODY }` can be unwound to the following program:

```
if (B){
    BODY
    if (B){
        BODY
        .
         .
          .
            if (B){
                assert false; // unwinding assertion
            }
          .
         .
        .
    }
}
```

### Unbounded Model Checking

Uses model in form of a Kripke structure to check statements in temporal logic (see chapter on temporal logic)

Finding out for which states a given formula in temporal logic (either CTL or LTL) holds:

1. split up formula into subformulas (tree-like)
2. recursively find states for which subformulas hold → begin at leaves (propositional formulas) and work your way to the root

## Temporal Logic

### CTL*

Table 3: Syntax and semantics for temporal operators

| Syntax | Semantics |
|---|---|
| **X** *path formula* | $M, \pi \models \mathbf{X}\phi$ iff $M, \pi^1 \models \phi$ |
| **F** *path formula* | $M, \pi \models \mathbf{F}\phi$ iff $\exists k \in \mathbb{N} \cup \{0\} : M, \pi^k \models \phi$ |
| **G** *path formula* | $M, \pi \models \mathbf{F}\phi$ iff $\forall k \in \mathbb{N} \cup \{0\} : M, \pi^k \models \phi$ |
| *path formula* **U** *path formula* | $M, \pi \models \phi_1 \mathbf{U} \phi_2$ iff<br>$\exists k \in \mathbb{N} \cup \{0\} : M, \pi^k \models \phi_2 \ \wedge$<br>$\forall j, 0 \leq j < k : M, \pi^j \models \phi_1$ |
| *path formula* **R** *path formula* | //todo |

Table 4: Syntax and semantics for path quantifiers

| Syntax | Semantics |
|---|---|
| **A** *path formula* | $M, s \models \mathbf{A}\phi$ iff $\forall \pi$ starting at $s$:$M, \pi \models \phi$ |
| **E** *path formula* | $M, s \models \mathbf{E}\phi$ iff $\exists \pi$ starting at $s$:$M, \pi \models \phi$ |

## Subsets of CTL*

**CTL**: Like CTL* but every temporal operator has to be preceded immediately by a path quantifier

**LTL**: The formula has to start with the **A**-operator but apart from that, no path quantifiers are allowed. (Usually the preceding **A** can be omitted)

There are CTL formulas that can't be expresed in LTL and vice versa

CTL has 10 basic operators – 5 temporal operators times 2 path quantifiers – but all of them can be expressed through **EX**, **EG** and **EU**

$$\mathbf{AX}\varphi \equiv \neg\mathbf{EX}(\neg\varphi) \qquad \mathbf{EF}\varphi \equiv \mathbf{E}(\mathsf{true}\,\mathbf{U}\,\varphi)$$

$$\mathbf{AG}\varphi \equiv \neg\mathbf{EF}(\neg\varphi) \qquad \mathbf{AF}\varphi \equiv \neg\mathbf{EG}(\neg\varphi)$$

$$\mathbf{A}(\varphi_1\,\mathbf{R}\,\varphi_2) \equiv \neg\mathbf{E}(\neg\varphi_1\,\mathbf{U}\,\neg\varphi_2) \qquad \mathbf{E}(\varphi_1\,\mathbf{R}\,\varphi_2) \equiv \neg\mathbf{A}(\neg\varphi_1\,\mathbf{U}\,\neg\varphi_2)$$

$$\mathbf{A}(\varphi_1\,\mathbf{U}\,\varphi_2) \equiv \neg\mathbf{E}(\neg\varphi_2\,\mathbf{U}\,(\neg\varphi_1 \wedge \neg\varphi_2)) \wedge \neg\mathbf{EG}\neg\varphi_2$$

Figure 3: expressing the remaining 7 basic operators of CTL through **EX**, **EG** and **EU**

# SAT

## Tseitins Transformation

Goal: come up with CNF formula that is equisatisfiable to a given propositional logic formula

