Program and System Verification (SS2022) TU Wien

Summary

15.06.2022 Summary PSV

From: Thinklex

To: The masses

1 Overview

This section contains an overview of all the chapters. Chapters that basically always come to the exam are marked **bold**. The chapters with the exlamation mark (!) also come often, but it seems that per exam only one of them comes.
In the following some selected chapters are summarized, such that they can be used for the open book exam.

1. Introduction
   - Nothing of value for the exam
2. Bugs
   - Classes of Bugs, Fault, Error and Failure
   - Semantics and Formal Semantics (e.g. Operational Semantics and Small-Step Semantics)
3. **Assertions**
   - General overview of assertions, Assertions as specifications and Invariants (Inductive and Non-Inductive)
4. Testing
   - In science speak: Empirical falsification, Testing in the Development Cycle and Equivalence class and Boundary Testing
5. **Coverage**
   - Set of reachable states and Collecting Semantics
   - Control flow-based coverage (Statement, path, branch,...) and Data flow-based coverage
6. Test Case Generation (TCG)
   - Does not come in general
7. Logic!
   - Definition of Propositional and First-Order logic, Equi-Satisfiability, Tseitin Encoding and Binary Decision Trees
8. **Hoare**
   - The axioms of the Hoare Logic
9. Satisfiability Checking!
   - SAT Solver strategies and a bit of learning
10. Satisfiability Modulo Theories (SMT)!
    - Equality Logic for propositional, for uninterpreted functions and for constants
11. **Modelchecking**
    - Temporal Logics (Temporal Logics (CTL^*), Linear Temporal Logic (LTL) and Computational Tree Logic (CTL))
12. Spin
13. Bounded Model Checking (BMC)

2 Assertions

Assertions are partial specifications of the program written as a logic statement, usually a special form of propositional logic which includes arithmetics. They generally consist of pre- and postconditions. Preconditions must hold at the beginning of a procedure, therefore the callee (i.e. the procedure) can trust the assertion that it holds. In contrast to this the caller can trust the Postcondition. From this it follows that preconditions can be strengthened and post conditions can be weakened.
Important for this lecture is the concept of an **invariant assertion** and even more important a **loop-invariant assertion**. Simply put an invariant assertion must hold throughout the procedure or certain parts of the procedure. A loop-invariant assertion is an assertion which holds throughout all iterations of a loop, where it must hold at four positions: **Before the loop**, **at the beginning of each loop iteration**, **at the end of each loop iteration** and **after the loop** (see also the *Hoare* loop/while rule). I.e. from this it follows, that to check if such an invariant holds, one must check the four points.

The lecture distinguishes between two types of such loop-invariants, **inductive-invariants** and **non-inductive-invariants**. The difference between those two can best be explained by the following pseudo-code:

```
0 procedure() {
1     ...INIT...
2     {P}
3     while (B) {
4         {P/S}
5         ...Statements S..
6         {P}
7     }
8     {P}
9 }
```

One now assumes that we have an invariant P, which we check if it is an inductive-one. The invariant P is in line 4 altered as the statements S are incoperated, which leads to P/S. This process is similar to the way Hoare-Logic works in such a case, so we essentially apply assignment rules from the *Hoare* part.
With this information we can finally do our prove, where we must do things:

1. Base-Case: Check if the invariant holds in the first iteration of the loop.
2. Induction-Step: $(B \wedge P) \supset P/S$

An example for this is the following (where we assume the x and y can never overflow):

```
00 procedure(int x, int y) {
01     if (x > y) {
02         swap(x,y)
03     }
04     {P}
05     while (y > x) {
06         {P/S}
```

```
07          x = x + 1;
08          y = y - 1;
09          {P}
10      }
11      {P}
12 }
```

Now we assume that $P = (x - y) \leq 1$ and we prove that this is an inductive assertion:

1. Base-Case: Case distinction:
   - Case 0: $x \leq y$, therefore $(x - y) \leq 0 < 1$.
   - Case 1: $x > y$, therefore we swap the variables in line 02 and we know, that after the if $x < y$ holds, therefore $(x - y) < 0 < 1$.
2. Induction-Step:
   (a) We know: $B = y > x$
   (b) We know: $P = (x - y \leq 1)$
   (c) After applying P/S we get $P/S = (x + 1 - (y - 1)) \leq 1$
   (d) So: $((y > x) \wedge ((x - y) \leq 1)) \supset ((x + 1 - (y - 1)) \leq 1)$
   (e) Which is the same as: $((y > x) \wedge ((x - y) \leq 1)) \supset ((x - y)) \leq -1)$
   (f) This holds, as we know $y > x$.

