

# Software Verification and Validation

---

Dr. Martin Wittiger

WS 2021/22

Hochschule für Technik Stuttgart

## Presentation Contents

**Introduction:** Testing Myths · Planning Phase · Analysis Phase · Design Phase I · Implementation Phase I · Testing Phase · Maintenance Phase ·  $\Omega$

**Users and Bugs:** First Bug · Standard Names for Bugs · Safety & Security ·  $\Omega$

**Probability:** Random Variables · Central Limit Theorem · Expected Value · Law of Large Numbers · Variance · Standard Deviation ·  $\Omega$

**Downtimes:** Reliability · Availability · Serviceability/Maintainability ·  $\Omega$

**Programs & Functions:** LOOP Programs: Syntax · WHILE Programs: Syntax · Primitive Recursive Functions · From Primitive to  $\mu$ -Recursive ·  $\Omega$

**Proof by Induction:** Full Induction · Complete Induction · Structural Induction · Pitfalls ·  $\Omega$

**Computational Power:** Ackermann Function · Ackermann Function is not Primitive Recursive I ·  $\Omega$

**Proof of Majorization:** Helpful Lemmas · Collatz Conjecture · Proof of the Majorization Theorem ·  $\Omega$

**Decidability:** Turing Machines · Halting Problem · Reduction I · Implications for Verification ·  $\Omega$

**Software Test:** Fuzz Testing · Systematic Testing · Code Coverage ·  $\Omega$

# Acknowledgements

This lecture course including slides and exercise sheets were originally designed and written by

Prof. Dr. Peter Heusch

Exercise sheets also contain material by

Torsten Görg

Dr. Timm Felden

Recently received my doctors' degree from the University of Stuttgart

Research Reliability, Safety and Security of embedded automotive systems (static analysis)

Teach programming language use and design including compilers

Work at Mercedes-Benz/Daimler as a senior cyber security architect

We will play a little game

On after another everyone

- tells us their name,
- states three things about themselves,
- answers the question of his predecessor and
- asks his sucessor a creative question

- We meet roughly fortnightly
- Initially we meet in person, this might change later
- We will usually do 2 h lecture and then 2 h exercise classes (discussion of assignments)
- We will deviate from this plan in the beginning, and maybe towards the end

I will upload class assignments to moodle

Assignments are due for the next exercise class

No hand-in/marking of assignments, but discussion

Your work must be your work! (Karl-Theodor zu Guttenberg, Annette Schavan, Silvana Koch-Mehrin—What do they have in common?)

Discuss with others but code for your own

Make sure you do the assignments



Introduction to SVV

Terminology—Reliability, Availability, Serviceability, Safety, Security

Probability

LOOP Programs, WHILE Programs and Recursive Functions

Undecidability and Incompleteness

Validation, Verification: Methods & Reasons

# Introduction

---

Does the product conform to its specification?

Stepwise refinement, starts with the requirements and ends with the executable code

Can be partially automated, but nevertheless needs additional human support

*Do we build the product right?*

Does the product meet the requirements of its intended users?

Mostly written as plain text in documents

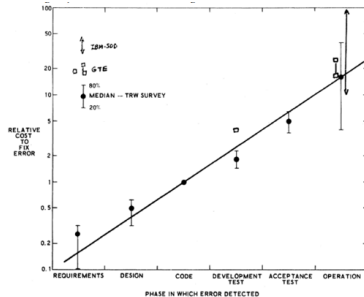
Can unfortunately not be automated

*Do we build the right product?*

# Reasons for Verification & Validation I

Defective or non-functional software is widely accepted (*unfortunately*)

According to Boehm, 1976 cost of software defects increases exponentially with the phase where the defect is found



Source: Boehm, 1976

Some systems simply cannot tolerate defective or dysfunctional software:

- Flight management systems
- Railway control systems
- Nuclear power plants
- Autonomous cars

The human factor must be considered:

- Human perception
- Misunderstanding
- Sloppiness

## Interlude: Capability Maturity

The Capability Maturity Model (CMM) describes maturity of (software development) processes. It features:<sup>1</sup>

- Initial: Ad hoc, typically undocumented, reactive
- Repeatable: Somewhat consistent results, process discipline not rigorous
- Defined: Standardization and some optimization happens
- Managed (Capable): Refining and adapting quantitatively
- Optimizing (Efficient): Deliberate process optimization

Newer replacement: Capability Maturity Model Integration (CMMI)

---

<sup>1</sup>See: [https://en.wikipedia.org/w/index.php?title=Capability\\_Maturity\\_Model&oldid=940969931](https://en.wikipedia.org/w/index.php?title=Capability_Maturity_Model&oldid=940969931)

CMMI level 3 guarantees less than 5 defects per 1000 lines of code

Some safety-critical embedded system in cars have about 55 000 lines of code



Debugging is a form of testing: No! Debugging tracks down a failure to the corresponding defect in the source code, while testing shows that (at least) the given test cases are handled correctly

Testers are solely responsible for quality: No! Testers are responsible for writing tests according to the requirements and to check the software by running the tests

Testing is expensive: No, untested software at the customer site is much more expensive

However, car manufactures spend hundreds of M€ on SVV of embedded software

Include stakeholders:

- Customer
- End users
- Product Manager
- Project Manager
- Development
- Operations
- Subcontractors

One must identify all stakeholders

Hence V & V must be planned before the development process is started

Some stakeholders might be opposed to the project

Some stakeholders might be unavailable

V & V must be carried throughout the whole development process:

1. Planning
2. Analysis
3. Design
4. Implementation
5. Testing
6. Maintenance

Usually V & V efforts are focused on phases 2–5, but this is wrong!

Establish the rules necessary to achieve the desired results

Identify stakeholders to be included

Identify methods to be used for establishing program correctness:

- Proving
- Review
- Testing
- Statistics
- Static analysis

Identify criteria to distinguish between success and failure

Estimate V & V cost!

Quality can not be tested, it must be produced!

Ultimately we expect every program to be correct

For some programs the cost of incorrectness is especially high:

- Fatalities (medicine, aviation, railway, space flight or cars software)
- Damage (robots, buildings, cars, ships)
- Wealth (banking, insurance, complex production processes)

Often stakeholders are not properly identified when a project starts, for a variety of reasons

Some stakeholders are not available at all

Stakeholders usually have different, sometimes divergent interests in projects

Stakeholders might not express their real opinions about a project

Stakeholders might be unable to communicate in “expected” language

Often stakeholders in systems are identified from a single point of view, e.g. the procurement department

However, there are a lot of people that might be stakeholder outside procurement

These people are usually unable to contribute to the development process except they might know about the cost

Further problems can arise if stakeholders express their personal opinion, not the opinion of the group they are representing!

Sometimes the availability of stakeholders is severely restricted:

End users for a web service, which is offered worldwide

Malevolent users that would like to misuse the program under construction

Competitors on the market that offer software solving the same problems

End users might be grouped in an unforeseen way, so one user does not cover all use cases



Stakeholders can have different, even divergent interests:

- Project management likes an “easy” implementation
- Procurement is interested in a cheap product
- Marketing is interested in a fancy design
- Developers prefer “sophisticated” code
- End users might fear unemployment

V & V must take all these different interests into account in order to deliver a good compromise

Sometimes a bit of psychology is necessary to achieve this

Sometimes stakeholders do not tell their true opinion about a software:

- Employees fear unemployment
- Operations prefer another product
- Management believes the invest in the software is too high

Some stakeholders simply do not understand the technical language used:

- Missing education
- Missing knowledge of technical terms
- Misunderstanding terms

Demonstrate correctness by mathematical proofs

Often, program proving specific logical calculi are used in conjunction with automatic provers

Note: Automatic provers help, but usually do not do the work for you

They can follow a proving path, but it is difficult to come up with one themselves

Create a comprehensive set of tests that establish sufficient confidence in the software

Many categories of tests exist

- Unit, Integration, System and Acceptance Tests
- Smoke Test
- Automated, manual
- Hardware in the Loop, Software in the Loop, Model in the Loop
- Penetration Tests (Security)

Analyse all artefacts during the construction process by multiple persons

Variant: Peer code review

Variant: Formal review (See Ludewig and Lichter 2007)

Count the number of identified errors to estimate the total count

Static analysis tool can demonstrate properties of programs without running them

Those properties can imply absence of certain errors

Similar concept: Abstract interpretation

Also, dynamic analysis: Sometimes helps to focus static analysis

Properly debugging software ...

