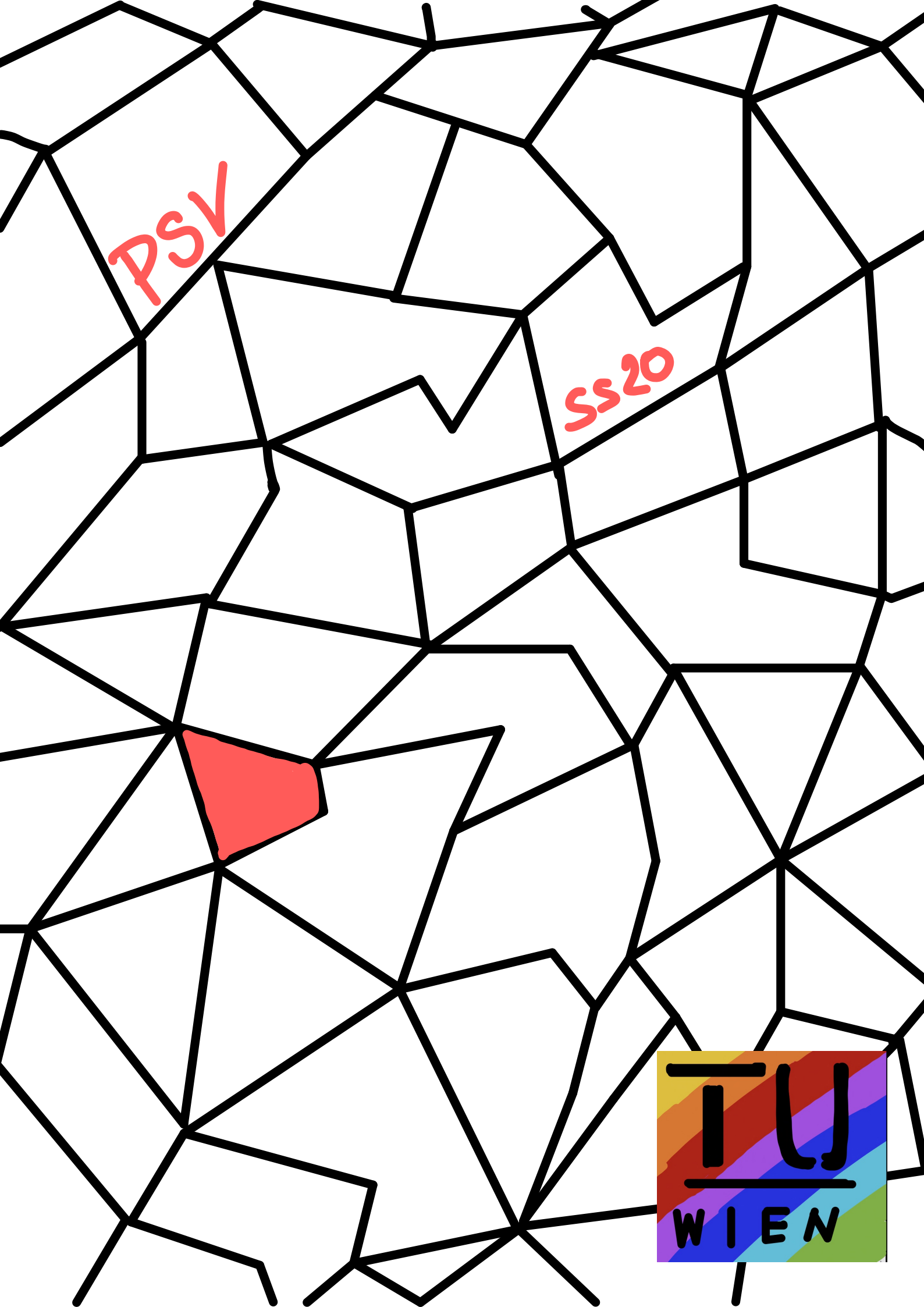
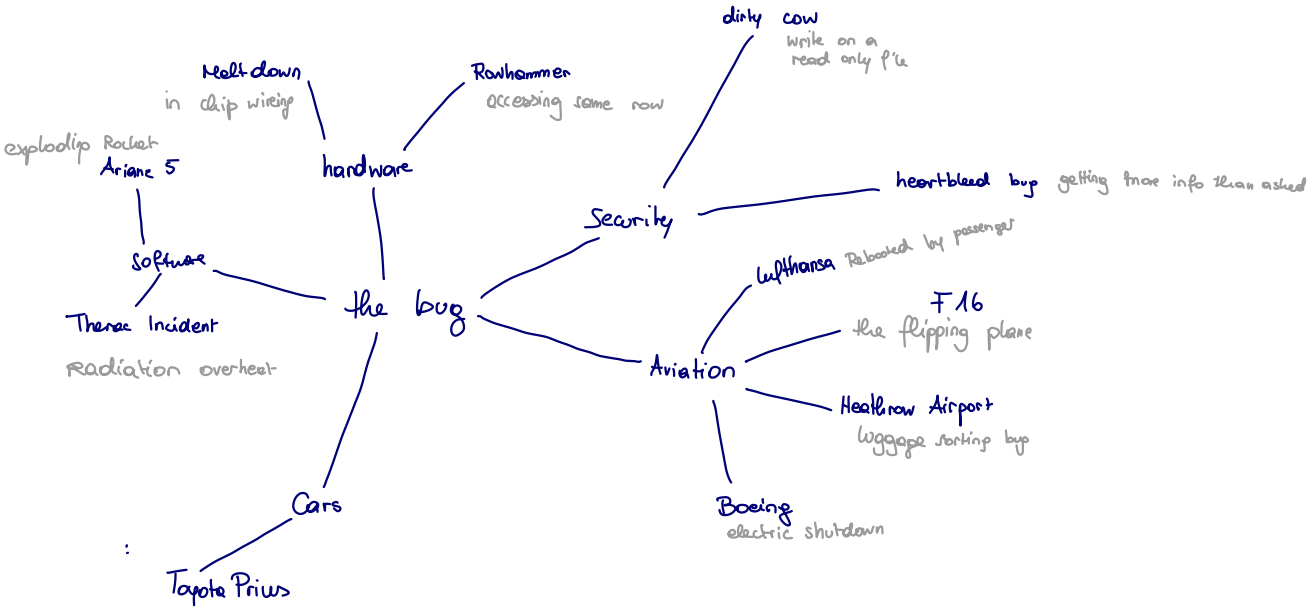

PSV
SS20
TU
WIEN

Motivation:



Halting Problem: | von Neuman Architectures | assertions: . -

Contents

Chapter 1: A bug's life

lecture 5.03.20

What is a bug? A flaw in a system that results in unintended behaviour

Kinds of Bugs

- arithmetic bug — overflow, division by 0, precision/conversion errors
- logic bug — infinite loops, off-by-one
- resource bug — null pointer deref, uninitialised variables, wrong data type ...
- multi-threading bugs — deadlock, livelock/starvation, race condition, atomicity violation
- syntax/semantic bugs — wrong operators/semantics
- interfacing bug — incorrect API usage/protocoll implementation/handeling...
- performance/timing bugs — computational complexity, random memory access
- teamworking/developement bugs — copy&paste errors, wrong source code, ...