How to prove that the assertion is non-inductive? For this one must argue first, that it is an assertion and always holds (without the formal part above). Then one can take the formal part above, show that the base case holds and write down the induction step. When doing the induction step one can come up with a counter example (remember: To prove the implication ("$\supset$") false, one must make the left side true and the right side false).

How to prove that the given assertion is not an invariant? For this one must come up with a counter example.

3 Coverage

Generally the lecture distinguishes between two kinds of coverage measures: **Control flow-based coverage** and **Data flow-based coverage**.

3.1 Control flow-based coverage

The lecture provides six different types of coverage metrics, where in in normal speak four of them are sometimes used interchangeably. Those four are the *path-*, *branch-*, *decision-* and *condition-coverage*. In contrast to this, the lecture provides a distinction between those four coverage metrics and in the following a summary of those four metrics is given.

3.1.1 Statement/basic block coverage

For achieving statement-coverage one must define statements. There exist many different definitions of statement, but generally for this lecture one can state that: Assignments (e.g. min = x;), conditions (e.g. x ¡ ¡), function calls (e.g. f(a)) and special lexems (such as *return*) can be seen as statements. Not considered as statements are for example curly braces, line breaks, etc.. Also not considered as a statement is a variable declaration in the function header.
Coverage is now measured as a percentage, of the amount of executed statements vs. the total amount of statements.

3.1.2 Path coverage

Path coverage and a path is a vague term, in this lecture it is used in a semantic sense, which means that one can see a path as a sequence of executed statements, which lead to the termination of the program. Paths differ, iff they execute different statements, a different amount of statements or the statements in a different order.
Branch coverage is now achieved when all such possible paths are executed, therefore in general it is not possible to achieve path coverage due to loops, as there are infinitely many paths in a loop.

3.1.3 Branch coverage

In this lecture a branch is defined syntactically, i.e. it can be defined as "A computer program construct in which one of two or more alternative sets of programs statements is selected for execution" or "A point in a computer program at which one of two or more alternative sets of program statements is selected for execution" (and this point is called a branch point).
To check if branch coverage exists one must identify all such branch points and for each such branch point each alternative set must be executed.

3.1.4 Decision coverage

A decision is a boolean expression, which is composed of conditions and zero or more boolean operators. I.e. in contrast to branch coverage decision coverage does not only take branch points into account, but also boolean statements in the normal code.
To achive decision coverage all decisions must evaluate once to true and once to false.

3.1.5 Condition coverage

As defined in decision coverage, as decision is *composed* of *conditions*. Therefore a *condition* is an atomic boolean expression. E.g. take the expression $C = (A \wedge B)$, then A and B are conditions, where $(A \wedge B)$ is a decision.
To achieve condition coverage one must evaluate each condition to true and false.

3.1.6 MC/DC coverage (Modified Condition/Decision Coverage)

For MC/DC coverage a strict definition exists:

1. Every entry and exit point in the program has to be visited
2. Every conditional statement (i.e. branchpoint) has to take all possible outcomes (i.e. branches)
3. Every non-constant Boolean expression has to evaluate at least once to 1 and at least once to 0
4. Every non-constant condition in a Boolean expression has to evaluate at least once to 1 and at least once to 0
5. Every non-constant condition in a Boolean expression has to affect that expressions's outcome independently

So generally one can state, that for decision coverage the requirements (1,2,3) must hold, for decision/condition coverage (1,2,3,4) and for MC/DC all 5.