**No!**

*Debugging cannot be used to establish correctness of a program!*



One must ensure that all documents are written such that V & V is possible in later steps, by checking that requirements are:

- Clear
- Simple
- Verifiable
- Complete
- Consistent
- Unambiguous
- Free of interpretation
- Correctly located
- Positively formulated
- Free of non-verifiable terms
- Free of unused functionality
- Realistic
- Traceable to implementation
- Unique (non-redundant)
- Explicit about open points (TBD, TBR)

During the design phase, V & V must take care that all requirements are truly transferred to the design artefacts:

- Every requirement must be fulfilled in the design
- Every design decision must be driven by some requirement
- Avoid overly complicated designs

Test plans must be created in accordance with the requirements

Think: Unit Tests, Integration Tests, System Tests, ...

Define which data passes which interface:

- Consistency with the requirements
- Guarantee testability

Check that design decisions are consistent with requirements

Select appropriate tools to support the build process during implementation

- Compiler (Compiler-flags, Warnings, Think: **-Wall -Werror**)
- Maketool
- Style-checker
- Smell-Detection, Clone detection, Static analysis

During the implementation phase, V & V must ensure that all design artefacts as well as all requirements are truly transferred to the implementation artefacts:

- Every design artefact must be implemented
- There must be no implementation artefact without a corresponding design artefact

In parallel V & V must check that all tests are coded according to the requirements document

During implementation many tools are used

Furthermore, V & V checks that reviews, audits, etc. are carried out as specified

If errors are found during the implementation phase, V & V must investigate the reason for the error:

- Wrong requirements?
- Wrong design decisions?
- Wrong implementation?

If outcomes of previous steps are affected, care must be taken to analyse the effects of all changes, not only locally but globally

Both the investigation and the change analysis are error prone

During the testing phase V & V must check that the testing outcomes are consistent with the requirements and design artefacts

If a test fails, V & V must determine the source:

- Wrong implementation, i. e. preconditions of the test are met, but postconditions are not met
- Wrong specification (or wrong test), i. e. preconditions of the test are not met

In any case V & V must decide what is to be changed

Studies show that more than 70 % of the total life-cycle cost of software is spent for operation and maintenance

Maintenance by itself is not a sign of defective or dysfunctional software but often necessary because of:

- New legal regulations
- Changing environment

Care must be taken that during maintenance V & V controls all effects of potential changes

## Summary

---

Verification and Validation play an important role during the whole software development process, from planning to maintenance

Validation guarantees that the system indeed meets all the requirements of all stakeholders: “Are we building the right product?”

Verification guarantees that every phase of the development process produces exactly those artefacts that were demanded by its inputs: “Are we building the product right?”



## Users and Bugs

---

Human errors cause many failures. Hence, the “failing” human must be taken into account

Failures can be caused by simple sloppiness, by misunderstanding or by misinterpretation:

- Hotel safes usually have a master code allowing the hotel to open the safe in case the guest leaves the room with the safe locked
- Early Airbus 320 planes showed the sink rate as 33 (100 ft per minute) or 3.3 (degree), these differ by a factor of 5!
- In March 1977 a pilot misunderstood “Ready for take-off?” as “Cleared for take-off!”, and killed 583 people during the Tenerife airport disaster

Human errors can be corrected by changes in the technical system:

- A hotel safe could be intentionally dysfunctional as long as the default unlocking code is not changed
- The Airbus 320 shows sink rate either as 3.3 degree or 3300 feet per minute
- The word “take-off” was completely removed from radio communication except in “Cleared for take-off!”
- TCAS checks whether both pilots follow its instructions and corrects its instruction for the “other” pilot if one pilot does not (Überlingen crash)

The human role is very difficult to formalize, it can depend on the cultural background:

- Pilots from the former USSR tend to follow the air traffic controller, if he gives an instruction
- Pilots from the rest of the world tend to follow the TCAS, since this is the official rule

As a consequence care must be taken to perform tests under stress and with different people

Note: Western, Educated, Industrialized, Rich, Democratic → *WEIRD* psychological results

In everyday language computer scientists talk about bugs and errors in software, but this is not unambiguous, since it can mean:

- A place in the code where some instruction does not do what it should do
- Data anomalies, arising from executing faulty code
- Externally visible signs of data anomalies, from nonsense data to a program crash & dump

First actual case of bug being found was 1947!

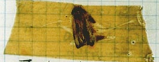
# First Bug

First actual case of bug being found was 1947!

9/9

0800 Action started  
1000 " stopped - action ✓ { 1.2700 9.037 847 025  
13'40 (032) MP-MC 2.13047645 9.037 846 985 correct  
033 PRO 2 2.13047645 4.615925059(-2)  
convert 2.13047645  
Relays 6-2 in 033 failed speed test  
in relay 11.00 test.

1100 Started Cosine Tape (Sine check)  
1525 Started Multi-Adder Test.

1545  Relay #70 Panel F  
(moth) in relay.

First actual case of bug being found.  
1650 Action started.  
1700 closed down.

Relay 3145  
Relay 3370

Picture: Public Domain

IEEE-Nomenclature:

- Errors are human actions leaving a technical system in a faulty state
- Defects are faulty (internal) states of a technical system that can impair the required functionality of the system and may lead to a failure
- Failures are externally visible dysfunctions of technical systems

The special problem with software is that it may contain lots of errors and even defects that never show up *until some day* ...

Zeller proposes a 4-stage naming for bugs:

- Defects are pieces of code that can modify data unwantedly
- Infections are the modified data that have been caused by defects
- Infection propagation denotes the chain that links infections to failures or in some cases masks the infections
- Failures are externally visible dysfunctions of the program

To debug a program it is essential to find a path from the failure back to the defect



70 % of the software life cycle cost is spent after the initial release, this includes

- Cost of defect correction for the initial release
- Cost of defect correction for later releases
- Cost of software operations (which is out of scope for this lecture)

As a general rule, the cost of corrections grows exponentially with the distance between the time when the error has been made and the time when the error has been detected

Cost of error correction depends on “phase difference”:

- The error is made during implementation & found while testing  
→ Correction is cheap
- The error is made during requirements & found while integrating  
→ Correction is expensive

The most expensive errors are those being “found” at the customer site

Consequently, software maintenance due to changing requirements produces the most expensive “errors”

# What is Error Cost?

Insurances usually have three forms of cost:

- Pecuniary loss (losing money)
- Physical damage (factory burns down, car engine breaks)
- Personal injury, fatalities

A fourth generally non-quantifiable form is loss of reputation, e.g. Toll Collect

One problem is the estimation of personal injury/fatalities in money, so these two risks should be accounted differently, e.g. fatalities in Mikromort

Injuries in compensation or medication/medical treatment cost

A micromort is a unit of risk defined as  $1/1\,000\,000$  chance of death<sup>2</sup>

The Value of a Statistical Life (VSL) is the amount an organization is willing to spend to statistically avert a fatality

Some values quoted for the VSL lie between 1 Million and 50 Million EUR

Details in both are intricate. Here a very basic understanding suffices

---

<sup>2</sup>Worthwhile reading: <https://en.wikipedia.org/wiki/Micromort>

One way to determine the maximum acceptable damage is to compare the damage of the system without software, to the system with software

- In autonomous driving, failure of the software likely will immediately cause damage
- For airbag control the software needs to work (actively) only in very rare situations (car accidents)

Usually software needs to be safe and secure:

- Safe software will not harm its environment
- Secure software cannot be harmed by the environment

Security and safety are intimately linked

- Safety without security is impossible
- Security without safety is unethical

For unsafe software the risk can be estimated w.r.t. the damage

For insecure software the risk analysis must also include the possible gain

Safety: “Relative freedom from danger, risk, or threat of harm, injury, or loss to personnel and/or property”<sup>3</sup>

Security: “The extent to which a computer system is protected from data corruption, destruction, interception, loss, or unauthorized access.”<sup>4</sup>

Safety is protection against random incidents

Security is protection against intended, deliberate or planned acts

Note that here security requires a actor, often called “threat actor”, a thinking foe

---

<sup>3</sup>Source: <http://www.businessdictionary.com/definition/safety.html>

<sup>4</sup>Source: <http://www.businessdictionary.com/definition/security.html>

## Summary

---

Standard Names for Bugs

Always consider the human role:

- Software used by benevolent users
- Software used by malevolent users

Definitions of Safety & Security



# Probability

---

Reliability and availability are defined based on probabilistic notions

Understanding Probability is important for everyone working in Verification and Validation

A random variable is a function that maps the outcome of an unpredictable experiment to a (number) value

The behaviour of these things can be modelled by a random variable:

- A coin shows head or tail
- A die shows 1 – 6
- A bulb burns for 0 – 100 000 hours before failing

Every single evaluation of a random variable can yield any possible value, however not all values are drawn with the same probability

# Discrete vs. Continuous Probability

There are two distinct but similar fields in probability theory:

Discrete and continuous probability

In discrete probability random variables have a finite or countably-infinite number of possible values. One often *sums* over those values

In continuous probability random variables have an uncountably-infinite number of possible values (from an interval of the reals). One often *integrates* over those values

In discrete probability an outcome with 0 probability never occurs

In continuous probability all outcomes have 0 probability

# Discrete Distributions with Examples

Bernoulli distribution: Coin flip, child is male

Discrete uniform distribution: Fair die, coin flip

Binomial distribution: Number of male children out of  $n$

# Continuous Distributions with Examples

Continuous uniform distribution: Pseudo randoms over interval  $[1; 0[$

Poisson distribution: Births per hour

Exponential distribution: Time between two births

Gamma distribution: Time until  $n$  children born

Gauss distribution (also called normal distribution): Errors of measurement

The Gauss distribution is the mother of all distributions. For any distribution the sum of  $n$  events is almost normally distributed for large  $n$ <sup>5</sup>

---

<sup>5</sup>Some terms and conditions apply

## Expected Value

Intuition: The expected value of a random variable is the “average” value after repeating the experiment “often enough”

For a discrete random variable  $X$  with values  $x_1, x_2, \dots, x_n$  with probabilities  $p_1, p_2, \dots, p_n$  holds:

$$\mathbb{E}[X] = \sum_{i=1}^n x_i p_i$$

For a continuous random variable  $X$  with density function  $f$  holds:

$$\mathbb{E}[X] = \int_{-\infty}^{\infty} x f(x) dx$$



The law of large numbers says that for every random variable  $X$  the average of  $n$  experiments converges to the expected value  $E[X]$  as  $n$  tends to infinity

# Variance

While the expected value denotes the mean value, the variance denotes the mean squared deviation from the expected value

$$\text{Var}[X] = E[(X - E[X])^2] = E[X^2] - E[X]^2$$

(The second part can make calculations easier)

The variance describes—more or less—the width of a probability distribution:

- A fair die with only 3 and 4 on its sides has the same expected value as a regular fair die, 3.5
- The regular die has a variance of 35/12
- The 3/4-die has a variance of 3/12

While the expected value denotes the mean value, the variance denotes (something like) the mean deviation from the expected value

One advantage is that units are not squared, if a random variable describes time in seconds, its variance would be in seconds times seconds, a meaningless unit

We define the standard deviation like this:

$$\text{SD}[X] = \sqrt{\text{Var}[X]}$$

The two dice from the last slide have standard deviations of ca. 1.7 and 0.5

If the values of a random variable  $X$  are known to be  $x_1, x_2, \dots, x_n$  with probabilities  $p_1, p_2, \dots, p_n$  we make these calculations:

$$\mu = E[X] = \sum_{i=1}^n x_i p_i$$

$$\text{Var}[X] = E[(X - \mu)^2] = \sum_{i=1}^n (p_i (x_i - \mu)^2)$$

$$\text{SD}[X] = \sqrt{\text{Var}[X]}$$

Note however, that this algorithm is not in general numerically stable, better alternatives are known

The standard deviation is just the square root of the variance. Calculate it this way and do not try to be too clever!

# Independence of Random Variables

Two random variables are independent iff the outcome of one tells nothing about the outcome of the other

In reality random variables are often not independent. Bayes's rule covers those cases

## Combining Expected Values—Addition

For any two random variables  $X$  and  $Y$  the expected value of their sum is the sum of their expected values:

$$E[X + Y] = E[X] + E[Y]$$

This holds even if the variables are completely dependent, e. g. the chance that throwing a die yields an odd result ( $X$ ) or an even result ( $Y$ ) is 0.5, and the expected value of  $X + Y$  is 1

## Combining Expected Values—Multiplication

With multiplication things are not as clear cut

For independent random variables the expected value of the product is the product of the expected values:

$$E[X \cdot Y] = E[X] \cdot E[Y]$$

For dependent random variables the product may even be zero, see the example on the last slide



# Combining Variances

For  $n$  uncorrelated random variables  $X_1, X_2, \dots, X_n$  the variance of the sum is given as

$$\text{Var} \left[ \sum_{i=1}^n X_i \right] = \sum_{i=1}^n \text{Var} [X_i]$$

If the variables are correlated the notion of covariance comes up:

For  $n$  correlated random variables the variance of the sum is given as

$$\text{Var} \left[ \sum_{i=1}^n X_i \right] = \sum_{i=1}^n \text{Var} [X_i] + 2 \sum_{1 \leq i < j \leq n} \text{Cov} [X_i, X_j]$$

For independent random variables the covariance is 0

# Combining Uniform Distributions I

Let  $A$  be uniformly distributed from 0 to 6. Then  $\text{Var}[A] = 4$

Let  $B$  be uniformly distributed from 0 to 60. Then  $\text{Var}[B] = 310$

Let  $C = 10A$ . Then  $\text{Var}[C] = 400$

Let  $D_1, D_2, \dots, D_{10}$  be 10 *independent* random variables distributed just like  $A$ .  
Then  $\text{Var}[D_1 + D_2 + \dots + D_{10}] = 40$



What does this teach us?

Enlarging the range by the factor of 10 increases the variance by a factor of (approximately) 100

However, the variance of the sum of 10 uncorrelated distributions of Type A is only 40, 10 times as large

This sum is not distributed uniformly, it is “almost” normally distributed

## Summary

---

Random variables, distributions both discrete and continuous

Expected value, Variance and Standard Deviation of random variables

Central Limit Theorem, Law of Large Numbers and Independence

## Downtimes

---

Reliability denotes the probability that a given system fulfils its required functions during a certain amount of time

The inverse property is the failure probability that denotes the probability that the system fails its required functionality

Reliability is measured as MTBF (Mean Time Between Failure) for repairable items (cars) and MTTF (Mean Time To Fail) for items that cannot be repaired (light bulbs)

Availability denotes the percentage that a given system fulfils its required functions during a certain amount of time

The inverse property is the unavailability that denotes the time when the system is not functional during that time

Availability is measured in percent by computing

$$\text{Availability} = \frac{\text{Uptime}}{\text{Uptime} + \text{Downtime}}$$

Often availability is measured as “number of nines”:

90 %, 99 %, 99.9 %, 99.99 %, 99.999 %

Serviceability or maintainability denotes the simplicity and speed of repairing a technical system

Usually serviceability can be increased by

- Predictive maintenance, by replacing parts before they become defective and create damage
- Servicing by replacement and repairing elsewhere
- Precise configuration management and logging

Serviceability is measured as MTTR (Mean Time To Repair) or MDT (Mean Down Time)



For technical systems without redundancy the expected failure probabilities simply add up:

- Assume that we have a million LEDs forming a display and the average LED fails with a rate of  $1 \cdot 10^{-3}$  in one year
- This results in an average daily failure rate of approximately  $2.74 \cdot 10^{-6}$
- Hence on the average there will be about 2.7 defective LEDs per day
- So we can only expect our display to work perfectly for some 8.8 hours

# Predicting MTBF and MTTF—With Redundancy

## A Very Real Example

I recently bought a monitor and I found this in the user manual:

maximize the life of the monitor, we recommend the monitor be turned off periodically.

Periodic cleaning is recommended to keep the monitor looking new and to prolong its operation lifetime (refer to “Cleaning” (page 4)).

The LCD panel is manufactured using high-precision technology. Although, missing pixels or lit pixels may appear on the LCD panel, this is not a malfunction. Percentage of effective dots: 99.9994 % or higher.

The backlight of the LCD panel has a fixed lifetime. Depending on the usage pattern, such as usage for long continuous periods, the lifespan of the backlight may run out sooner, requiring replacement. When the screen becomes dark or begins to flicker, please contact your local EIZO representative.

Excerpt: Eizo User's Manual EV3285

The display's resolution is 3840 x 2160 and every pixel consist of three dots. If all dots are broken independently with probability 0.999 994, what is the probability the whole display works fine?