Popular Bugs

- Bohrbug - simple
- Heisenbug - isn't there
- Schrödinbug - disappiers
- Mandelbug - chaotic

The bug "evolves" through a snowball system.

1) programmer makes fault — fault - mistake in coding
2) fault results in error — Error - incorrect state
3) error results in system failure — Failure - deviation from desired behavior

Focus your attention on

1) careful programming (use comments)
2) semantics & syntax — Programmer, compiler, CPU need to agree on semantics
   ↳ standards often in Backus Naur Form
3) fitting testcases (not inductive!)
4) use assertion statements
5) KISS — keep it simple and stupid

parallel programming leads to cross-compiling!

eg: MISRA Standard

Lecture 10.03.20

Chapter 2: Assertions

Definition: not executed, just tells programmers what to hold

Content: one (pre more) relations as equalities, inequalities, ... evaluate to true or false, no side effects

don't guarantee correctnes

partial specification (no complete programm description)

How do we proof invariants?

Assertions with sideeffects – bad idea

Increment: assert (++value)

Allocation: assert (p=(char *)malloc(5*sizeof(char)))

Function call: assert(fwrite(str,1,sizeof(str),fp))

Design by contract

pre & postcondition represent contract → pre condition doesn't hold
no postcondition holds

caller establishes precondition

callee guarantees post-condition

no double checking

History variables: no side effect on control flow / data flow

Only use assertions if you can controll whether they hold or not.

Java: IllegalArgumentException, NullPointerException, IllegalStateException (no throws clause!)

C++: domain error, invalid argument, length error, out of range

allow

strengthen precondition: fewer states

weaken postcondition: more states

Invariant assertion:

Checked at individual points during execution

assertion in loops must always hold (throughout & at the end of loop)

Express intent of programmer

powerful debugging tequnice

to proof programs correct

not useful for concurrent programs

Inductive Invariant assertion.

- n is number of loop iterations
- holds in every loop iteration
- if it holds for one, it holds for all

stream 21.03.19

Chapter 3: Testing

"Beware of bug in the above code, I have only proved it correct, not tried it."

→ Proofs are never 100% fool proof! (confidence by testing again and again...)

Empirial Falsification:

Think of hypothesis as falsifyable! (otherwise it is useless → needs to be put to test! → just confirm theory)

eg: All swans are white → no also black swans (good theory)

proofs rely on abstraction and assumptions (eg test with N → bitfactors behave different!)

not falsifyable specs: "Software shall be fast" → very subjective (bad specs)

assertions are falsifyable through counterexample (good specs)

How verify? → increase confidence

Systematic Testing specify the expected output! (input + expected output is needed)

eg Triangle Testing (equilateral, isosceles, scalene) → Blackbox testing

Testing ...

analyse subset, GOAL: Bugfinding (Person implementing ≠ Person testing!)

What is testing: intent to find errors, no proof of correctness!

Who should test: implicity testing during writing, (debug!), biased, document

What to test for: expected, valid behavior & not expected inputs eg to high temperature

Where to look for bugs: heuristic (more bugs in same area), sections that change often, high complexity (cyclomatic complexity)

When to stop: Coverage Criteria never 100% done

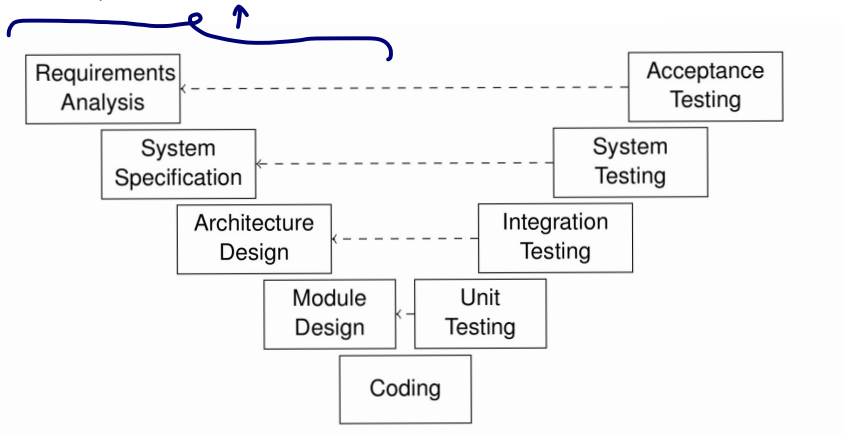
test cases (scenarios): in & out

test suit: collection of test cases

test run: execution of test case

Validation vs Verification: Are we building the right system? vs Are we building the system right?

Waterfall model to v-model



Unit Testing : look at code
formal : error checklists (interface errors, I/O errors, Comparisions, Controll flow bugs, aritmetic bugs ...)
lightweight : pair programming
libraries!

Running test cases: (run overnight) manually, tool assistance, automated

white box testing → look into it, come up with testcases

black box testing → triangles

- equivalence partitioning: many inputs are going to result in same output

1) Identify equivalence classes (valid & invalid) → cover as many as possible
not always coresponding

2) Define test cases

eg: Password rules

→ boundary testing:

- pick boundary values (elements close to boundaries of equivalence class)
- writing testcases: use assertions (but if 1 fails → everything else not executed)

eg: floating points

eg: passwords

eg: Balanced Binary Search Tree (AVL-Tree)

signature: |left height - right hight| ≤1

equivalence classes: deriving by elements etc (not always invalid classes!), rotating trees, ...

boundary testing: concrete examples

Random Testing: inferior, hard to generate valid input, finds simple bugs quickly!

Important:

specify your testcases

enumerate covered equivalence classes

cover output equivalence classes

equivalence classes can overlap ↓

critical systems → high redundancy process oriented

Chapter 4: Coverage Criteria

test invalid equivalence classes individually!

abstraction from testing, social construct (no criteria that -then are no more bugs)

blackboxtesting: acceptance testing, system testing → automate: gcov (contains coverage info)

Coverage Criteria: Whitebox testing

What is enough? : Not every input can be covered!
Visit all possible states? → too many
→ abstraction of the programm

cyclomatic complexity: gives upperbound of max. testcases
stack: Used for recursive function calls
heap: Memory that can be allocated

- Common criteria on what sufficiently tested means (confidence & certification) → never look at code and think of test cases!
- achieving coverage is not a goal itself (not look for testcases with highest coverage!)
- general criteria: controlflow (structure of Programm), dataflow (where are assignements to variables actually used), mutation ...