1. express formula only through conjunctions and disjunctions ($a \Rightarrow b \equiv \neg a \vee b$ and so forth)
2. build syntax tree of formula, introducing new variables for every subformula
3. build CNF formula by expressing each subtree through three CNF clauses, bottom-up (see illustration)

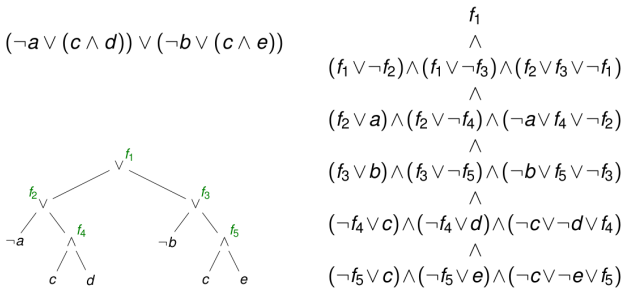


Figure 4: Tseitin Transformation: build CNF formula from tree

## Resolution Rules

$$\frac{(C \vee a) \quad (D \vee \overline{a})}{(C \vee D)}$$

and in particular

$$\frac{(C \vee a) \quad \overline{a}}{C} \text{ unit propagation}$$

## Decision making

whenever possible, propagate units, but what if there ware no units to propagate? $\rightarrow$ make a decision for one variable (i.e. assignment)

What if decision a decision leads to a conflict? *rightarrow* backtracking: determine a "learnd clause" and return to highest decision level that is not contained in conflict clause (or to 0)

How to find good conflict clause? Choose conflict clause such that it contains the first unique implication point (UIP), i.d., a node (other than the conflict node) that lies on all paths from the decision node to the conflict node and is closest to the conflict node. (the decision node is a UIP by definition)

DPLL algorithm:

1. if conflict at decision level 0 $\rightarrow$ UNSAT
2. repeat:

(a) if all variables assigned, return SAT
(b) make decision
(c) propagate constraints
(d) if no conflict, goto 1.
(e) if decision level is 0, return UNSAT
(f) analyse conflict and add conflict clause
(g) backtrack and go to 3.

Like with BDDs, variable order makes difference. How to choose which variable to assign next if a decision has to be made? Heuristics:

- Dynamic largest individual sum (DLIS): choose assignment such that number of satisfied clauses is maximised (high overhead)
- Variable state independent decaying sum (VSDIS): favour literals in recently added conflict clauses. With right data structures, decision is possible in $\mathcal{O}(1)$

Table 5: comparison of BDDs and SAT-solvers

| | BDDs | SAT solvers |
|---|---|---|
| #(variables) | hundreds | hundreds of thousands |
| complexity | PSPACE-complete | NP-complete |
| assignments | $\mathcal{O}(n)$ | SAT-run |
| canoncial | yes | no |
| equality check | $\mathcal{O}(1)$ | SAT-run of $F \oplus G$ |
| quantifier elimination | yes | co-factoring |

# Examples

Alle Angaben ohne Gewähr. Etwaige Fehler bitte auf Github anmerken/ändern.

## Coverage

### Example 1, taken from the exam in June 2016

Consider the following program fragment and test suite:

```
1  int maxsum (int max, int val){
2      int result = 0;
3      int i = 0;                                  //       Test Suite
4      if (val < 0)                                //  --------------------
5          val = -val;                             //   max   val   result
6      while ((i < val) && (result <= max)){       //  ----- ----- --------
7          i = i+1;                                //    0     0      0
8          result = result + i;                    //    0    -1      0
9      }                                           //   10     1      1
10     if (result <= max)                          //  --------------------
11         return result;
12     else
13         return max;
14 }
```

### A) Control flow based criteria

Indicate (X) which of the following coverage criteria are satisfied by the test-suite above (assume that the term "decision" refers to all non-constant Boolean expressions in theprogram).

| Criterion | Satisfied | Not Satisfied |
|---|---|---|
| path coverage | | X |
| statement coverage | X | |
| branch coverage | X | |
| decision coverage | X | |
| condition/decision coverage | X | |