## Summary

---

Reliability, availability and serviceability

MTBF (Mean Time Between Failures), MTTF (Mean Time To Fail) and MTTR (Mean Time To Repair)

Effects of redundancy

## Programs & Functions

---

Computability as term is inherently imprecise, so mathematicians and computer scientists use mechanical definitions, hence a function is computable if its values can be computed by:

- Ada, Java, C++, Prolog, BASIC, FORTRAN, Assembler, ...
- Turing or register machines
- LOOP- or WHILE- or GOTO-programs
- Recursive functions

All functions that are intuitively computable can be computed by one of the mechanisms given above (Church' thesis)

Syntactically a LOOP program consists of the following statements:

- Simple statements:  $x_i := x_j$ ,  $x_i := x_j - c$ ,  $x_i := x_j + c$ ,
- Composition: If P and Q are LOOP programs, then  $P ; Q$  is a LOOP program
- Loops: If P is a LOOP program, then **LOOP**  $x_i$  **DO** P **END** is a LOOP program

The numerical constants  $c$  are natural numbers

We use an infinite pool of variables  $x_i$ , where  $i$  is a natural number. When convenient we can also use **a**, **b**, **r**, **s**, **t**, **y**, **z** or similar as fresh variables

Initial conditions:

- $x_0$  is the return value. It is initially 0
- $x_1 \dots x_n$  are the parameters
- all other variables are initially 0

Statements act on the global state



Simple statements:

- $x_i := x_j$  sets the value of  $x_i$  to the value of  $x_j$
- $x_i := x_j - c$  sets the value of  $x_i$  to the value of  $x_j - c$  or to 0, whichever is larger
- $x_i := x_j + c$  sets the value of  $x_i$  to the value of  $x_j + c$

Note that this ensures that the values of the variables are natural numbers

Composition and loop statements:

- The statement `P ; Q` executes first `P`, then `Q`
- `LOOP xi DO P END` executes `P` exactly `xi` times. If `xi` is modified within `P`, this has no influence on the number of repetitions

Note that For loops in Ada or Python are equivalent to LOOP programs, but `for(a; b; c)` in C, C++, Java is different

A WHILE program consists of the following statements:

- Simple statements like those of LOOP programs
- Composition: If P and Q are WHILE programs, then `P ; Q` is a WHILE program
- Loops: If P is a WHILE program, then `LOOP  $x_i$  DO P END` is a WHILE program
- If P is a WHILE program, then `WHILE  $x_i \neq 0$  DO P END` is a WHILE program

Note that WHILE programs are a superset of the LOOP programs

WHILE programs have the same semantics as LOOP programs

The statement **WHILE**  $x_i \neq 0$  **DO** P **END** executes P until  $x_i$  becomes 0.  $x_i$  may be modified inside P

While loops in all(?) programming languages as well as **for(a; b; c)** in C, C++, Java are equivalent to WHILE Programs

Primitive recursive functions are those functions that can be defined from:

- Zero-functions  $z_k(x_1, \dots, x_k) = 0$
- Constant functions  $c_k^i(x_1, \dots, x_k) = i$
- Projections  $p_k^j(x_1, \dots, x_k) = x_j$
- Successor function  $s(x) = x + 1$
- Composition of an  $m$ -ary primitive recursive function  $f$  with  $m$   $k$ -ary primitive recursive functions  $g_1, \dots, g_m$ :  
$$h(x_1, \dots, x_k) = f(g_1(x_1, \dots, x_k), \dots, g_m(x_1, \dots, x_k))$$
- Primitive recursion

The constituting scheme of primitive recursive functions is given by combining two functions  $f$  ( $k$ -ary) and  $g$  ( $k + 2$ -ary) by primitive recursion as follows:

$$\begin{aligned}h(0, x_1, \dots, x_k) &= f(x_1, \dots, x_k) \\h(y + 1, x_1, \dots, x_k) &= g(y, h(y, x_1, \dots, x_k), x_1, \dots, x_k)\end{aligned}$$

The addition function  $pl$  is primitive recursive. It can be build using  $s$  and  $p$ :

$$\begin{aligned}pl(0, x) &= p_1^1(x) \\ pl(y + 1, x) &= s(p_3^2(y, pl(y, x), x))\end{aligned}$$

To make it more clear that this follows the pattern of primitive recursion we can define  $g(x, y, z) = s(p_3^2(x, y, z))$  and write

$$\begin{aligned}h(0, x_1) &= p_1^1(x_1) \\ h(y + 1, x_1) &= g(y, h(y, x_1), x_1)\end{aligned}$$

## Primitive Recursion—Examples II

Subtraction must be defined such that  $x - y$  is 0, if  $y > x$ . We first define the predecessor:

$$\begin{aligned}pr(0) &= z_0 \\pr(y + 1) &= p_2^1(y, pr(y))\end{aligned}$$

And use this to define subtraction *sub*:

$$\begin{aligned}sub(0, y) &= p_1^1(y) \\sub(y + 1, x) &= pr(p_3^2(y, sub(y, x), x))\end{aligned}$$

Note that  $sub(x, y) = y - x$ , not the other way around!



Multiplication, power, factorial, etc. are defined analogously to addition (homework)

Division, etc. can be defined analogously to subtraction

Note that all numbers in all functions are natural numbers in the mathematical sense (i. e. no range limit)

## Encoding Tuples and Lists

Variables in primitive recursive functions have true integer values, hence it is possible to encode two, three, ... values as one:

$$\begin{aligned} \text{enc}(x, y) &= (x + y)(x + y + 1)/2 + y \\ \text{enc}(x, y, z) &= \text{enc}(\text{enc}(x, y), z) \\ &\dots = \dots \end{aligned}$$

The extraction function *dec* for elements from the encoded tuple is primitive recursive, too

By encoding the number of elements as last number it is possible to encode arbitrary lists

Primitive recursive functions are equivalent to LOOP programs:

- Every primitive recursive function is LOOP-computable
- Every LOOP program can be computed by a primitive recursive function

We will show the  $\Rightarrow$ -direction by showing that:

- The basic functions are LOOP-computable
- The composition is LOOP-computable
- The primitive recursion is LOOP-computable

# Translating Basic Functions

The basic functions are LOOP-computable:

$$z_k(x_1, \dots, x_k) = 0 \quad \Rightarrow \quad x_0 := z$$

$$c_k^i(x_1, \dots, x_k) = i \quad \Rightarrow \quad x_0 := z + i$$

$$p_k^i(x_1, \dots, x_k) = x_i \quad \Rightarrow \quad x_0 := x_i$$

$$s(x_i) = x_i + 1 \quad \Rightarrow \quad x_0 := x_i + 1$$

We assume that  $z$  is a fresh variable, that is always zero

For simplicity, we further assume that function programs never modify their arguments  $x_1, \dots, x_k$ , except for  $x_0$  that contains the “return value”

Hence, for computing functions we sometimes need (wlog) fresh variables

# Translating Composition of Functions

Function composition is LOOP-computable:

Let  $h(x_1, \dots, x_k) = f(g_1(x_1, \dots, x_k), \dots, g_m(x_1, \dots, x_k))$

and let  $P_f, P_{g_1}, \dots, P_{g_m}$  be Loop programs computing  $f, g_1, \dots, g_m$  such that the variables  $x_1, \dots, x_k$  are not modified by the programs

Then the following program  $P_h$  computes  $h$ :

$z_1 := x_1 ; z_2 := x_2 ; \dots ; z_k := x_k ;$

$P_{g_1} ; y_1 := x_0 ; x_0 := 0 ; \dots ; P_{g_m} ; y_m := x_0 ; x_0 := 0 ;$

$x_1 := y_1 ; x_2 := y_2 ; \dots ; x_m := y_m ; P_f ;$

$x_1 := z_1 ; x_2 := z_2 ; \dots ; x_k := z_k$

Primitive recursion is LOOP-computable:

We define  $h$  by primitive recursion using  $f$  and  $g$

$$h(0, x_1, \dots, x_k) = f(x_1, \dots, x_k)$$

$$h(y + 1, x_1, \dots, x_k) = g(y, h(y, x_1, \dots, x_k), x_1, \dots, x_k)$$

and assume that  $f$  and  $g$  are computed by  $P_f$  and  $P_g$

Then  $P_h$  is can be constructed as follows:

```
z1 := x1 ; z2 := x2 ; ... ; zk := xk ; zk+1 := xk+1 ;  
x1 := z2 ; x2 := z3 ; ... ; xk := zk+1 ;  
Pf ; v := x0 ; y := 1 ; r := z1 ;  
LOOP r DO  
  x1 := y ; x2 := v ; x3 := z2 ; ... ; xk+2 := zk+1 ;  
  Pg ;  
  v := x0 ; x0 := 0 ; y := y + 1 END ;  
x0 := v ; x1 := z1 ; x2 := z2 ; ... ; xk+1 := zk+1
```

## LOOP and Primitive Recursion—Converse

We will now show the  $\Leftarrow$ -direction like the  $\Rightarrow$ -direction by structural induction. These steps are necessary:

- Addition, Subtraction and Assignment of simple variables are primitive recursive
- Composition maintains primitive recursivity
- LOOP maintains primitive recursivity

Note that while LOOP programs can have arbitrary many variables, primitive recursive programs must encode everything as one

We will not go into the details of this proof in this lecture



Hilbert conjectured that every computable function is primitive recursive

It is easy<sup>TM</sup> to see that this is not the case:

- List all primitive recursive functions “alphabetically”. We write  $f_n$  for the  $n$ -th function on this list
- Define  $d(n) = f_n(n) + 1$ . Using Cantors diagonal argument, clearly the function  $d$  cannot be on the list
- Since  $d$  is computable, the primitive recursive functions are unable to compute all functions

We need a more powerful mechanism!