① CONTROLFLOW

a) Path coverage: Path can be seen as equivalence class → loops can't be tested by path (n iterations → 2^n paths ∴)

b) Statement coverage: every statement at least once (overlap of equivalence classes) → stop after 1 testcase (not ex all outcomes!)

c) branch coverage: every branch ≥ statement coverage (if then else → both)

d) decision coverage: every boolean decision (all decisions covered without testing)

} not always the same! → definition of 'branch'

Battle of definitions:

'branch'

a) one/two statements selected for execution
b) execute each outcome
→ unconditional branches (goto), fall throughs (switch, break) function calls

'decision'

a) "expression with sideeffects"
b) boolean expression

branch coverage ≥ desicion coverage (if decision = booleans only)

desicion coverage > branch coverage (if branch doesn't include unconditional jumps)

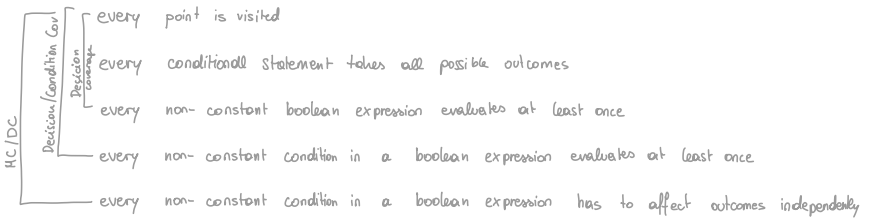
e) condition coverage: exercise every sub-expression/atom (no booleans) / condition-outcome ≯ decision coverage

f) condition/decision coverage: still not all branches executed

g) MC/DC (modified): every condition outcome must affect the decision outcome independently

h) mulliple conditional coverage: cover all possible combinations

eg:



② DATAFLOW

Values propagete through: definition → use → def-use chain

assign value → value is read → new value assigned etc

- flow into boolean expressions in conditional statements (p-use(x))
- flow into variables used to define other values (c-use(x))
- notation def(x); p-use(x); c-use(x); $dpu(l,x)$; $dcu(l,x)$ (set of all potential predicate values that are reachable (def influence the use))
  - $dcu(l,x)$: x is used to compute new value in assignment statement
- def-clear: no loop iterates twice

```
while ()
  i=i+1
```

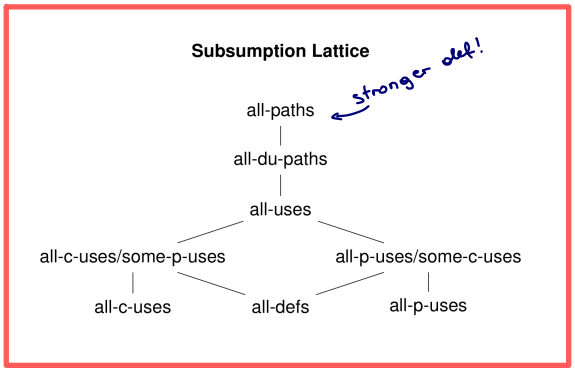
Criteria: For each def of x and for every $l \in def(x)$, the test suit traverses:

weak

- all defs: all definitions are used $(dpu(l,x) \cup dcu(l,x))$
- all c uses: all comp (computation) uses affected by each def are executed $l' \in dcu(l,x)$ (if emty, do none → bad, so?)
- all p uses: all desicions affected by each def are executed $l' \in dpu(l,x)$
- all c uses / some p uses: all defs & affect comp then these comps are executed $dcu(l,x)$ & $dcu(l,x) \neq \emptyset \Rightarrow dpu(l,x)$
- all p uses / some c uses: all defs & affect desicions then these des are executed (only predicates are used!) $dpu(l,x)$ & $dpu(l,x) \neq \emptyset \Rightarrow dcu(l,x)$
- all uses: every use has to be covered

strong

- all du-paths: all possible ways of reaching the use



→ Dataflow criteria show dependencies between variables

→ Set all pairs can be aprox by static analysis

allways give reason why dcu/... can't be achieved

③ MUTATION TESTING

test testsuit through built in bugs (also prg mutations)

each mutant should be caught by a testcase

Obstacle:

weak-mutation testing: requires triggered fault → error

strong-mutation testing: requires programm failure

equivalent mutant (Modification in programm doesn't change it), Based on hypothesis

Fuzzing: Mutating the input data (checking the robustance) fault injection

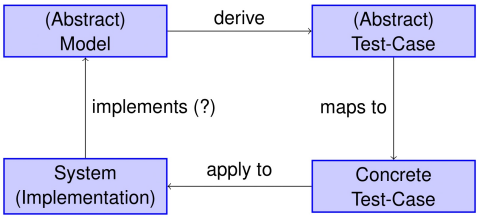
Stream von 28.03.19

Chapter 5: Test Case Generation

testing is a manual process → can we automate that?

derive results from implementation not from specification!

never generate exclusively from the programm

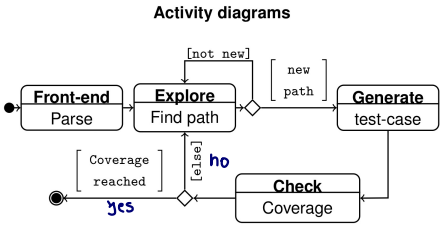


derive test cases from a model (more abstract level, not necessarily executable, easier to understand)

Common modelling languages: UML (structure & architecture), OCL (expressions with pre/post conditions), Matlab/Simulink (Automotive industry)

more abstract than C/C++ ...

Abstract with UML



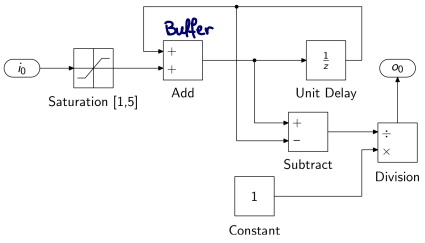
derive abstract testcases

feasability: concrete testcase?

predict outcome?

Coverage guarantee not given!

less abstract with Simulink:



Saturation cuts of unnecessary things, computes $\frac{1}{t_n - t_{n-1}} - 1$

allows code generation

derive expected output is possible

→ used for prototyping

Never ever

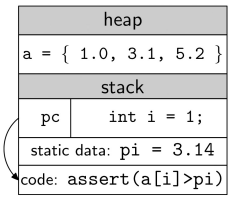
- extract testcases from the implementation (at least a specification)
- apply testcases from a model to generated code
- let coverage criteria derive your test case generation

Model-based-testing: decouple TCG (test case generation) from specification

Inputs that can make the system crash: buffer overflows, division by zero, invalid pointer references, assertion violations (most general mechanism)

assertion violation:

expression evaluates to false (depends on values, heaps...)



