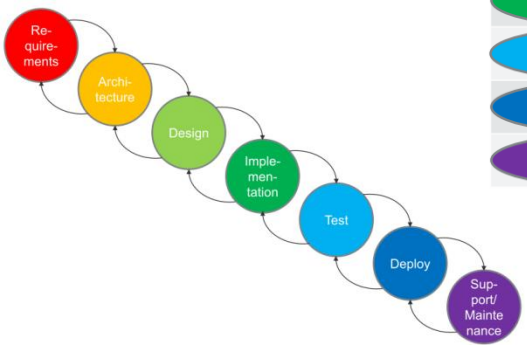
28. Januar 2013

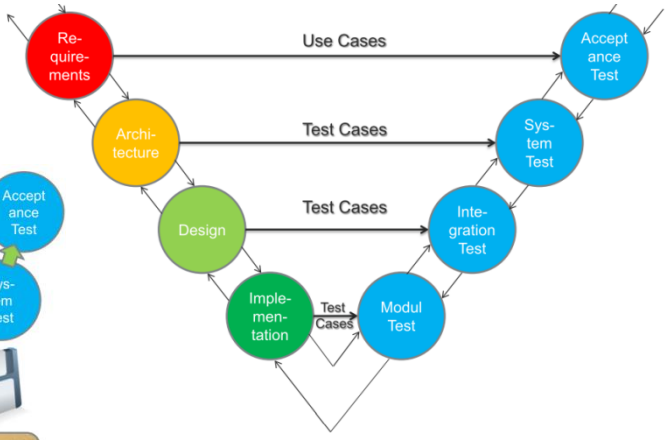
Kapitel 1 - Introduction

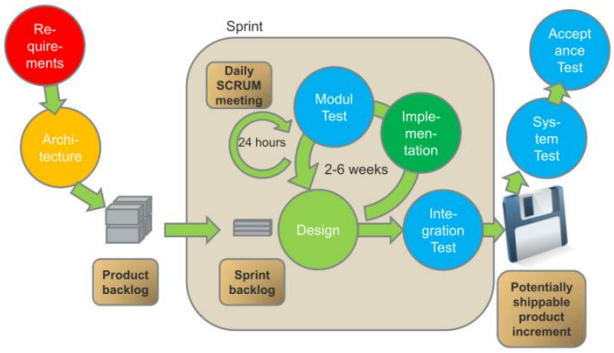
Zuverlässigkeit:

- Funktionsfähigkeit (Reliability)
- Erreichbarkeit (availability)
- Sicherheit (savety, security)

Phase	Participants	Aspects	Methods/ Outcome
Concept	Marketing, PM, Management	Feasibility, cost, risk	Cost-benefit-Analysis, Risk-Management plan
Requirements	PM, clients	Client's needs	Functional requirements document, use-cases
Architecture	Architect, developer, tester	Dependability, maintain-ability, testability, …	Architecture document
Design	Architect, developer, QA	Transform requests into system design	Design documentation and development plan
Implemen-tation	Developer, QA	Implement design in SW	Software + SW documentation
Test	Developer, QA	Design and execute tests based on requests	Various validation and verification methods, reviews
Deploy	Developer, architects, sales	Manufacturing, distribution, installation	
Support/ Maintenance	Developer, sales customer support	Ensure systems dependability in operation	Post implementation and in-process reviews







Arten von Testplänen:

- "mission plan (why)": am wenigsten detailliert
- "strategic plan (what & when)": Generelle Kriterien wie zur Abdeckung(coverage) der Testkriterien
- "tactical plan (how & who)": detailliert, pro Produkt, beinhaltet Test Kriterien, Ergebnisse, ...