## From Primitive to $\mu$ -Recursive

We augment the power of primitive recursive functions by adding the so-called  $\mu$ -operator. Let  $f$  be any  $(k + 1)$ -ary function

Then  $\mu f$  is defined by:  $(\mu f)(x_1, \dots, x_k) =$

$$\min\{n \in \mathbb{N} \mid f(n, x_1, \dots, x_k) = 0 \wedge f(m, x_1, \dots, x_k) \text{ is defined } \forall m < n\}$$

If no such  $n$  exists, then  $(\mu f)(x_1, \dots, x_k)$  is undefined

In other words, for any set  $x_1, \dots, x_k$ ,  $f$  is executed as long as no  $f(n, x_1, \dots, x_k)$  is zero

What is  $(\mu f)$  for  $f(a, b) = b - (a \cdot a)$ ?

Without proof, we mention that  $\mu$ -recursive functions have the same computing power as WHILE programs

This is trivial for all operations/constructs we have already considered when comparing LOOP programs and primitive recursive functions

The idea is to encode the WHILE loop via suitably applied  $\mu$ -operators and vice versa

## Summary

---

Syntax and semantics of LOOP programs and WHILE programs

Primitive recursive function and their equivalence to LOOP programs

$\mu$ -recursion and  $\mu$ -recursive functions

## Proof by Induction

---

Induction is a proof technique often used in mathematics and computer science. Induction comes in different flavours:

- Full induction (most well-known)
- Complete induction or strong induction
- Structural induction (extremely useful in computer science)
- Forward-backward induction
- Prefix induction, in inductive step show  $\forall k \in \mathbb{N} . P(k) \rightarrow P(2k) \wedge P(2k + 1)$
- Transfinite induction

Full induction proves a property holds for all (or almost all) natural numbers

There are always two steps required:

- Base case, show  $P(0)$
- Inductive step, show  $P(n) \rightarrow P(n + 1)$

Make sure that the inductive step works from base case onwards! If necessary prove more than one base case!

Examples:  $n^2 + n$  is even,  $n^3 - n$  is divisible by 6,  $n \cdot \sqrt{n} > n + \sqrt{n}$

## Interlude: Fibonacci sequence

The Fibonacci sequence is defined by

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2} \text{ for } n > 1$$

Its first terms are: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377

Similarly, the Tribonacci sequence is defined by

$$F_0 = F_1 = 0, \quad F_2 = 1, \quad F_n = F_{n-1} + F_{n-2} + F_{n-3} \text{ for } n > 2$$

Its first terms are: 0, 0, 1, 1, 2, 4, 7, 13, 24, 44, 81, 149, 274, 504, 927

Check out the “On-Line Encyclopedia of Integer Sequences”!

Fibonacci: <https://oeis.org/A000045>

Tribonacci: <https://oeis.org/A000073>



Complete induction proves a property hold for all natural numbers

We only need one step:

- Inductive step, show  $(\forall a < n . P(a)) \rightarrow P(n)$

We are however allowed to use extra base cases, if that is easier

Examples:  $F_n < 2^n$ ,  $T_n < 2^n$

Structural induction is a proof scheme often used to prove properties of expressions in languages such as logic

This requires, that the language is specified as basic components and combining rules

One proves a base case for all basic components and an inductive step for all combining rules

It is easy to mess up induction proofs. Here are some pointers:

- Do not leave a gap between base case and inductive step!
- Start with the right base case or use multiple ones!
- Use a good standardized proof layout!
- Sometimes proofs are easier with bolder chosen induction hypotheses

## Summary

---

Full induction

Complete induction

Structural induction

Base cases, inductive steps and pitfalls

# Computational Power

---

# Power of Recursive Functions

The primitive recursive functions are all those functions that are computable by for-loops (like those in Ada)

- The number of iterations through the loop is known before entering the loop

The  $\mu$ -recursive functions are all those that are computable by for- or while-loops

- The number of iterations might be unknown in advance
- $\mu$ -recursive functions are intuitively more powerful than primitive recursive ones

Can we find a concrete example of a non-primitive recursive but  $\mu$ -recursive function?

After Hilbert conjectured every computable function to be primitive recursive his student Wilhelm Ackermann found a computable but not primitive recursive function, later named after him:

$$ack(x, y) = \begin{cases} y + 1 & \text{if } x = 0 \\ ack(x - 1, 1) & \text{if } x > 0 \wedge y = 0 \\ ack(x - 1, ack(x, y - 1)) & \text{if } x > 0 \wedge y > 0 \end{cases}$$

The Ackermann function grows very fast:

$$ack(0, y) = y + 1$$

$$ack(1, y) = y + 2$$

$$ack(2, y) = 2y + 1$$

$$ack(3, y) = 2^{y+3} - 3$$

$$ack(4, y) = 2 \uparrow (y + 3) - 3$$

(here  $\uparrow$  is tetration i. e.  $2 \uparrow 4 = 2^{2^{2^2}}$ )



# Computing the Ackermann Function using $\mu$ -Recursion

It turns out that the Ackermann function can be computed using  $\mu$ -recursive functions

$\mu$ -recursion is used to count down  $y$ , rest follows in place

Actual “implementation” is not very trivial

# Ackermann Function is not Primitive Recursive I

It can be shown that the Ackermann function is not primitive recursive by providing the following theorem about the growth of primitive recursive functions:

**Majorization Theorem:** For every  $k$ -ary primitive recursive function  $f$  there exists an  $n \in \mathbb{N}$  such that for all  $x_1, x_2, \dots, x_k$

$$f(x_1, x_2, \dots, x_k) < \text{ack}(n, \max(x_1, x_2, \dots, x_k))$$

Idea: The Ackermann function grows faster than any primitive recursive function

## Ackermann Function is not Primitive Recursive II

If the Ackermann function was primitive recursive, and  $m$  would be the function that was fastest growing among all binary primitive recursive functions, this would lead to this contradiction:

$$ack(n, y) \leq m(n, y) < ack(n, y) \quad \text{for all } y > n$$

The Ackermann function is however  $\mu$ -recursive

Proof of the majorization theorem left as homework ;-)

Kruskal's algorithm for finding a minimum spanning tree for a weighted undirected connected graph proceeds as follows:

- Every node  $u$  becomes a singleton set  $S_u$
- Sort all edges by non-decreasing weight
- For all edges  $e = (u, v)$  in sorted order:
  - If minimum node in  $S_u \neq$  minimum node in  $S_v$ :  
put  $e$  into spanning tree
  - Unify (compute union of)  $S_u$  and  $S_v$

## Application of the Ackermann Function II

If minimum find and set union are computed with weighted union and path compression, (two techniques that help keep path lengths short) the algorithm has a runtime less than  $O(n \cdot k)$  if the size  $n$  of the graph is less than  $ack(k, k)$

The proof is rather complex, and thus left as second homework exercise :-)

Moreover, the fascinating analysis of this algorithm is not a topic in this course :-)

## Summary

---

LOOP programs have the same computational power as primitive recursive functions

WHILE programs have the same computational power as  $\mu$ -recursive functions

The Ackermann function is WHILE-computable, but it is not primitive recursive

The inverse of the Ackermann function has applications in the Union-Find algorithm used for instance in Kruskal's minimum spanning tree algorithm

## Proof of Majorization

---

We will set out to prove the majorization theorem:

**Majorization Theorem:** For every  $k$ -ary primitive recursive function  $f$  there exists an  $n \in \mathbb{N}$  such that for all  $x_1, x_2, \dots, x_k$

$$f(x_1, x_2, \dots, x_k) < ack(n, \max(x_1, x_2, \dots, x_k))$$

It will be an arduous, but worthwhile journey

We will employ rigour and induction!