Fail an Assertion: find suitable, reachable state

satisfyable: an valuation exists that makes it true

unsatisfyable: no valuation exists that makes it true
no assignment makes it

decision procedure alias SMT (Satisfiability Modulo Theory — LISP

→ witness for satisfiability (if it works)

23 - SmT Programm Crash course:

- variable need to be declared and typed (variable are const/ null array functions)
- Add assertions over declared variables
- check satisfiability "sat" ✓
- ask for a model

what we can do:

bit vectors implementation eg pi, eg i

negate assertion → find preconditions that is reachable

Symbolic execution

1) perform symbolic path execution
2) ask smt solver at any assertion if violated

symbolic values: placeholder
concrete values: real values we know from bitvectors...

running programm → symbolic values → ask smt solver if prediction is satisfied
→ with conditions: all conditions must be satisfied

Reachability tree (feasable & infeasable branches)

Scalability 1) constraint solving

2) Path explosion → the deeper we explore, the longer it gets → solution: Search heuristics

BFS: explore all paths with k, before going to k+1 ⇒ short paths.

DFS: follow the path to the end (& backtrack) ⇒ memory efficient

(Best-First Search) Coverage based: close to uncovered instruction, favor recently visited new ones path

Random selection: higher probability to shorter paths (⇒ no starvation)

KLEE select combination of BFS, DFS, ...

Always possible: Eliminate redundant paths!, merge paths (bounded model checking)

TOOLS:
KLEE
PEX

TCG also used to check contracts / oracle (inefficient as specs, optimized as implementation)

(Chapter 5.1) Examples for Bugs, Assertions, ...

Heartbleed bug: length never checked

Assertions as formal spec: write a failing assertion - can't use assertion as oracle!

Euclid's Algorithm: ggT

Data Flow Based coverage Metrics → Sometimes not 100% coverage (set testing goals)

If you don't trust testing - prove the programm correct

with assertions (in concurrent environment problematically)

with invariants

Byte swapping trick (using XOR instead of 3rd helper variable

strengthen precondition

Assertions & Concurrency: Locks are used to prevent simultaneous/concurent access to critical regions

↪ Deadlocks can be spec. using assertions (happening be acquired in wrong order

make assertion fail when there is a deadlock, (realise with flags)

Stream von 09.04.2019

Chapter 6: Propositional and First Order logic (Formal Languages)

repeating: Semantics

`x = ++y` y is increased for 1 and assigned to x.

`*p++ = &c` Add Adress of c to the pointer p adress and point to next element

`a[i++] = b[i++]` Not defined! undefined = compiler-dependent

Now we are going to use formal languages to specify semantics & formally reason about the correctness

Propositional logic (PL - Aussagenlogik)

First-Order-Logic (FOL, Prädikatenlogik)

① Propositional logic

Syntax: generates everything inductively, formulars can be built recursively from syntax, can be parsed recursively, specs the structure

formula := formula ∧ formular | formula ∨ formula | formular ⇒ formula | formula ⇔ formula | ¬formula | (formula) | atom

atom := identifier | constant

constant := true | false

identifier ∈ {P, Q, R, ...}

Semantics: identifies the formulars, interpetations assign truth variables to identifiers

n Propositions → 2^{2^n} possible Operators to specify

Also Inductive Definition of Semantics (Base case & induction step)

Don't confuse meta language with formal language

Satisfiability: is satisfiable if there exists an interpretation which models the formula

Validity: If all possible interpretations make the formular true.

F is Valid if ¬F is unsatisfiable.

↳ NP Complete Problem, SAT Solvers ...

equivalent: all interpretations are ex. the same

models: mappings from variables to values

entails: model is subset of other model

equisatisfiable: works if formulars are completely different

$P \wedge Q$ and $R \wedge (R \Leftrightarrow (P \wedge Q))$

good data structure to represent this?

-) Proprositional logic
-) Negation normal form
-) Polynomials in GF(2)
-) Boolean Circuits
-) BDDs

works for everything

Canonicity: getting it back to 0 / 1

works for

- truth table → blowing up representations
- propositional logic → Algorithm (expensive)
- BDDs

functional completeness:

Boolean structure is complete if all boo functs are expressed by data

resulting in functional completeness:

(¬, ∨) & (¬, ∧) & (NAND) & (NOR)

with BDDs: If you have two formulars, combine these through Shannon $(\neg x \wedge F[x/0]) \vee (x \wedge F[x/1])$

variable order matters! (size can expand from linear to exp() ⇒ heuristics pick order)

with Normal forms: nnf: negation can only show up next to atom

cnf: every junction contains clause of negated or not negated

Tseitin Encoding: construct satisfiably-equivalent formular

with Bitvectors: fresh PL variable for every single bit variable — calculating through circuits

$\sum_{i=0}^{n-1} d_i \cdot 2^i$ unsigned

$-2^{n-1} d_{n-1} + \sum_{i=0}^{n-2} d_i 2^i$ signed

lsb → number is even/odd

Restrictions for prop logic: sufficient to encode bit-vector operations

② FIRST ORDER LOGIC

Syntax

formula := formula ∧ formular | formula ∨ formula | formular ⇒ formula | formula ⇔ formula | ¬formula | (formula)|

predicate(term, ..., term) | term = term | ∀ variable . formula | ∃ variable . formular

term := variable | constant | function(term, ..., term)

logical symbols: ∀, ∃, ∧, ∨, ⇒, ⇔, ¬

* entirely depends of interpretation!

variables: unique identifiers, logical symbols.

functions: unique identifiers, fixed arity (Stellenwert) non logical symbols*

predicates: unique identifiers, fixed arity (Stellenwert) non logical symbols

constants: unique identifiers, non logical symbols

By itself FOL has only very loose constrains about it's interpretation

Semantics

determined by models

- Pick domain in which we interpret (arbitrary non empty domain)
- can't determine truth unless all variables are quantified

Satisfiability: If there exists a model such that the model makes formula true

validity: If for all possible interpretations it holds that the interpretations (all domains) make the formula true.

F is valid if not ¬F is unsatisfiable.
↳ Proof by contradiction!

Substitution:

Replace variable / term : Save if renaming variable (no terms that are in scope of quantifiers can be replaced!)

Define Formular scheme & side conditions

Inference Rules

Premise
Conclusion

Resolution:

$$\frac{P \vee Q \quad \neg P \vee R}{Q \vee R}$$

$$\frac{\neg\neg P}{P} \qquad \frac{P}{\neg\neg P}$$

$$\frac{}{\text{conclusion}}$$

▶ For instance, for arbitrary formulas P, Q, R:

$$\frac{\neg\neg P}{P} \quad \frac{P}{\neg\neg P} \quad \frac{P \quad Q}{P \wedge Q} \quad \frac{P \wedge Q}{P} \quad \frac{P \wedge Q}{Q \wedge P} \quad \frac{P}{P \vee Q}$$

$$\frac{P \vee Q \quad \neg P \vee R}{Q \vee R} \quad \frac{P \Leftrightarrow Q \quad Q}{P} \quad \frac{P \Rightarrow Q \quad Q \Rightarrow P}{P \Leftrightarrow Q}$$

used by satisfyability checker

interpreted as placeholders / propositional rules

Proof system is incomplete, it can be entailed ($\models$) but not derived $\vdash$