3.2 Data flow-based coverage

Answers the question "how do values propagate through the program?". For this one must define three basic terms:

- **definition**: Assignment of a value to a variable (sometimes the vars. in the function header are included, sometimes not). Termed as $def(x)$, for variable x
- **use**: Where the value is used, can either be a **computational-use (c-use)**, i.e. right hand side of assignments (e.g. $a = b||c$, then it is a **c-use** for b and c) or a **predicate-use (p-use)**, i.e. used in conditional statements (near branchpoints, e.g. in $if(a||b)$ a and b are p-uses).
- **def-use-chain**: A cycle-free path, where the first statement is a definition (def) and the last statement a usage (use).
- **dpu(l,x)**: All predicate use locations for x, where there is a def-clear-path from an assignemnt to a predicate use.
- **dcu(l,x)**: All computational use locations for x, where there is a def-clear-path from an assignemnt to a computational use.

3.2.1 all-defs

Is achieved, iff there exists one path in the test-suite to some $l' \in (dpu(l,x) \cup dcu(l,x))$

3.2.2 all-c-uses

Is achieved, iff there exists one path to each $l' \in dcu(l, x)$ in the test suite.

3.2.3 all-p-uses

Is achieved, iff there exists one path to each $l' \in dpu(l, x)$ in the test suite.

3.2.4 all-c-uses/some-p-uses

Is achieved, iff *all-c-uses* and *all-defs* is achieved. I.e. iff there exists one path to each $l' \in dcu(l, x)$ and if $l' = \emptyset$, then at least one path to $l' \in dpu(l, x)$.

3.2.5 all-p-uses/some-c-uses

Is achieved, iff *all-p-uses* and *all-defs* is achieved. I.e. iff there exists one path to each $l' \in dpu(l, x)$ and if $l' = \emptyset$ then at least one path to $l' \in dcu(l, x)$.

3.2.6 all-uses

Is achieved, iff there exists one path to each node $l' \in (dpu(l, x) \cup dcu(l, x))$. I.e. iff *all-c-uses/some-p-uses* and *all-p-uses/some-c-uses* are achieved.

3.2.7 all-du-paths

Is achieved, iff all possible paths to each node $l' \in (dpu(l, x) \cup dcu(l, x))$ are contained in the test-suite.

3.2.8 all-paths

If all paths are covered?

4 Hoare

The Hoare calculus can be used to prove certain parts of a program. In general the hoare calculus can be written on top of a given program, where certain aspects are proven to be true by Axioms, which operate on Hoare Triples (which are $\{P\}C\{Q\}$, where P is the pre- and Q the post-condition; C are program statements).
To come up with a proof for given pre- and post-conditions (nearly all exam assignments are in this way) one must usually find a strong enough invariant. Finding such an invariant is more an art than a technique and is highly dependent on practice, therefore it is recommened to look at the Hoare sections of the old exams.
In the following the Axioms from the lecture are presented:

$$\frac{}{\{P\}skip\{P\}} \tag{1}$$

$$\frac{}{\{Q[E/x]\}x := E\{Q\}} \tag{2}$$

$$\frac{\{P\}C_1\{R\} \qquad \{R\}C_2\{Q\}}{\{P\}C_1; C_2\{Q\}} \tag{3}$$

$$\frac{\{P \wedge B\}C_1\{Q\} \qquad \{P \wedge \neg B\}C_2\{Q\}}{\{P\}if\ B\ then\ C_1\ else\ C_2\{Q\}} \tag{4}$$

$$\frac{P' \supset P \qquad \{P\}S\{Q\} \qquad Q \supset Q'}{\{P'\}S\{Q'\}} \tag{5}$$

$$\frac{\{P \wedge B\}C\{P\}}{\{P\}while\ B\ do\ C\{\neg B \wedge P\}} \tag{6}$$

- Formula 1 is called the *Skip* rule.
- Formula 2 is called the *Assignment* rule.
- Formula 3 is called the *Composition* rule.
- Formula 4 is called the *Condition* rule.
- Formula 5 is called the *Consequence* rule.
- Formula 6 is called the *While* rule.