We will proof some properties of the Ackermann function mostly using induction:

- Lemma 1: The Ackermann function is well-defined
- Lemma 2: The Ackermann function's result is always larger than its second argument
- Lemma 3a: The Ackermann function is strictly monotonic w. r. t. to its *second* argument
- Lemma 3b: The Ackermann function is strictly monotonic w. r. t. to its *first* argument
- Lemma 4: The Ackermann function increases stronger on the first than on the second argument

# Collatz Conjecture

The property that a function is well-defined, is essential. We will look at the well-known Collatz conjecture:

$$\text{coll}(x) = \begin{cases} 1 & \text{if } x = 1 \\ \text{coll}(x/2) + 1 & \text{if } x \text{ is even} \\ \text{coll}(3 \cdot x + 1) + 1 & \text{otherwise} \end{cases}$$

Today, it is still unknown whether  $\text{coll}(x)$  is defined for every  $x$ . Will the recursion always terminate or is there a loop?

A similar problem is based on abundant/perfect/deficient numbers: when summing up the divisors of a number, the sum can be larger, as large as, or smaller than the number itself.

# Proof of Lemma 1

We use two induction arguments: Inner induction on  $y$  and outer induction on  $x$

To show:  $ack(x, y)$  is well-defined for all  $y$ , for all  $x$ . Proof by induction on  $x$ :

- Base case 0:  $ack(0, y) = y + 1$  is defined
- Inductive step  $x + 1$  with induction hypothesis  $ack(x, y)$  is well-defined for all  $y$ :

Proof by induction on  $y$ :

- Base case 0:  $ack(x + 1, 0) = ack(x, 1)$  is defined by (outer) induction hypothesis
- Inductive step  $y + 1$  with induction hypothesis  $ack(x + 1, y)$  is well-defined:  
 $ack(x + 1, y + 1) = ack(x, ack(x + 1, y))$  is well-defined using first inner then outer induction hypothesis

The same principle will also be applied in other proofs

## Proof of Lemma 2

We use again two nested induction arguments:

To show:  $ack(x, y) > y$  for all  $y$ , for all  $x$ . Proof by induction on  $x$ :

- Base case 0:  $ack(0, y) = y + 1 > y$
- Inductive step  $x + 1$  with induction hypothesis  $ack(x, y) > y$  for all  $y$ :

Proof by induction on  $y$ :

- Base case 0:  $ack(x + 1, 0) = ack(x, 1) > 1 > 0 = y$  by outer induction hypothesis
- Inductive step  $y + 1$  with induction hypothesis  $ack(x + 1, y) > y$ :  
 $ack(x + 1, y + 1) = ack(x, ack(x + 1, y)) \geq ack(x, y + 1) > y + 1$  applying first inner then outer induction hypothesis

To prove: The Ackermann function is strictly monotonic w. r. t. to its second argument i. e.  $ack(x, y + 1) > ack(x, y)$  for all  $y$ , for all  $x$

We prove this directly:

- Either  $x = 0$ ,  
then  $ack(x, y + 1) = ack(0, y + 1) = y + 2 > y + 1 = ack(0, y) = ack(x, y)$
- Or  $x > 0$ ,  
then  $ack(x, y + 1) = ack(x - 1, ack(x, y)) > ack(x, y)$  by Lemma 2

Lemma 4: The Ackermann function increases stronger on the first than on the second argument

To show:  $ack(x + 1, y) \geq ack(x, y + 1)$ . Proof by induction on  $y$ :

- Base case 0:  
 $ack(x + 1, y) = ack(x + 1, 0) = ack(x, 1) = ack(x, 0 + 1) = ack(x, y + 1)$
- Inductive step  $y + 1$  with induction hypothesis  $ack(x + 1, y) \geq ack(x, y + 1)$ :  
 $ack(x + 1, y + 1) = ack(x, ack(x + 1, y)) > ack(x + 1, y) \geq ack(x, y + 1)$  using first Lemma 2 then the induction hypothesis

Lemma 3b: The Ackermann function is strictly monotonic w. r. t. to its first argument

To show:  $ack(x + 1, y) > ack(x, y)$  for all  $x$ , for all  $y$

Again, we prove this directly:

- Either  $y = 0$ ,  
then  $ack(x + 1, y) = ack(x + 1, 0) = ack(x, 1) > ack(x, 0) = ack(x, y)$  by Lemma 3a
- Or  $y > 0$ ,  
then  $ack(x + 1, y) = ack(x, ack(x + 1, y - 1)) > ack(x + 1, y - 1) \geq a(x, y)$  using first Lemma 2 then Lemma 4

**Majorization Theorem:** For every  $k$ -ary primitive recursive function  $f$  there exists an  $n \in \mathbb{N}$  such that for all  $x_1, x_2, \dots, x_k$

$$f(x_1, x_2, \dots, x_k) < \text{ack}(n, \max(x_1, x_2, \dots, x_k))$$

We primarily use structural induction on the definition of a primitive recursive function



For the zero function:  $z_k(x_1, x_2, \dots, x_k) = 0$  we have:

$$z_k(x_1, x_2, \dots, x_k) = 0 < \max(x_1, x_2, \dots, x_k) + 1 = \text{ack}(0, \max(x_1, x_2, \dots, x_k))$$

So  $n = 0$  works

For the constant function:  $c_k^i(x_1, \dots, x_k) = i$  we have:

$$c_k^i(x_1, \dots, x_k) = i < i + 1 = \text{ack}(0, i) < \text{ack}(i, 0) \leq \text{ack}(i, \max(x_1, x_2, \dots, x_k))$$

So  $n = i$  works

For the projection function:  $p_k^j(x_1, \dots, x_k) = x_j$  we have:

$$p_k^j(x_1, \dots, x_k) = x_j < \max(x_1, x_2, \dots, x_k) + 1 = \text{ack}(0, \max(x_1, x_2, \dots, x_k))$$

So  $n = 0$  works

For the successor function:  $s(x) = x + 1$  we have:

$$s(x) = x + 1 < x + 2 = \mathit{ack}(1, x)$$

So  $n = 1$  works

## Inductive Step: Function Composition I

Let  $f$  be an  $m$ -ary primitive recursive function and let  $g_1, \dots, g_m$  be  $m$   $k$ -ary primitive recursive functions, then functions composition gives this  $k$ -ary Function  $h$ :

$$h(x_1, \dots, x_k) = f(g_1(x_1, \dots, x_k), \dots, g_m(x_1, \dots, x_k))$$

Our induction hypothesis says that  $C, D_1, \dots, D_m$  exist such that

$$f(x_1, \dots, x_m) < \text{ack}(C, \max(x_1, \dots, x_m)) \text{ for all } x_1, \dots, x_m$$

and

$$g_i(x_1, \dots, x_k) < \text{ack}(D_i, \max(x_1, \dots, x_k)) \text{ for all } 1 \leq i \leq m \text{ and } x_1, \dots, x_k$$

Let  $B = \max(C, D_1, \dots, D_m)$  then we have: (next slide)

## Inductive Step: Function Composition II

$$\begin{aligned}\text{LHS} &= \text{ack}(B + 2, \max(x_1, \dots, x_m)) \\ &\geq \text{ack}(B + 1, \max(x_1, \dots, x_m) + 1) && \text{(Lemma 4)} \\ &= \text{ack}(B, \text{ack}(B + 1, \max(x_1, \dots, x_m))) && \text{(Ackermann.3)} \\ &> \text{ack}(B, \text{Max}_{i \leq k} \{\text{ack}(D_i, \max(x_1, \dots, x_m))\}) && (B + 1 > D_i + \text{Lemma 3a}) \\ &> \text{ack}(B, \text{Max}_{i \leq k} \{g_i(x_1, \dots, x_k)\}) && \text{(induction hypothesis + Lemma 3a)} \\ &\geq \text{ack}(C, \text{Max}_{i \leq k} \{g_i(x_1, \dots, x_k)\}) && (B \geq C + \text{Lemma 3b}) \\ &> \text{ack}(C, f(x_1, \dots, x_m)) && \text{(induction hypothesis)} \\ &\geq f(g_1(x_1, \dots, x_k), \dots, g_m(x_1, \dots, x_k)) && \text{(Specialization)}\end{aligned}$$

## Inductive Step: Primitive Recursion

Here we use a simplified version of primitive recursion to keep our proof short

Let  $f$  be a unary function. Then simplified primitive recursion gives us a unary function  $h$  defined by:

$$h(0) = 0$$

$$h(x + 1) = f(h(x))$$

## Inductive Step: Primitive Recursion

Our structural induction hypothesis is that we have some  $E$  such that:

$$f(x) < ack(E, x) \text{ for all } x$$

To show:  $h(x) < ack(E + 1, x)$  for all  $x$ . Proof by induction on  $x$ :

- Base case 0:  $h(x) = h(0) = 0 < 1 = ack(0, 0) < ack(E + 1, 0) = ack(E + 1, x)$
- Inductive step  $x + 1$  with induction hypothesis  $h(x) < ack(E + 1, x)$ :  
 $h(x + 1) = f(h(x)) < ack(E, h(x)) < ack(E, ack(E + 1, x)) = ack(E + 1, x + 1)$   
using first the structural and then the full induction hypothesis.