Proof system is unsigned, it can't be entailed ($\models$) but derived ($\vdash$)

reductio ad absurdum: $\dfrac{P \vdash Q \quad P \vdash \neg Q}{\neg P}$

deduction theorem: $\dfrac{P \vdash Q}{P \Rightarrow Q}$

case analysis: $\dfrac{P \vee Q \quad P \vdash R \quad Q \vdash R}{R}$

Axioms (inferences without premisses)

used in arrays

Peano arithmetic

Chapter 7: Hoare Logic

→ formally reason about assertions holding a programm (logical interpretation / meaning)

Hoare - Floyd: Systems of axioms

basic element Precondition - Statement - Postcondition $\{P\}\ C\ \{Q\}$

translate program to FOL & reason it $P(x_0,\dots) \wedge C(x_0,\dots,x_1\dots) \Rightarrow Q(x_{1/4},\dots)$

→ not strongest possible outcome

BASIC RULES

"Skip" rule : Condition holds afterwards $\dfrac{}{\{P\}\ \text{skip}\ \{P\}}$

"assignment" rule : (reversed: starting with Post Condition – what must be satisfied) $\dfrac{}{\{Q[E/x]\}\ x := E\ \{Q\}}$

Q states something about outcome - same thing for E in $P/Q \Rightarrow Q$ will hold!

Don't bind the variables in the expression! Careful if one variable is quantified.

"composition" rule : With two hoare triples and Precondition triple 1 = Postcondition triple 2, (patch / merge) these together

"conditional" rule : only linear execution, reasoning about branches individually ; P, Q must be perfect match!
replace precoditions, bc other precon is stronger

"consequence" rule: Combining FOL with Hoare weaken precondition / strengthen post condition
strengthen pre-condition / weaken post condition

$Q \to Q'$ logical weaker than Q Q' character

"while loops" rule : Statement doesn't change P, P holds throughout loop (always require / inductive invariant)

With all rules we can syntactically find the values (except inductive invariants)

Challenge: Find the inductive invariant

1st example Stream xx – 0:51:00

2nd ex: greatest common divisor

3rd ex: Euclids algorithm proof by casesplit

$$\frac{}{\{P[E/x]\}\ x := E\ \{P\}} \qquad \frac{\{P\}\ C_1\ \{Q\},\{Q\}\ C_2\ \{R\}}{\{P\}\ C_1\ ;\ C_2\ \{R\}}$$

$$\frac{\{B \wedge P\}\ C_1\ \{Q\}\ \{\neg B \wedge P\}\ C_2\ \{Q\}}{\{P\}\ \text{if } B \text{ then } C_1 \text{ else } C_2\ \{Q\}}$$

$$\frac{P' \to P \quad \{P\}\ C\ \{Q\} \quad Q \to Q'}{\{P'\}\ C\ \{Q'\}}$$

not possible to automate finding a loop variant
→ requires intuition to find it

$$\frac{\{P \wedge B\}\ C\ \{P\}}{\{P\}\ \text{while } B \text{ do } C\ \{\neg B \wedge P\}}$$ Loops

Chapter 8: Bounded Model Checking

hard to synthesize invariants → ignore them, restrict to other rules (get rid of loops) replace loops with ITE

For bug finding not proofing programm correct. (Response time & limited memory ressource)

stronger than symbolic execution (more than one path at the same time)

Predicate Transformer: Strongest predicate queue – Post condition holds (weaken Post condition → holds)

ignore loops

Strongest post-condition: (no quantifier needed?) → new variable memorizing history

$\{P\}\ stmt\ \{Q'\} \triangleq sp(stmt, P) \Rightarrow Q'$ (holds / statement) — doesn't have to hold after (only for old values)

strongest postcondition

... for assignment $sp(x := e, P) \overset{def}{=} \exists x'.\ (x = e[\frac{x}{x'}]) \wedge P[\frac{x}{x'}]$ → can't rly get rid of quantifier → simple theory yes → arrays no (decision procedure)

... for assertion $sp(assert(R), P) \overset{def}{=} P \wedge R$ restricting the states & assertion if P/R inconsistent ⇒ sp is false (no reachable state)

... for sequential execution $sp(stmt_1; stmt_2, P) \overset{def}{=} sp(stmt_2, sp(stmt_1, P))$ (precon / precondition)

... for conditionals $sp(\text{if } B \text{ then } C_1 \text{ else } C_2, P) \overset{def}{=} sp(C_1, B \wedge P) \overset{\text{merging}}{\vee} sp(C_2, \underset{\text{ite doesn't hold}}{\neg B} \wedge P)$ interpret as set of states reachable (union of set of states)

path wise unwinding → mehr ≙ (formular would grow n, with paths e^n!)

Loops: can't easily compute strongest postcondition (= reachable set of states → inductive invariant) ⇒ would proof turing incorrect (halting problem)

⇒ only deal with fixed number of loop iteration

≙ single static assignment form

unwind loop bodies and merge individually

Unwinding assertion: provide bound n

compute strongest post condition } $\{P\}\{stmt\}\{Q\}$ (FOL) ; if $P \wedge \neg B$ is sat → error

CBMC – Tutorial bounds-check, non deterministic (return value = arbitrary) everything could happen

non deterministic: all paths are checked → (test harness: calls function under test, parameters non det)

randomness: certain probability for each path → (function stubs: modeling functions for operating systems)

Chapter 9: SAT-Solvers

switching between algorithms & logic (describing behavior)

SAT:

- boolean satisfyability (central problem → is expression satisfyable ≙ is it true) "existence of assignment"
- contains literals (variables) and clauses (conjunctions of literals)
- NP complete (reduction to SAT) simplifying, finding consequence and (no) conflict

some SAT examples:

$(a \vee b) \wedge \bar{a} \wedge \bar{b}$	not SAT
$(a \vee \bar{b}) \vee (\bar{a} \wedge \bar{b})$	SAT ($b = false$)
$(a \vee b) \wedge c \wedge (\bar{a} \vee b \vee \bar{c}) \wedge (a \vee c) \wedge (\bar{b} \vee c)$	SAT ($b \vee c = true$)
$(a \vee b) \wedge c \wedge (a \vee b \vee c) \wedge (\bar{a} \vee \bar{c}) \wedge (\bar{b} \vee \bar{c})$	not SAT (c is true → false :-)
$(d \vee \bar{b}) \wedge (\bar{a} \vee \bar{c}) \wedge (\bar{d} \vee e) \wedge (a \vee b) \wedge (e \vee c \vee \bar{b}) \wedge c \wedge (\bar{b} \vee \bar{c}) \wedge (a \vee b \vee c)$	not SAT

how to use SAT in verification process? "Are there initial values that violate the spec"
↳ unSAT: there are no values

Software bounded model checking: Bug appears in first n steps → n steps in formula N & required property in formula A
(if $N \to A$ is SAT = A is not violated ≠ no bugs!)
(if $(\overline{N \to A})$ is SAT ⇒ BUG

Test case generation for circuit: concatenate C' with C, if $(\overline{C \Leftrightarrow C'})$ is SAT ⇒ fault