5 Modelchecking

In Modelchecking the main topic is temporal logic. For this to work out one must first define a transition system and Kripke structures (so i.e. Modal-Logic stuff):

Theorem 1. *The triple $\langle S, T, I\rangle$ is a Finite-State Transition System, iff:*

- *S is a finite set of states*
- *I is the set of initial states, so $I \subseteq S$*
- *T denotes the transitions, so formally $T \subseteq SxS$. Put simpler, $T(S0, S1)$ denotes that a transition from State $S0$ to $S1$ is possible.*

This is not enough, we need to assign truth values - here comes the Kripke structure into play:

Theorem 2. *A quadruple $\langle S, T, I, L\rangle$ is a Kripke structure, iff*

- *$\langle S, T, I\rangle$ comprises a finite-state transition system.*
- *L is a labelling function $S \to 2^{AP}$, where AP denotes atomic propositions. I.e. the function L assigns truth values to atomic expressions per state.*

As we are here in the realm of modal logic, we write $s \models F$, iff F holds in state s:

$$s \models p \quad \Leftrightarrow \quad p \in L(s)$$
$$s \models \neg F \quad \Leftrightarrow \quad s \not\models F$$
$$s \models F_1 \vee F_2 \quad \Leftrightarrow \quad s \models F_1 \text{ or } s \models F_2$$
$$s \models F_1 \wedge F_2 \quad \Leftrightarrow \quad s \models F_1 \text{ and } s \models F_2$$

5.1 Temporal Logics

With this information one can define temporal logics, for which one must first define a path. A path is an infinite sequence of states with $T(s_i, s_{i+1})$. Generally a path is denoted as π and a certain state in the path as π^{State}.
Therefore the above define syntax and semantic is slightly changed to accomodate the needs of the paths, therefore we now evaluate formulas on Kripke Structures and Paths, i.e.: $M, \pi \models \phi$ means, that ϕ holds for the given Kripke Structure M and the given path π.
With these definitions in mind one can now reason about certain paths, where five operators are defined:

Theorem 3. *The unary operator* ***X*** *denotes, that something holds in the next step. Formally:*

- $M, \pi \models \mathbf{X}\phi \quad iff \quad M, \pi^1 \models \phi$

Theorem 4. *The unary operator* ***F*** *denotes that something must hold eventually. Formally:*

- $M, \pi \models \mathbf{F}\phi \quad iff \quad M, \pi^k \models \phi$ *for some* $k \geq 0$

Theorem 5. *The unary operator* ***G*** *denotes that something must hold globally, i.e. forever. Formally:*

- $M, \pi \models \mathbf{G}\phi \quad iff \quad M, \pi^k \models \phi$ *for all* $k \geq 0$

Theorem 6. *The binary operator* ***U*** *denotes that something prior to* ***U*** *holds until something after* ***U*** *holds. Formally:*

- $M, \pi \models \phi_1 \mathbf{U} \phi_2 \quad$ *iff there is a* $k \geq 0$ *such that* $\quad M, \pi^k \models \phi_2$ *and* $M, \pi^i \models \phi_1$ *for* $0 \leq i < k$.

Importantly ϕ_2 *must happen eventually and further for this operator it also holds, if* ϕ_2 *directly holds and* ϕ_1 *never holds.*

Theorem 7. *The binary operator* ***R*** *denotes that something prior to* ***R*** *releases something after* ***R*** *(inverse formulation to* ***U****). Formally:*

- $M, \pi \models \phi_2 \mathbf{R} \phi_1 \quad$ *iff one of the two following conditions hold:*
  1. $\exists k \geq 0$ *such that* $\quad M, \pi^k \models \phi_2$ *and* $M, \pi^i \models \phi_1$ *for* $0 \leq i < k$.
  2. $M, \pi^i \models \phi_1$ *for* $j \geq 0$

With these theorems in mind one can now define the **E** and **A** operators:

Theorem 8. **E** *denotes that there exists a path where something will hold (pretty much the* $\exists$ *Operator of First-Order-Logic). Formally:*

- $M, s \models \mathbf{E}\phi \quad iff \quad \exists \pi$ *starting at* s *such that* $M, \pi \models \phi$.

Theorem 9. **A** *denotes that a property must hold for all possible paths (pretty much the* $\forall$ *operator of First-Order-Logic). Formally:*

- $M, s \models \mathbf{E}\phi \quad iff \quad \exists \pi$ *starting at* s *such that* $M, \pi \models \phi$.

5.2 Families of Temporal Logics

- $CTL*$ (Computation Tree Logic*) is the above defined logic.
- CTL (Computation Tree Logic) is a real subset of $CTL*$, as every operator must be immediatelly be preceded with either **A** or **E**.
- LTL (Linear Temporal Logic) is a real subset of $CTL*$, as every formula must start with **A**.