### B) Data flow based criteria

Indicate (X) which of the following coverage criteria are satisfied by the test-suite above(here, the parameters of the function do not constitute definitions, and the return statements are c-uses)

| Criterion | Satisfied | Not Satisfied |
|---|---|---|
| all-defs | X | |
| all-c-uses | X | |
| all-p-uses | X | |
| all-c-uses/some-p-uses | X | |

| Criterion | Satisfied | Not Satisfied |
|---|---|---|
| all-p-uses/some-c-uses | X | |

**Example 2, taken from the exam in June 2017**

Consider the following program fragment and test suite:

```
1  bool prime (unsigned n){
2      bool result = true;              //    Test Suite
3      unsigned i = 2;                  //  ---------------
4      while ((i != n) && result){      //   n      result
5          if (n % i == 0)              //  ----    ------
6              result = false;          //   0      false
7          else                         //   3      true
8              i = i + 1;               //   42     false
9      }                                //  ---------------
10     return result;
11 }
```

### A) Control flow based criteria

Indicate (X) which of the following coverage criteria are satisfied by the test-suite above (assume that the term "decision" refers to all non-constant Boolean expressions in theprogram).

| Criterion | Satisfied | Not Satisfied |
|---|---|---|
| path coverage | | X |
| statement coverage | X | |
| branch coverage | X | |
| decision coverage | X | |
| condition/decision coverage | X | |

### B) Data flow based criteria

Indicate (X) which of the following coverage criteria are satisfied by the test-suite above(here, the parameters of the function do not constitute definitions, and the return statements are c-uses)

| Criterion | Satisfied | Not Satisfied |
|---|---|---|
| all-defs | X | |
| all-c-uses | | X |
| all-p-uses | X | |
| all-c-uses/some-p-uses | | X |
| all-p-uses/some-c-uses | X | |

**Example 3, taken from the exam in June 2018**

Consider the following program fragment and test suite

```
1  bool range_check (unsigned m, unsigned n){
2      if (m > n){
3          unsigned t = m;                    //    Test Suite
4          m = n;                             //  ---------------
5          n = t;                             //   m   n   result
6      }                                      //  --- ---  ------
7      bool result = false;                   //   3   7   true
8      bool tmp = true;                       //   1   0   false
9      unsigned i = m;                        //   2   5   true
10     while ((i > 0) && (i < n)){            //  ---------------
11         i = i + 1;
12         if (i % m == 0)
13             result = result || tmp;
14     }
15     return result;
16 }
```

### A) Control flow based criteria

Indicate (X) which of the following coverage criteria are satisfied by the test-suite above (assume that the term "decision" refers to all non-constant Boolean expressions in theprogram).

| Criterion | Satisfied | Not Satisfied |
|---|---|---|
| statement coverage | X | |
| branch coverage | X | |
| decision coverage | X | |
| modified condition/decision coverage | ? | |

### B) Data flow based criteria

Indicate (X) which of the following coverage criteria are satisfied by the test-suite above (here, the parameters of the function do not constitute definitions, and the return statements are c-uses).

| Criterion | Satisfied | Not Satisfied |
|---|---|---|
| all-defs | X | |
| all-c-uses | | X |
| all-p-uses | X | |
| all-c-uses/some-p-uses | | X |
| all-p-uses/some-c-uses | | X |

### C) not given here

### D) MC/DC {#2018_mcdc}

Consider the expression $((a \wedge b) \vee c)$, where $a$, $b$, and $c$ are Boolean variables. Provide a minimal number of test cases such that modified condition/decision coverage is achieved for the expression. Clarify for each test case whichcondition(s) independently affect(s)the outcome.

| a | b | c | (a && b) \|\| c |
|---|---|---|---|
| 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |

**Example 4, taken from the exam in June 2020**

```
1  unsigned gcd (unsigned x, unsigned y) {
2      unsigned m, k;       //     Test Suite
3      if (x > y) {         //  ----------------
4          k = x;           //   x   y   return
5          m = y;           //  --- --- --------
6      } else {             //   0   0   0
7          k = y;           //   0   1   0
8          m = x;           //   3   2   1
9      }                    //  ----------------
10     while (m != 0) {
11         unsigned r = m % k;
12         k = m;
13         m = r;
14     }
15     return k;
16 }
```

### A) Control flow based criteria

Indicate (X) which of the following coverage criteria are satisfied by the test-suite above.

| Criterion | Satisfied | Not Satisfied |
|---:|:---:|:---:|
| path coverage | | X |
| statement coverage | X | |
| branch coverage | X | |
| decision coverage | X | |

### B) Data flow based criteria

Indicate (X) which of the following coverage criteria are satisfied by the test-suite above (here, the parameters of the function do not constitute definitions, and the return statements are c-uses).

| Criterion | Satisfied | Explanation |
|---:|:---:|---|
| all-defs | yes | |
| all-c-uses | no | No path from line 7 to 11 |
| all-p-uses | yes | |
| all-c-uses/some-p-uses | no | all-c-uses not satisfied |
| all-du-paths | no | all-c-uses/some-p-uses not satisfied |

## Hoare-Logic

### Example 1, taken from the exam in June 2016

Prove the Hoare Triple below (assume that the domain of all variables in the programare the natural numbers including 0, i.e., $x, y \in \mathbb{N}_0$ or, equivalently, both $x$ and $y$ are of type `unsigned`). You need to find a sufficiently strong loop invariant. Annotate the following code directly with the required assertions. Justify each assertion by stating which Hoare rule you used to derive it, and the premise(s) of that rule. Ifyou strengthen or weaken conditions, explain your reasoning

```
{true}

assert true; // if-then-else-rule

if (x > y){
    assert x > y && true; // strengthening
    assert y <= x+1; // assignment rule
    t := x;
    assert y <= t+1; // assignment rule
    x := y;
    assert x <= t+1; // assignment rule
    y := t;
    assert x <= y+1; // if-then-else rule
}
else{
    assert !(x > y) && true; // non-existing assignment + strenthening
    skip;
    assert x <= y+1; // if-then-else rule
}


assert x <= y+1; // loop rule

while (x < y){
    assert x < y; // strengthening (actually, it's equivalent),
                  // also it's implied by loop condition => induction step
    assert x+1 <= y; // assignment rule
    x := x + 1;
    assert x <= y; // assignment rule
    y := y - 1;
    assert x <= y+1; // invariant
}
assert !(x < y) && x <= y+1; // loop rule
assert (x-y) <= 1; // weakening (actually, it's the same)
{x-y<=1}
```

**Example 2, taken from the exam in June 2018**

Prove the Hoare Triple below (assume that the domain of all variables in the program are the unsigned integers including zero, i.e., $x, y, n, m \in \mathbb{N} \cup \{0\}$). You need to find a sufficiently strong loop invariant. *Hint*: you will need an expression that represents how often the loop has been executed.

```
{true}
assert true; // strengthening
assert 0 == 0 && n == n; // assignment rule
x := n;
assert 0 == 0 && x == n; // assignment rule
y := 0;
assert y == 0 && x == n; // if-then-else rule
if (m != 0){
    assert m != 0 && y == 0 && x == n; // strengthening
    assert y == (n-x)*m; // loop rule
    while (x != 0){
        assert y+m == (n-x+1)*m; // assignment rule,
                                 // implied by invariant => inductiveness
        x = x - 1;
        assert y+m == (n-x)*m; // assignment rule
        y = y + m;
        assert y == (n-x)*m; // invariant
    }
    assert x == 0 && y == (n-x)*m; // strengthening / loop rule
    assert y == n*m; // if-then-else rule
}
else{
    assert m == 0 && y == 0 && x == n; // strengthening
    skip;
    assert y == n*m; // if-then-else rule
}

assert y == n*m;
{y=n*m}
```