Testing in software: encode in boolean formula & bmc for bug finding

hardware: functionality correct/equivalence & test case gen

Algorithmic aspects of SAT

Clause = possibility for conflict

Naïve: (worst case) try every possible explanation (De Morgan & Distributivity)

not naive: CNF (conjunctive Normal form)

- transform to CNF (convenient set representation) → each clause is subset containing literals
- Tseitin transformation: per sub formula (1 new variable, clauses capturing relations between new var + org var)

$a \wedge b$ to f ⇒ $f \leftrightarrow (a \wedge b)$ in CNF $(\bar{f} \vee a) \wedge (\bar{f} \vee b) \wedge (\bar{a} \vee \bar{b} \vee f)$

DPLL algorithm: search algorithm for SAT

boolean constant propagation: simplify clauses

unit propagation: simplify formulars

notion of clause learning: search better with the algorithm
↳ prevent backtracking

```
                                          get formular as input
let rec DPLL(Δ) =                                  more clever (not fixed order) unterstandable
  let (Δ', L) = unit-propagation(Δ) in   get simplified formula with literals
  if Δ' = ∅ then if the similar formula is empty set
    return L resolved formular give back a SAT assignment
  else if ∅ ∈ Δ' then (the set is empty)
    return unsat
  else
    let a be some literal in Δ' in pic one literal
    let L_T = DPLL(Δ'|a) in call this algorithm with guessed literal (left side)
    if L_T ≠ unsat then
      return L_T ∪ L ∪ {a}
    else
      let L_F = DPLL(Δ'|¬a) in call right side of tree
      if L_F ≠ unsat then
        return L_F ∪ L ∪ {¬a}  SAT
      else
        return unsat
```

(Erfüllbarkeitsproblem für Formeln in KNF nach Backtracking)

Chapter 10: SMT Solvers

smt ... satisfiability modulo theories (FOL + new theories: equality logic, uninterpreted functions, linear arithmetic

equalities/inequalities possible, predicates possible, function possible, variables can be more than integers

Theories

equality logic: reason about SAT without knowing the domain (= don't care for output)

equality logic with uninterpreted functions: variables are variables

linear arithmetic: arithmetic interpretations ⇒ variables are numbers

general arithmetic: formulars, squares ...

bit vectors: int in C replaced with bits

quantifiers: ∀, ∃, etc SAT is about existencial quantifiers

example: assertions with smt

```
int n = input();
int x = input();

int m = n;
int y = x;
int z = 0;

assume(n >= 0);

/* loop invariant:
   m * x == z + n * y */

while (n > 0) {
  if (n % 2) {
     z += y;
  }
  y *= 2;
  n /= 2;
}
assert (z == m * x);
```

branching

proof with loop invariant

① precondition (before loop) implies the invariant
invariant holds before loop

② loop iteration
- copy of invariant after 1 iteration
- still satisfies loop invariant assuming loop condition was true
if it holds for 1 iteration it holds inductively for the others

terminates because n is getting smaller by every step

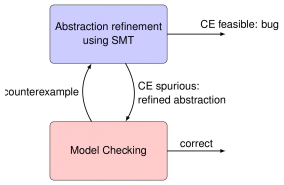
automatic generating invariants is problematic

not possible in SAT bc of fixed bit size, if no precision fixed → reason about arithmetic, use specifics

if code runs on fixed size → no verification for general arithmetic

example of SMT (Guided Abstraction Refinement)

spurious behaviour: might get infinite execution leads to spurious paths



simple decision procedures to use to reason about formulas

equality logic : logical connectives ($\wedge, \vee, \neg$), atoms (term), terms (variable, constant), domain (int, real ...)

how to SAT

1) replace constants by variables

2) add constraints

3) check SAT with merging (if there are two equivalent variables with $a \neq b$ return unsat)

equality logic with uninterpreted functions: logical connectives ($\wedge, \vee, \neg$), atoms (term), terms (variable, constant; predicate, function with parameters), domain (int, real ...)

two programms with different domains are equivalent

using other axioms (beluga – can get bigger!)

better: reducing to equality logic (replace functions by variables, capture functional consistency

Arithmetic: Gaussian elimination $\triangleq$ SAT for formulas! (sometimes simpler with branch & bound)

Summary

SAT is useful for integers & small numbers

SMT is useful for general results / special specs / logic extensions

Chapter 11: Model Checking

What is my problem? How do I want to solve it?

Need for automated verification, mathematical object (eg programm is just a formula), describe algorithms that can apply to objects

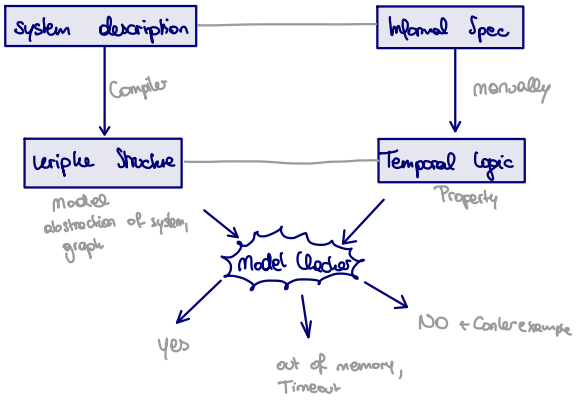
way to describe programs, systems as mathematical things in order to reason about them!

Display programs as graphs (the more traces the more branches!)

Model Checking in a nutshell

GOAL: System description as input

Checks whether kripke structure satisfies property



Model Checker Input: kripke structure

- M=(states, initial states, transition relation, atomic proposition, labeling)
  - states: vertices
  - initial states: incoming edge without prev. state
  - transition relation: lines between vertices
  - atomic proposition: highlight certain property about states
  - labeling: which atomic proposition holds in which state

- mathematical problem, no data structures
- compiling in kripke structure: state explosion

Model Checker Input: Specifications

Describe desired properties of kripke structure (describe property of computation tree → Comp tree Logic)

Verification Process:

① modelling phase: model system with compiler – sanity check (simulation) – formalize property to be checked (in temp. logic)

② running phase: run model checker to check validity

③ analysis phase: satisfied? → check next: analyze counter eg, refine model/design/property, repeat ①

OUT_OF_MEMORY: Reduce model, try again

Kripke Structures (System description – how program is evolving)

represent dynamic behavior of the system (labbeled transition system / labbeled graph)

five tuple $M = (S, S_0, R, AP, L)$ — finding states that are labelled with the propositions

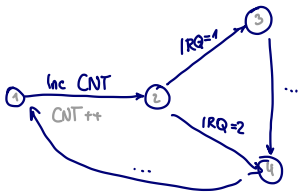
- S ... finite set of states
- S_0 ... initial state : $S_0 \subseteq S$
- R ... transition Relation: $R \subseteq S \times S$ with $\forall s \exists s' : (s, s') \in R$ (no deadlocks per definition)
- AP ... set of atomic propositions
- L ... Label : subset of atomic propositions $L: S \to 2^{AP}$

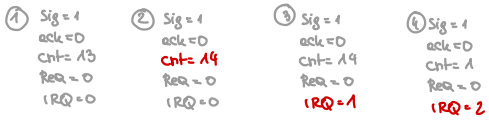
represented as graph (nodes = reachable system states, edges = state transitions)
in each node: set of atomic propositions $L(s) \in 2^{AP}$ that hold on $s \in S$

Path: infinite sequence of states $\pi = s_0, s_1, \ldots$ such that for $i \geq 0$ $(s_i, s_{i+1} \in R)$ π^i denotes the suffix of π starting at s_i

Path s_1, s_2, s_3, s_4 $\pi^3 = s_3, s_4, \ldots$
it is possible to construct infinite path through Kripke struct.