Grundbegriffe

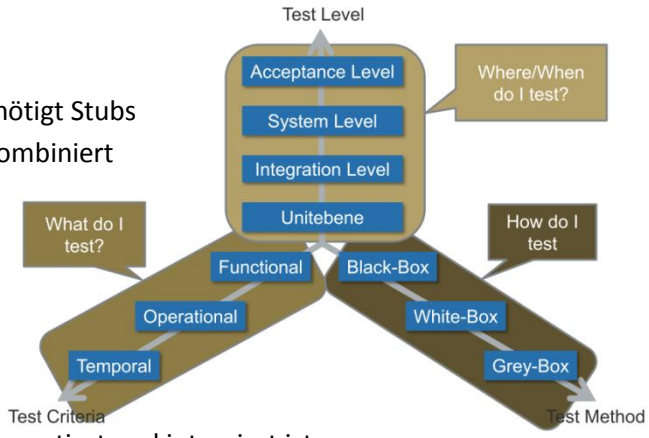
- **Validation**: Der Prozess am Ende der Softwareentwicklung der gewährleistet, dass man sie auch funktioniert. ??
- **Verification**: Bestimmt ob das Produkt in der aktuellen Phase die Bedingungen der letzten vollständig abdeckt. ??
- **Static Testing**: Testen ohne das Programm auszuführen (das beinhaltet formale Analysen)
- **Dynamic Testing**: Testen durch das Ausführen des Programms mit realen Werten.
- **Software Fault**: Ein Designfehler im Code. (z.B. falsche Abbruchbedingung in einer For-Schleife)
- **Software Failure**: Ein Fehler, der sich nach außen hin bemerkbar macht. (z.B. ein falsches ausgegebenes Ergebnis, Programmabsturz, …)
  3 Bedingungen um einen Failure zu beobachten:
  - **Reachability**: Die Stelle muss erreicht werden können (bei Ausführung)
  - **Infection**: Der Zustand muss falsch sein.
  - **Propagation**: Der Zustand muss den Output fehlerhaft beeinflussen.
- **Software Error**: Ein Zustand in der Ausführung, in welchem das Programm etwas macht, was es laut Spezifikation nicht machen sollte. (z.B. das Programm muss alle Listenelemente

prüfen, aber lässt einen aus) In diesem Moment ist das Programm im Error-Zustand. (das Ergebnis muss nicht falsch deswegen sein)

- **Testing**: Eingaben finden mit denen die Software failt.
- **Debugging**: Der Prozess den Fault von einem „Failure“ zu finden.
- **Software Observability**: Wie einfach ist es das Verhalten zu überwachen?
- **Software Controllability**: Wie einfach sind die Eingaben für die Software? (z.B. Tastatur einfach, über Sensoren schwer)

Kapitel 2 – Testing Basics

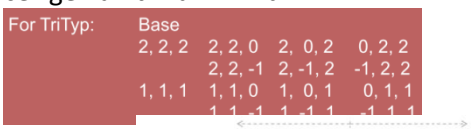
- **JUnit**
  - Eine "Suite" kann eine Menge von Testfällen ausführen.
  - SetUp() und tearDown() Methoden für Testklassen (... assertEquals(), ...)
- **Integration Test**: Kombiniert verschiedene Einheiten zu einem Ganzen. Der Test prüft, dass es immer noch stabil läuft wie davor die einzelnen Komponenten.
- **Big-Bang-Integration**: Alle Komponenten müssen vollständig implementiert sein, damit es funktioniert. Späte Tests und späte Erkennung von Fehlern auch in der Architektur.
- **Structure-Based Integration**:
  - Bottum-up: keine Stubs nötig
  - Top-Down: früh Prototyp vorhanden, benötigt Stubs
  - Outside-In: Vorteile der oberen Beiden kombiniert
  - Inside-Out: nicht genutzt
- **Function-Oriented Integration**:
  - Schedule-driven: first come first serve
  - Risk-driven: schwerstes zuerst
  - Test-driven: Test ist vorhanden
  - Use-case-driven

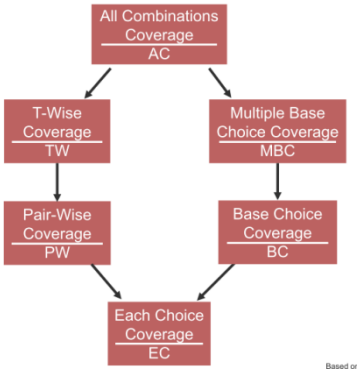


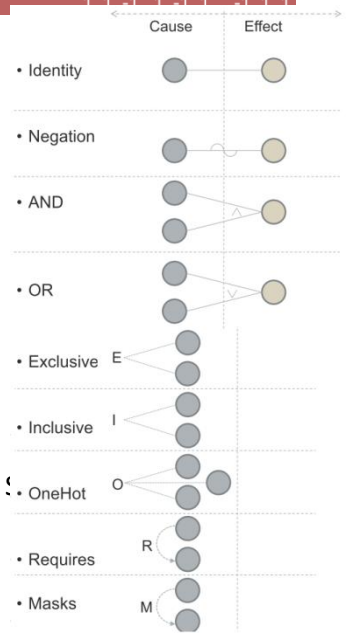
- **System Test**: Startet wenn alles vollständig implementiert und integriert ist.
- **Acceptance Test**: Wird üblicherweise in der Testumgebung des Kunden durchgeführt.
- **Functional Test**: passt es zu den Vorraussetzungen, ist es robust, ist es kompatibel, random input
- **Operational Test**: Installation möglich, Usability, Sicherheit
- **Temporal Test**: Komplexität, Stresstest, größte inputmenge, Geschwindigkeit allgemein
- **Black-Box Test**: Testfälle anhand Spezifikation schreiben
- **White-Box Test**: Testfälle mit Wissen über den internen Code schreiben
- **Gray-Box Test**: Testfälle die der Entwickler schreibt, ohne den Code nochmals genau anzuschauen
- **Diversifying Tests**:
  - Regression Tests: Alte Version wird gegen neue getestet.
  - Back-to-Back Tests: 2 oder mehr Teams entwickeln voneinander unabhängig das selbe
- **Mutation Testing**: Um die Testfälle auf effizient zu prüfen (Absolutwerte ändern, +/-, </>, zugriff ändern (private, public), in geerbter Klasse Variablen hinzufügen, Methode überschreiben)
  Mutation Score: "Dead mutants" / ("number of mutants" - "number of equivalent mutants")
  Bei 100% ist der sind die Testfälle ausreichend.