# We are done!

And we only cheated a little

We have shown the majorization theorem and with it that the Ackermann function is not primitive recursive.

## Summary

---

The Ackermann function grows faster than any function that can be formulated by primitive recursion

First we proofed some lemmas

Then we proofed the majorization theorem

We employed full induction, in some cases nested and structural induction.

## Decidability

---

Alan Turing invented a primitive type of machine, the Turing machine, consisting of:

- An unlimited (infinite) storage tape, divided into fields
- A read/write head, that can modify the tape and change its position by one field
- A finite state machine that controls how the head acts on the tape using the current state and tape

Church' thesis: Everything that is computable can be computed by a Turing machine

Like in a virtual machine, Turing machines are able to simulate other Turing machines

For this simulation, unlimited storage is a must, since a machine with limited storage capacity cannot simulate a machine with the same storage capacity

We call a Turing machine that simulates other Turing machine given as a “parameter” a universal Turing machine

Nothing keeps an Ada program from using unlimited storage to simulate the working of a Turing machine

There are no inherent limits for Ada programs, so everything that is computable can also be computed by an Ada program

This can be used to create a universal Ada machine, that is able to simulate any program written in Ada

Storage of this universal Ada machine could be organized as a linked list, so there is no need to keep counters (which are inherently limited to some bit size)

The halting problem, from German “Halteproblem” is in some sense the “easiest question” that can be asked about an Ada program:

*Given some input will it ever terminate?*

The same question can be asked about Turing machines or any other computational model

Turing showed that some problems are not computable (decidable) by programs (i. e. Turing machines)

## Interlude: Gödel's Incompleteness Theorems

In 1931 Gödel proved that any sufficiently powerful logical system contains statements that can neither be proven nor refuted

Logical systems may contain quantifiers like  $\forall$  or  $\exists$  which are hard to handle with programs

There are some parallels between Gödel's Incompleteness Theorems and Turing's Undecidability results



# Definition of Decidability

A problem is a yes/no question asked about an optional input

A problem is *decidable* iff there is an Ada program that terminates with the correct answer on any input

A problem is *semi-decidable* iff there is an Ada program that

- if the answer is yes terminates with yes and
- if the answer is no either terminates with no or fails to terminate

A problem is *undecidable* iff it is not decidable

We will later show, that a problem is decidable iff it is semi-decidable and its complement is semi-decidable as well

It turns out that for some problems (Busy Beaver) neither the problem nor its complement are semi-decidable

The halting problem is semi-decidable and undecidable

# Special Halting Problem

The special halting problem is in some sense easier than the general halting problem.

It answers the question, whether a program terminates if it gets its own code as input

Our proof of the undecidability of the general halting problem consists of two steps:

1. Show that the special halting problem is undecidable
2. Reduce the special halting problem to the general halting problem

# Undecidability of the Special Halting Problem

We assume that the program to be analysed is provided as source code

Furthermore, we assume that there is some magic Ada code:

```
function SHalt(Code_Data : String) return Boolean
```

**SHalt** shall return **true** if the program in **Code\_Data** will terminate when it is provided itself as input, else **false**

We will demonstrate, that the existence of a program **SHalt** inevitably leads to a contradiction. Therefore, it does not exist and the special halting problem is undecidable

## Modifying SHalt

Now we modify the program that defines **SHalt** to get a new Program **MSHalt**:

```
procedure MSHalt is begin
  if SHalt (<MyOwnCode>) then
    loop
      null;
    end loop;
  end if;
end MSHalt;
```

We know that an Ada program can handle its own source code

We find a contradiction when **SHalt** is used to examine **MSHalt**!

## A Contradiction

Since **SHalt** is the same function in both cases, the call of **SHalt** from within the **SHalt** main program and from within **MSHalt** will give the same result for the same input

If **SHalt** returns true in **MSHalt**, then **MSHalt** will not terminate, but **SHalt** will say that it will—that cannot be

If **SHalt** returns false in **MSHalt** then **MSHalt** will terminate, but **SHalt** will say that it will not—that cannot be either

*Contradiction!*

Reduction is a powerful tool in various areas of computer science e.g. when showing NP-completeness of problems in complexity theory

The idea is always the same:

- Take an instance of some problem  $A$
- An “easy to compute” function transforms it to an instance of a problem  $B$
- Solve the resulting instance of problem  $B$
- Transform the solution of the instance of  $B$  to a solution of the instance of  $A$

In this case we have reduced  $A$  to  $B$ . We say that  $A$  is reducible to  $B$  and write

$$A \sqsubseteq_{\text{TM}} B$$

If  $A \sqsubseteq_{\text{TM}} B$ , i. e. the problem  $A$  is reducible to the problem  $B$ , it intuitively means that that  $A$  is either as “difficult” or less “difficult” than  $B$

We now know that the special halting problem (We write  $SHP$ ) is undecidable

We want to show that the general halting problem (We write  $HP$ ) is also undecidable

We will do so by reducing the special halting problem to the general halting problem. We will show:

$$SHP \sqsubseteq_{\text{TM}} HP$$



Reducing the special to the general halting problem is easy:

- Take an instance of the special halting problem
- Duplicate the instance and feed it to the program that decides the general halting problem as code *and* data
- The answer of the general halting problem program is the answer of the special halting problem program

Using reductions, we can prove more problems to be undecidable

To show a problem  $A$  is undecidable, we have to reduce a problem that is already known to be undecidable to  $A$ . This can be  $HP$  or  $SHP$ :

$$HP \sqsubseteq_{\text{TM}} A \quad \text{or} \quad SHP \sqsubseteq_{\text{TM}} A$$

Suitable reductions can be found for most questions about algorithms/programs, e.g. whether an algorithm expects the empty input, no input at all, fails with some error, uses some resource etc.

Colloquially, all interesting properties of programs are undecidable!

## Reduction Example

We define the Martin problem (*MP*). It asks whether a given Ada program's output is "Martin" for a given input. We can reduce *HP* to *MP*

First we construct this program:

```
procedure Martin is begin
  -- Inject original code here (remove any output)
  Ada.Text_IO.Put_Line("Martin");
end Martin;
```

We pass this program with the specified input into the *MP* decider. Its answer is the answer to the *HP*, hence:

$$HP \sqsubseteq_{TM} MP, \text{ so } MP \text{ is undecidable}$$

Because almost all questions for programs are undecidable, one typically uses testing to verify programs

However, one has to keep in mind that even 1 billion successful tests do not guarantee that the program works correctly. But one test that fails shows that the program is incorrect

Nevertheless, carefully chosen tests can dramatically increase the confidence that some piece of software works as expected

Some see Turing undecidability results as a job guarantee for software testers ;-)

## Summary

---

The halting problem is semi-decidable but not decidable

We prove problems to be undecidable using reductions

In fact all interesting properties of programs are undecidable

Software testers need not fear unemployment

# Software Test

---

Software Testing is not Debugging:

- Testing tries to show that software is free of certain defects
- Debugging tries to find the root causes of failures, i. e. the defect in the code that caused the infection which finally led to the failure

Testing does not prove software correctness

Testing can only show that the given test case in the given environment is properly handled

Choosing good test data increases trustworthiness of software

Reintroduction of defects can be avoided (regression testing)

Usually test is a part of the software development process. It is done by:

- Developers, which can lead to intra role conflicts
- Specialized testers, which is usually expensive
- Automated testing equipment, which is the preferred way

Some say: “Testing is either automated or avoided!”