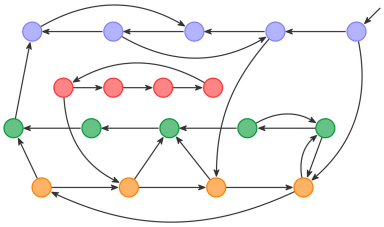
Models of dynamic systems





branching is possible?
eg: very complicated very fast

Do I reach green label? yes
Do I reach red label? no



Temporal logic: How to express certain properties?

→ extension: propositional logic + operators referring to behaviour of system over time

Models contain several states

formulas can be true / false in certain states (dynamic notion of truth)

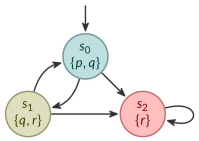
models = kripke structures / transition systems

properties = formulars in temporal logic
↳ can be specified: CTSL ← { Correctness, reachability, safety, liveness }

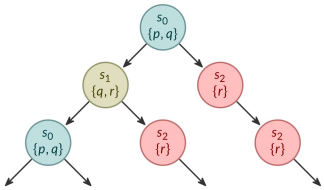
Computation Trees CTL

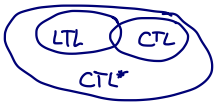
Kripke structure into infinite tree (root = initial state)

Does CT from temporal logic match CT from Kripke Structure ?



Kripke structure, models system





CTL*: Computational Tree Logic (invented after CTL)
Branching time Logic (repeatedly quantifying)

path quantifiers Does a certain path start at this state? only about branching $A\ (=\forall)\ / A\varphi$ & $E\ (\exists)\ / E\varphi$

temporal operators temporal evolution : $X\varphi$: next time φ $F\varphi$: eventually, in the future $G\varphi$: always, globally $\varphi\ U\ \varphi$ φ until φ

eg for CTL* formulas

- AGEF restart : always (AG) possible to restart the system
- AGEX progress : (AG) always is a next state = no deadlocks
- AG(req → AF ack) : (AG) always (req) request is (AF) for everything (ack) acknowledged
- EF(started ∧ ¬ready) : (E) it is possible to (F) reach a state in the future (started ∧ ¬ready) where system was started and is not ready
- A(GF enabled → GF executed) infinitely often

AGEF AG... on all paths something holds forever= always holds
GF... infinitely often

state formula : interpreted over states (boolean proposition, temp formula with path quantifier

path formula: interpreted over paths

if $\varphi \in AP$ → state formula
if φ is arbitrary CTL* formula, $A\varphi$, $E\varphi$ state formula
if φ, ψ are state formulas $\neg\varphi / \varphi \wedge \psi / \varphi \vee \psi$ are state formulas

Syntax:

A1 If $p \in AP$, then p is a CTL* formula.
A2 If φ is a CTL* formula, then $\neg\varphi$, $\mathbf{A}\,\varphi$, $\mathbf{E}\,\varphi$, $\mathbf{X}\,\varphi$, $\mathbf{F}\,\varphi$, and $\mathbf{G}\,\varphi$ are CTL* formulas.
A3 If φ and ψ are CTL* formulas, then $\varphi \wedge \psi$, $\varphi \vee \psi$, and $\varphi\,\mathbf{U}\,\psi$ are CTL* formulas.

Semantics:

⊨ modeling relation is inductively over the formula structure

$M, s \models \varphi$ holds at s

$M, \pi \models \varphi$ holds at π

State formulas

φ_1... state
φ ... path

(1) $M, s \models p \iff p \in L(s)$, for $p \in AP$
(2) $M, s \models \neg\varphi \iff M, s \not\models \varphi$
(3) $M, s \models \varphi_1 \vee \varphi_2 \iff M, s \models \varphi_1$ or $M, s \models \varphi_2$
(4) $M, s \models \varphi_1 \wedge \varphi_2 \iff M, s \models \varphi_1$ and $M, s \models \varphi_2$
(5) $M, s \models \mathbf{E}\,\psi \iff$ there is a path π starting at state s such that $M, \pi \models \psi$
(6) $M, s \models \mathbf{A}\,\psi \iff$ for every path π starting at state s we have $M, \pi \models \psi$

path quantifiers and temporal operators occur in pairs

B1 If $p \in AP$, then p is a CTL formula.
B2 If φ is a CTL formula, then $\neg\varphi$, $\mathbf{AX}\,\varphi$, $\mathbf{EX}\,\varphi$, $\mathbf{AF}\,\varphi$, $\mathbf{EF}\,\varphi$, $\mathbf{AG}\,\varphi$, and $\mathbf{EG}\,\varphi$ are CTL formulas.
B3 If φ and ψ are CTL formulas, then $\varphi \wedge \psi$, $\varphi \vee \psi$, $\mathbf{E}[\varphi\,\mathbf{U}\,\psi]$, and $\mathbf{A}[\varphi\,\mathbf{U}\,\psi]$ are CTL formulas.

Path formulas

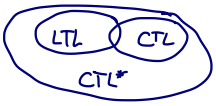
(7) $M, \pi \models \varphi \iff s$ is the first state on path π and $M, s \models \varphi$
(8) $M, \pi \models \neg\psi \iff M, \pi \not\models \psi$
(9) $M, \pi \models \psi_1 \vee \psi_2 \iff M, \pi \models \psi_1$ or $M, \pi \models \psi_2$
(10) $M, \pi \models \psi_1 \wedge \psi_2 \iff M, \pi \models \psi_1$ and $M, \pi \models \psi_2$
(11) $M, \pi \models \mathbf{X}\,\psi \iff M, \pi^1 \models \psi$
(12) $M, \pi \models \mathbf{F}\,\psi \iff$ there exists a $k \geq 0$ such that $M, \pi^k \models \psi$
(13) $M, \pi \models \mathbf{G}\,\psi \iff$ for all $i \geq 0, M, \pi^i \models \psi$
(14) $M, \pi \models \psi_1\,\mathbf{U}\,\psi_2 \iff$ there exists a $k \geq 0$ such that $M, \pi^k \models \psi_2$ and for all $0 \leq j < k, M, \pi^j \models \psi_1$