Kapitel 3 - Functional Testing - Black-Box Test

- Selbst für kleine Programme ist Wertebereich bereits unendlich.
- **Input Space Partioning** (ISP): Teilt den Wertebereich in verschiedene "Regionen" auf und zumindest von jedem Bereich wird ein Wert gewählt. Die Partitionen müssen *paarweise disjunkt* sein und alle zusammen sind exakt der Wertebereich.
- Schritte zur Modellierung des Wertebereichs (**Input Domain Model** (IDM))
    1. *Alle zu testbaren Funktionen identifizieren*
    2. *Alle Parameter finden* (Methoden (Parameter und globale Zustandsvariablen), Komponenten (Parameter der Methoden, Zustandsvariablen), System (Eingaben wie Datenbanken und Dateien)
    3. *Modell des Wertebereich*
    4. *Ein Testkriterium für Kombinationen der Werte wählen*: ein Block für jede Charakteristik, alle Kombinationen sind nicht machbar, das Coverage Kriterium erlaubt es eine Teilmenge zu wählen
    5. *Verfeinern der Kombinationen zu den Testeingaben*
- Annäherungen an das IDM
    1. **Interface-based**: Entwickelt Charakteristiken direkt aus den individuellen Eingaben, einfach zu entwickeln. Betrachte jeden Parameter isoliert.
    (Dreieck (drei Int): jeweils größer, gleich oder kleiner null als Charakteristik)
    2. **Functionality-based**: Entwickelt Charakteristiken aus einer Verhaltensicht des Programms, schwierig zu entwickeln
    (Dreieck (drei Int): (un-)gleichseitig, gleichschenklig, ungültig als Charakterstik)
- Verschiedene Kominationen der Werte
    1. **All Combinations** (ACoC): Alle Kombination der Blöcke aller Charakteristiken müssen verwendet werden. $\prod_{i=1}^{Q} B_i$
    2. **Each Choice** (EC): Ein Wert aus jedem Block für jede Charakteristik muss mindestens einmal im Test vorkommen. $Max_{i=1}^{Q} B_i$
    3. **Pair-Wise** (PW): Ein Wert aus jedem Block für jede Charakteristik muss mit einem Wert von jedem anderen Block jeder Charakteristik kombiniert werden.
    4. **t-Wise** (TW): Ein Wert von jedem Block für jede Gruppe mit t Charakteristiken.
    5. **Base Choice** (BC): Für jede Charakteristik wird eine "Base Choice" gewählt. Dann wird immer ein Teil davon variiert. $1 + \sum_{i=1}^{Q} B_i - 1$
    6. **Multiple Base Choice** (MBC): Eine oder mehr "Base Choice" werden gewählt für jede Charakteristik.







Spezifikationen um Testfälle zu generieren

Final State Machines

- Repräsentiert durch Graphen. Kanten stellen

Bedingungen oder Events dar.

- Zustandsbasiertes Testen hat als Ziel eine komplette Abdeckung der Zustandsmaschine.
- Testkriterien:
  - Alle Events → alle Transaktionen → alle Zustände
  - Ungültige Fälle (Events im falschen Zustand aufrufen)

Cause-Effect-Graphing & Decision Tables

- Graphische Beschreibung der Abhängigkeiten
- Mit einer Analyse können so unvollständige Spezifikationen/Widersprüche entdeckt werden

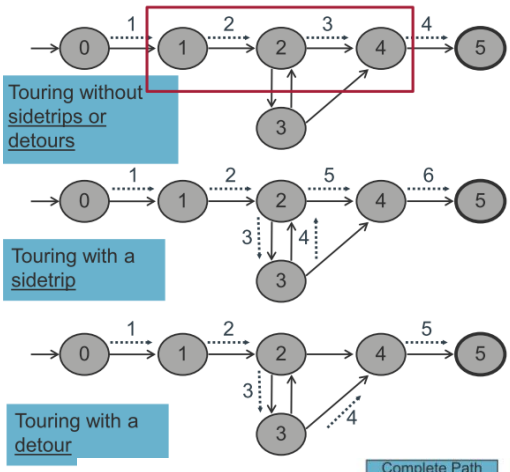
Kapital 4 - Graph Coverage and Algorithms