**Example 3, taken from the exam in June 2019**

Prove the Hoare Triple below (assume that the domain of all variables except done in the program are the unsigned integers including zero, i.e., $i, m, n \in \mathbb{N} \cup \{0\}$, and that `done` is a Boolean variable). You need to find a sufficiently strong loop invariant.

```
{true}
assert true; // if-then-else rule
if (m > n){
    assert m > n && true; // assignment rule and strengthening
    i = n;
    assert true; // if-then-else rule
}
else{
    assert m <= n && true; // assignment rule and strengthening
    i = m;
    assert true; // if-then-else rule
}

assert true; // strengthening
assert true || m%(i-1) == 0; // assignment rule
done = false;

assert (!done || m%(i-1) == 0); // loop rule

while ((i > 1) && !done) {
    assert !done || m%(i-1) == 0; // if-then-else rule,
                                  // implied by loop condition => induction step
    if ((m % i == 0) && (n % i == 0)){
        assert (m % i == 0) && (n % i == 0) && (!done || m%(i-1) == 0); // strengthening
        assert false || m%i == 0; // assignment rule
        done = true;
        assert !done || m%i == 0; // if-then-else rule
    }
    else{
        assert (m%i != 0 || n%i != 0) && (!done || m%(i-1) == 0); // assignment rule and s
        i = i - 1;
        assert !done || m%i == 0; // if-then-else rule
    }
    assert !done || m%i == 0; // invariant
}
assert (i <= 1 || done) && (!done || m%i == 0); // loop rule / strengthening
                                                // proof by case splitting
assert i == 0 || m%i == 0;
{(i = 0) || (m % i = 0)}
```

**Example 4, taken from the exam in June 2020**

Prove the Hoare Triple below (assume that the domain of all variables in the program are the integers, i.e., $t, m, n \in \mathbb{Z}$. You need to find a sufficiently strong loop invariant.

```
1  {true}
2  assert true; // if-then-else rule
3  if (m > n) {
4      assert m > n && true; // strenthening
5      assert n <= m; // assignment rule
6      int t = n;
7      assert t <= m; // assignment rule
8      n = m;
9      assert t <= n; // assignment rule
10     m = t;
11     assert m <= n; // if-then-else rule
12 } else {
13     assert m <= n && true; // strenthening
14     skip;
15     assert m <= n; // if-then-else rule
16 }
17 assert m <= n; // loop rule
18 while (m < n) {
19     assert m < n; // strenthening (equivalent)
20                   // is implied by loop condition => inductiveness
21     assert m+1 <= n; // assignment rule
22     m = m + 1;
23     assert m <= n; // invariant
24 }
25 assert !(m < n) && m <= n; // loop rule
26 assert m == n;
27 {(m = n)}
```

## Satisfiability

### Example 1, taken from the exam in June 2016

Check the satisfiability of the following SMT formulas. Assume that $x, y, z, a, b, c \in \mathbb{Z}$ are integer constants, and $f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ and $g : \mathbb{Z} \to \mathbb{Z}$ are binary and unary uninterpreted functions over integers respectively. Whenever a formula is satisfiable, give a satisfyingassignment for it, i.e., integer values for all variables and function interpretations overintegers that make the formula true under the assignment. Whenever a formula is notsatisfiable, give a reason why it is unsatisfiable.

| formula | SAT |
|---|---|
| $f(3,y) = 6 \wedge f(y,x) = f(x,y)$<br>$\wedge\, f(y,4) = 8 \wedge f(y,y) = 4$ | yes |
| $f(1,x) = 3 \wedge f(1,x) = f(x,1)$<br>$\wedge\, g(x) = f(1,x) \wedge g(g(g(1))) = 1$<br>$\wedge\, g(g(1))6 = f(x,1) \wedge x = g(g(1))$ | no,<br>$g(x) = 1 \not\downarrow\ g(x) = 3$ |
| $f(x,x) = x \wedge f(y,y) = y \wedge a6 = b \wedge f(x,y) = f(y,x)$<br>$\wedge\, f(0,1) = a \wedge f(1,0) = b \wedge (f(x,x) = 0$<br>$\vee\, f(x,x) = 1) \wedge (f(y,y) = 0 \vee f(y,y) = 1)$ | yes |