5.3 Model Checking for CTL

We learning an algorithm for model checking any CTL formula. For this one must know that every one of the 10 basic CTL operators can be expressed by $\mathbf{EX}, \mathbf{EG}$ and $\mathbf{EU}$. The 10 basic operators are:

	X	**F**	**G**	**U**	**R**
A	**AX**	**AF**	**AG**	**AU**	**AR**
E	**EX**	**EF**	**EG**	**EU**	**ER**

$$\mathbf{AX}\phi \equiv \neg\mathbf{EX}(\neg\phi)$$
$$\mathbf{AG}\phi \equiv \neg\mathbf{EF}(\neg\phi)$$
$$\mathbf{AF}\phi \equiv \neg\mathbf{EG}(\neg\phi)$$
$$\mathbf{A}(\phi_1\mathbf{R}\phi_2) \equiv \neg\mathbf{E}(\neg\phi_1\mathbf{U}\neg\phi_2)$$
$$\mathbf{A}(\phi_1\mathbf{U}\phi_2) \equiv \neg\mathbf{E}(\neg\phi_2\mathbf{U}(\neg\phi_1 \wedge \neg\phi_2)) \wedge \neg\mathbf{EG}\neg\phi_2$$
$$\mathbf{EF}\phi \equiv \mathbf{E}(true\ \mathbf{U}\phi)$$
$$\mathbf{E}(\phi_1\mathbf{R}\phi_2) \equiv \neg\mathbf{A}(\neg\phi_1\mathbf{U}\neg\phi_2)$$

The algorithm works as follows: Starting with the initial formula Φ one translates the formula with the above defined conversions into the form where only the operators $\mathbf{EX}, \mathbf{EG}$ and $\mathbf{EU}$ exists. From this state one applies the tableaux algorithm:

Start with $\mathcal{T}_\varphi = \{\varphi\}$, apply the rules until no applicable rule left:

1. if $\psi' \wedge \psi'' \in \mathcal{T}_\varphi$, then $\mathcal{T}_\varphi := \{\psi', \psi''\} \cup \mathcal{T}_\varphi$
2. if $\neg\psi \in \mathcal{T}_\varphi$, then $\mathcal{T}_\varphi := \{\psi\} \cup \mathcal{T}_\varphi$
3. if $\mathbf{EX}\,\psi \in \mathcal{T}_\varphi$, then $\mathcal{T}_\varphi := \{\psi\} \cup \mathcal{T}_\varphi$
4. if $\mathbf{EG}\,\psi \in \mathcal{T}_\varphi$, then $\mathcal{T}_\varphi := \{\psi\} \cup \mathcal{T}_\varphi$
5. if $\psi'\,\mathbf{EU}\,\psi'' \in \mathcal{T}_\varphi$, then $\mathcal{T}_\varphi := \{\psi', \psi''\} \cup \mathcal{T}_\varphi$

$$[\![p]\!] = \{s \in S \mid p \in L(s)\} \text{ for } p \in AP$$

$$[\![\psi' \wedge \psi'']\!] = [\![\psi']\!] \cap [\![\psi'']\!] \text{ and } [\![\neg\psi]\!] = S \setminus [\![\psi]\!]$$

procedure compEX(ψ) {
  $[\![\mathbf{EX}\psi]\!] := \{s \in S \mid \exists s' \in [\![\psi]\!] \text{ and } (s, s') \in T\}$
}

procedure compEG(ψ) {
  $Z' := [\![\psi]\!]$
  do
    $Z := Z'$
    $Z' := \{s \in [\![\psi]\!] \mid \exists s' \in Z \text{ and } (s, s') \in T\}$
  while $Z \neq Z'$

  $[\![\mathbf{EG}\,\psi]\!] := Z$
}

procedure compEU(ψ', ψ'') {
  $Z := \emptyset$
  $Z' := [\![\psi'']\!]$
  while $Z \neq Z'$
    $Z := Z'$
    $Z' := Z \cup \{s \in [\![\psi']\!] \mid \exists s' \in Z \text{ and } (s, s') \in T\}$

  $[\![\psi' \,\mathbf{EU}\, \psi'']\!] := Z$
}