Regression tests are especially useful to avoid the resurrection of defects

Projects that use DevOps integrate parts of the tests into the check in process, so that the repository contains only tested code



# Black Box vs. White Box Tests

Tests can be written in three different ways:

- In **black box tests** the tester has neither knowledge of nor access to the internals of the program (source code etc.)
- In **white box tests** the tester has good knowledge of the internals of the program
- In the middle, in **grey box tests** know something about the programs internals. They may be able to observe how the program handles specific situations

Typically, UX/UI tests are performed by external users, so they are mostly black box

# How Much Testing is Enough

Typically, exhaustive testing is impossible

A simple Ada function like

```
function Large (A, B: Integer) return Integer;
```

Requires 18 446 744 073 709 551 616 test cases to be tested exhaustively

Running them at a rate of 4 000 000 000 per second needs some 146 years

Usually functions have more parameters, furthermore in OOP testing must consider the state hidden inside the object

# Non-Exhaustive Testing

Since exhaustive testing is typically impossible, careful selection of test data is needed

Test cases are

- Data that is considered to be a valid input together with the expected output
- Data that is considered to be an invalid input together with some error messages

However, using random data is also possible:

- See how the system reacts in case of (mostly) wrong data
- Control “wrongness” by using specific random sources

The idea: What happens if you feed software more or less randomly created data?

Interestingly enough, fuzz testing was created by somebody who had a very noisy telephone line and thus often had nonsense data input

Fuzz testing is useful for embedded systems where binary data is transmitted over the wire

Fuzz testing can also be a good test for security

Approaches to fuzz testing are:

- Test the operating system resilience
- Test application input for text and command line
- Test application input for windows messages

The tool “CrashMe” tested operating system resilience by:

- Sending invalid pointers or numbers to system calls (i. e. negative file handles, `0xFFFFFFFF` as pointers, ...)
- Sending random data in user allocated and hence valid buffers to system calls

Tests showed in 1996 that Unix kernels (except IRIX) were mostly resilient against attacks

In 2016 a team still found 10 different ways to create a BSOD in Windows Server!

Application resilience can be tested in two ways:

- For traditional text oriented tools random data can be used in the command line parameters or in input files or streams (stdin)
- For GUI applications random data can additionally be inserted into the system event loop

Suitably constructed random data, can be used to test for buffer overflows

In 2000 empirical tests showed that more than 95 % of all Windows applications could be crashed by sending them fuzz data

In contrast to fuzz testing, systematic testing relies on software testers to create test cases

They typically try to test software with as few as possible test cases

Hence, one of the most important tasks while testing systematically is test data selection

## Testing for Teachers in Tertiary Education ;-)

In teaching, I often provide test cases for programming assignments. There is an inevitable outcome to this, which is exemplified by this example:

Say, students are tasked to write a function `gcd` that computes the greatest common divisor of two numbers

Further say, I provide this test case:

```
assertEquals (Gcd (42,35), 7);
```

Inescapably, I will get some implementations that look like this:

```
function Gcd (A : Natural, B : Natural) return Positive is  
begin  
    return 7;  
end Gcd;
```



## Test Case Selection: Production Data

One of the easiest ways to source test data is to use existing production data.  
However:

- Using real data might be problematic due to GDPR
- Real data will usually not contain invalid inputs
- There may well be security concerns using real data on development/integration systems

So real data might be a first try, but it is typically not sufficient

Test data needs to contain at least two parts:

- Valid data to show how the system normally works
- Invalid data to show that the system handles errors gracefully

## Anecdote: Beer Tax

Until recently the German customs service used a dated Cobol implementation to calculate how much beer tax breweries have to pay

A software project to replace this with a more modern Java application found what felt like decades of parsable tax data

The data included inputs, intermediate results and end results (tax amounts to be paid)

*These data have proved invaluable:*

Verifying that the new application yielded identical results induced tremendous confidence in it

## Anecdote: Employee Data on Test Systems

A software project wanted to use a limited set of production data for a performance test on an integration setup

To run the test, the integration setup was connected to a production employee data base. During the test it downloaded information on all employees of the company

The integration setup was cloud-based, poorly protected and it was not taken offline after the tests. As a result, substantial amounts of personal employee data was found on BitTorrent

Manually created test data should consist of:

- Typical valid program inputs
- Typical invalid program inputs
- “Empty” data: Either as a boundary case of valid or invalid test data
- Boundary cases: Data points that are just valid or just invalid.
- Realistically sized and huge test data, especially for performance tests
- After careful consideration: Real productive data

## Boundary Cases

- Numbers, that are the minimum/maximum allowed, and numbers that are one smaller/larger
- String that are just long enough or just too long
- Files that contain a new line at the end/no new line at the end/**CrLf**, **Lf**, **Cr**, Even **LfCr** or mixed newlines
- Strings containing special characters, like:
  - German Umlauts: ä, ö, ü, Ä, Ö, Ü, ß
  - French cédille Ç, accent aigu é/grav à, è, ù/tréma ë, ï, ü/circonflexe â, ê, î, ô, û
  - Hebrew, including Right-to-left characters
  - 🙈 😊 😂 😈 🤩 🤖 🦷 🦋 🍄 🍷 ☀️ ☢️ ☣️ including combos: 🙈🤖 🙈☢️
- Huge and small numbers, files, strings, arrays, ...
- Float: NaN,  $\infty$ ,  $-\infty$ ,  $-0$ , 0 and denormalized numbers

To assess whether test cases are well-chosen, code coverage metrics can be used:

- Statement coverage: Fraction of statements executed during the tests
- Branch coverage: Fraction of branches taken during the test
- Condition coverage: Complicated, branching conditions need to evaluate to true/false in specific ways
- Loop coverage: Every loop was executed zero, one and “many” times
- Path coverage: Every combination of statements was executed

## Statement & Branch Coverage

Statement Coverage can usually be checked using a profiler or instrumentation and marking every statement was executed at least once

If statement coverage is less than 100 %, usually more test cases need to be written

Branch coverage also considers whether:

- In every if-statement without an else-branch the condition evaluated to false at least once
- Every switch-statement without a default case the value matched none of the cases
- For every loop with pre-test the loop body was not executed at all

Simplified definition: When a condition can be (dis-)satisfied in multiple ways, all of them should be tested

The branching instruction

```
if S != Null and then S.Length > 0 then ...
```

should be tested with a null string, an empty string, and a non-empty string to achieve full condition coverage



Loop coverage checks that loops have been executed zero times, one time and multiple times

When this leads to testers feeling motivated to test loops with a high repetition count, memory leaks or inefficient code might be found

A somewhat typical error in C is seen here:

```
for(int i = 0; i < strlen(s); i++) {...}
```

This code might execute `strlen` at every iteration of the loop and thus yield a quadratic runtime

Path coverage is the most comprehensive form of testing, for complete path coverage all possible paths through the code need to be checked

This typically creates a combinatorial explosion of different test cases, so path coverage is the most comprehensive but also most complex coverage

In realistic tests, path coverage can only be reached up to a certain (low) percentage

It is typically used as a theoretical model to compare code coverage methods

Internal state coverage: Since OOP greatly depends on data stored inside the object, one should also consider the different behaviour of code depending on the object state

Dataflow coverage: Information can often flow between variables on very different paths, so a good test should also test along the dataflow

In practice, this is what I have seen used:

- Most projects do not check coverage—This is appalling!
- Statement coverage: Some software projects use this
- Branch Coverage: This should be the norm! Unfortunately it is not
- Condition Coverage: In safety-critical embedded systems high or complete condition coverage often is legally or contractually required.  
Achieving this can be exhausting and requires carefully written code

# Pitfalls of Code Coverage

Different tools measure coverage differently

If the code is bad, a good coverage might be unachievable. Dead code?

Low statement coverage (pretty much) guarantees that tests or code are bad, but high statement coverage does not guarantee that either is good. Developers might:

- Insufficiently check program outcomes
- Actually expect wrong behaviour

Like (almost) all software quality metrics, code coverage can be cheated, if programmers want to—If code coverage criteria are set in the “contract”, employ mutation testing as a sanity check

Another important way to create test data is by evaluation of defects

When tracing back a program failure to the originating defect, there must be some data that triggers the defect and the propagation of the infection to the failure

Using this data as test data at least assures that this specific failure will not be reintroduced later

# Mutation Testing

In Mutation testing one deliberately introduces simple bugs into an implementation. Then one checks which test fail indicating that there is a bug.

Mutation testing can serve multiple goals:

- Measure test quality (use as a metric)
- identify weak tests (those that fail on few or no mutations)
- identify weakly tested parts of the code (in those parts mutation seldomly cause test to fail)
- Potentially, to learn about error propagation

Mutation testing is done either manually or automated

## Summary

---

There are two approaches to testing, both of which are useful:

- Random tests (Fuzzing)
- Systematic tests

Black box, white box and grey box test

Code coverage metrics can help assess the quality of tests