## Temporal Logic

### Example 1, taken from the exam in June 2016

Consider the following Kripke Structure:

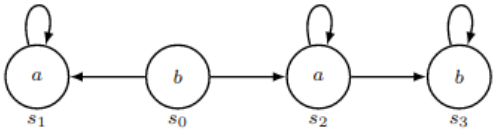


Figure 5: Kripke-structure

For each formula, give the states of the Kripke structure for which the formula holds. In other words, consider the computation trees starting with one of the states from the set $\{s_0, s_1, s_2, s_3\}$, and for each tree, check whether the given formula holds on it or not.

| formula | states in which it holds |
|---:|---|
| $b \wedge AXa$ | $\{s_0\}$ |
| $a \vee AXb$ | $\{s_1, s_2, s_3\}$ |
| $AFAGa \vee AFAGb$ | $\{s_0, s_1, s_2, s_3\}$ |
| $EFG\neg b$ | $\{s_0, s_1, s_2\}$ |
| $AGa$ | $\{s_1\}$ |

**Example 2, taken from the exam in June 2018**

Consider the following Kripke Structure:

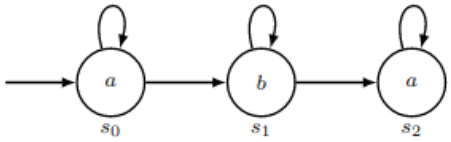


Figure 6: Kripke-structure

For each formula, give the states of the Kripke structure for which the formula holds. In other words, for each of the states from the set $\{s_0, s_1, s_2\}$, consider the computation trees starting at that state, and for each tree, check whether the given formula holds on it or not.

| formula | states in which it holds |
|---:|---|
| $AXa$ | $\{s_2\}$ |
| $EXa$ | $\{s_0, s_1, s_2\}$ |
| $AFb$ | $\{s_1\}$ |
| $EGa$ | $\{s_0, s_2\}$ |
| $A(aUb)$ | $\{s_1\}$ |

**Example 3, taken from the exam in June 2019**

**A)**

Consider the following Kripke Structure:

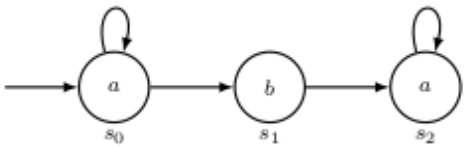


Figure 7: Kripke-structure

For each formula, give the states of the Kripke structure for which the formula holds. In other words, for each of the states from the set $\{s_0, s_1, s_2\}$, consider the computation trees starting at that state, and for each tree, check whether the given formula holds on it or not.

| formula | states in which it holds |
|---|---|
| $A(FGa)$ | $\{s_0, s_1, s_2\}$ |
| $AFAGa$ | $\{s_1, s_2\}$ (is $s_0$ correct?) |
| $A(a \wedge Xa)$ | $\{s_2\}$ |
| $E(bUa)$ | $\{s_0, s_1, s_2\}$ |

**B)**

Consider the following Kripke Structure with initial state $s_0$ :

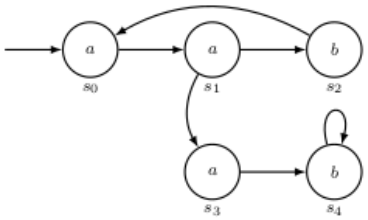


Figure 8: Kripke-structure

Does the LTL formula $AFXb$ hold in the initial state $s_0$ ?

$\rightarrow$ *yes, because all paths have to pass* $s_2$ *or* $s_4$ *eventually. Before they do so,* $Xb$ *will hold*

Does the CTL formula $AFAXb$ hold in the initial state $s_0$ ?

$\rightarrow$ *no, because the path that loops around* $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_0 \rightarrow \ldots$ *does not satisfy it.*