Basing logic on just EX, EG, EU

$$\mathbf{AX}\,\varphi \equiv \neg\mathbf{EX}\neg\varphi$$
$$\mathbf{EF}\,\varphi \equiv \mathbf{E}[\mathit{True}\,\mathbf{U}\,\varphi]$$
$$\mathbf{AG}\,\varphi \equiv \neg\mathbf{EF}\neg\varphi$$
$$\neg\mathbf{AF}\,\varphi \equiv \neg\mathbf{EG}\neg\varphi$$
$$\mathbf{A}[\varphi\,\mathbf{U}\,\psi] \equiv \neg\mathbf{E}[\neg\psi\,\mathbf{U}(\neg\varphi \wedge \neg\psi)] \wedge \neg\mathbf{EG}\neg\psi$$

(5) $M, s \models \mathbf{EX}\,\varphi \iff$ there exists a state t such that $R(s, t)$ and $M, t \models \varphi$
(6) $M, s \models \mathbf{EG}\,\varphi \iff$ there exists a path π starting at s such that for all $i \geq 0, M, \pi^i \models \varphi$
(7) $M, s \models \mathbf{E}[\varphi_1\,\mathbf{U}\,\varphi_2] \iff$ there exists a path π starting at s and there exists a $k \geq 0$ such that $M, \pi^k \models \varphi_2$ and for all $0 \leq j < k, M, \pi^j \models \varphi_1$

This definitions together with the following definition of $\models$ over M are equivalent to to the semantics derived from CTL*.
(8) $M \models \varphi \iff$ for all $s \in S_0, M, s \models \varphi$



ACTL*: Action Computation tree logic Branching time logic (repeatedly quantifying)

Sublogic of CTL, where only "A" is allowed as path quantification

All ACTL* formulas are in NNF

Syntax:

- If $p \in AP$, then p and $\bar{p}$ are ACTL* formulas
- if φ is an ACTL* formula, then $A\varphi, X\varphi, F\varphi, G\varphi$ are ACTL* formulas.
- If φ and ψ are ACTL* formulas, then $\varphi \wedge \psi$ / $\varphi \vee \psi$ / $\varphi U \psi$ are ACTL* formulas

ACTL Syntax: (ACTL = CTL + ACTL*)

- If $p \in AP$, then p and $\bar{p}$ are ACTL formulas
- if φ is an ACTL* formula, then $AX\varphi, AF\varphi, AG\varphi$ are ACTL formulas.
- If φ and ψ are ACTL* formulas, then $\varphi \wedge \psi$ / $\varphi \vee \psi$ / $A[\varphi U \psi]$ are ACTL formulas

LTL: Linear Time logic -> Linear time logic (single universal path quantifier)

Syntax:

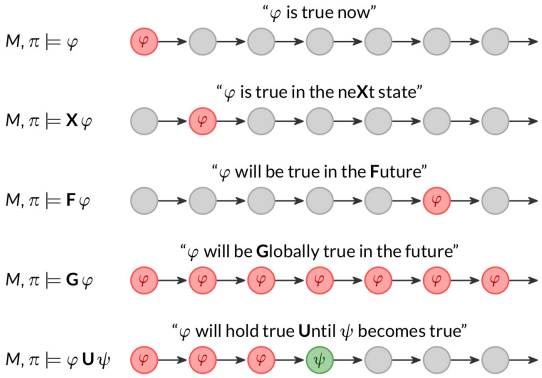
If $p \in AP$, then p is a LTL formula,

If φ is an LTL formula, then $\bar{\varphi}, X\varphi, F\varphi, G\varphi$ are LTL formulas

If φ and ψ are LTL formulas, then $\varphi \wedge \psi$ / $\varphi \vee \psi$ / $\varphi U \psi$ are LTL formulas

Semantics: derived from CTL*

$M \models \varphi \Leftrightarrow M \models A\varphi$



The model checking PROBLEM:

„If I have a Kripke structure and a property in temporal logic, I want to find all states s so that $M, s \models \varphi$"

„Want to solve a problem, check whether initial states are a subset of the states that satisfy this formula"

Solution:

Theorem: The time complexity of CTL model checking is linear in the size of the specification and the transition system

Verification: based on systematically analyzing transition systems

↳ Runtime: determined by number of states... exponential growth

Program graph representation: infinite many states (N^k)

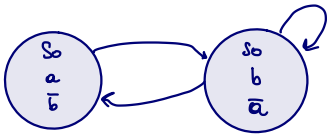
also: symbolic model checking

Chapter 12: Symbolic Model Checking

Verification for synchronous circuits

Kripke Structure to Logic formulas

a)



transitions

$R(a, b, a', b') \equiv (a \wedge \bar{b} \wedge \bar{a}' \wedge \bar{b}') \vee (\bar{a} \wedge b \wedge \bar{a}' \wedge b') \vee (\bar{a} \wedge b \wedge a' \wedge \bar{b}')$

starting with S_0

b) $EX\, f(a,b) \rightarrow f(a,b) \equiv a \wedge \bar{b} \Rightarrow \{s : a \in L(s) \wedge b \notin L(s)\}$

Formula that describes all states

states

$pre(f(a,b)) \equiv \exists a', b' \,.\, R(a,b,a',b') \wedge f(a',b')$

$pre(f) \equiv \exists a'b'.\ R(a,b,a',b') \wedge f(a',b')$
$\equiv (R(a,b,\text{false},\text{false}) \wedge f(\text{false},\text{false})) \vee$
$(R(a,b,\text{false},\text{true}) \wedge f(\text{false},\text{true})) \vee$
$(R(a,b,\text{true},\text{false}) \wedge f(\text{true},\text{false})) \vee$
$(R(a,b,\text{true},\text{true}) \wedge f(\text{true},\text{true}))$

$M, s \models EX\, f(a,b)$ (states)

$s \rightarrow pre(f)$ if only if $M, s \models EX f(a,b)$

```
let rec fixed-point (f) =
  if pre(f) ↔ f then
    f
  else
    fixed-point (pre(f) ∨ f)
```

c) Combine to formula and test SAT (has to stop - fixed point is reached)

→ Efficiency reached with OBDDs

SMV Tool — Symbolic Model Verifier

Language characteristics

Fairness constraint: true infinitely often

- description of (a)synchronous systems
  - assign: new state variable, invariant
  - define: macro definition, simple expressions, with BDDs
- modularized / hierachial descriptions
  - module: instantiated variable descriptions, module main, scoping, passed by reference, contain fairness and specs
    individual finite state machines
    - synchronous composition: parallel
    - asynchronous composition: prefixing module with keyword
- finite data types: booleans, enum, integer
- nondeterminism and partial implementations — nondeterminism: model undefined implementation, in abstract models
- variety of specs: safety, lienes, deadlock freedom

– parallel assignment language, checks for circularities & duplicate assignments

Promela (Process Meta language for asynchronous systems)

Tool for interpretation: Spin

interprocess communication to change variables

(asynchronous) message passing (→ recieving blocked)

Mars Pathfinder

Problem: Concurrent threads – low priority task to high priority task ...→resulted in deadlock