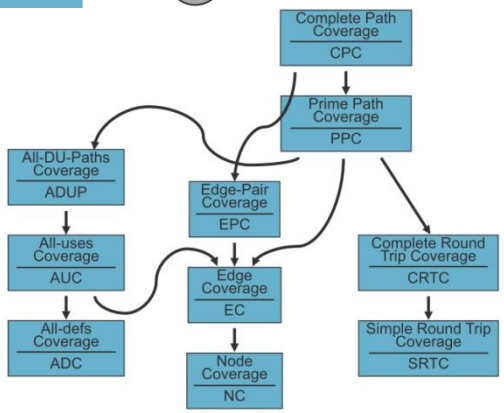
Grundbegriffe:

- SESE (single-entry, single exit) Graph
- "Syntactic reach": Ein Pfad existiert mit dem der Knoten erreicht werden kann
- "Semantic reach": Ein Test existiert der den Pfad ausführen kann.

Coverages:

- **Node Coverage** (NC), **Edge Coverage** (EC)
- **Edge-Pair Coverage** (EPC): Beinhaltet jeden erreichbaren Pfad der Länge 2.
- **Complete Path Coverage** (CPC): Beinhaltet alle Pfade.
- **Specified Path Coverage** (SPC): so viel wie der Tester für nötig hält ;)
- *Simple Path*: Ein Pfad der keinen Knoten doppelt durchläuft. (z.B. {1,2}, {1,2,1})
- *Prime Path*: Ein "simple Path" der kein Teilpfad eines anderen "simple Path" ist
- **Prime Path Coverage** (PPC): Beinhaltet alle "Prime Path"
- *Round-Trip Path*: Ein Pfad der am selben startet und endet.
- **Simple Round Trip Coverage** (SRTC): Beinhaltet mindestens einen "round-trip-path" für jeden erreichbaren Knoten der mit einem "round-trip-path" beginnt und endet.
- **Complete Round Trip Coverage** (CRTC): Alle "round-trip-path"
- *Tour With Sidetrips/Tour With Detours*:
- *DU pair*: Paar aus Knoten das defined and used
- *Def-clear*: Ein Pfad bei dem eine Variable nicht neu definiert wird.
- *du-path*: Ein "simple" Teilpfad der def-clear ist.
- *du(n_i,v)*: die Menge an du-paths die an n_i starten
- du(n_i,n_j,v): die Menge an du-paths von n_i bis n_j
- **All-defs coverage** (ADC): Für jede Menge von du(n,v) ist mind. ein Pfad enthalten
- **All-uses coverage** (AUC): Für jede Menge an du(n_i,n_j,v) ist mind. ein Pfad enthalten
- All-du-paths coverage (ADUPC): Für jede Menge du(n_i,n_j,v) sind alle Pfade enthalten





Formale Sprache über Graphen (RegEx)

Definition: +, x*/x^n
Anzahl an Pfaden in einem Graph (Beispiel):

$$pe = (b + c)\big(d(b + c)\big)^2 = (1 + 1) * \big(1 * (1 + 1)\big)^2 = 2 * (\sum_{i=0}^{2} 2^i)$$

Minimale Anzahl an Pfaden um alle Kanten zu erreichen (Beispiel):

$$pe = 2 * (1 * (2))^2 = 2 * (1 * 2) = \max(2,1,2) = 2$$

Kapitel 5 - Specification and Design Coverage

- **OO Call Coverage**: Beinhaltet jeden erreichbaren Knoten im Call-Graphen ... ?
- **OO Object Call Coverage**: Beinhaltet jeden erreichbaren Knoten im Call-Graphen ... ?

Covering FSMs:

- **Node Coverage**: Jeden Zustand ausführen (State Coverage)
- **Edge coverage**: Jede Transition ausführen (Transition Coverage)
- **Edge-pair coverage**: Jedes Paar von Transitions

Kapitel 6 - Code Coverage

Statement Coverage (C_0)

- Entdeckt nicht ausgeführte Codeteile
- Problem: Schleifen, Logic Operators

Branch Coverage (C_1)

- schließt Statemente Coverage ein
- löst einige Probleme der Statement Coverage

Path Coverage (C_2)

- Alle Pfade zwischen Start und Endknoten
- schließt Branch Coverage ein
- Verwendet Loops aber iteriert nicht darüber
- Benötigt exponential viele Testfälle

Condition Coverage (C_3)

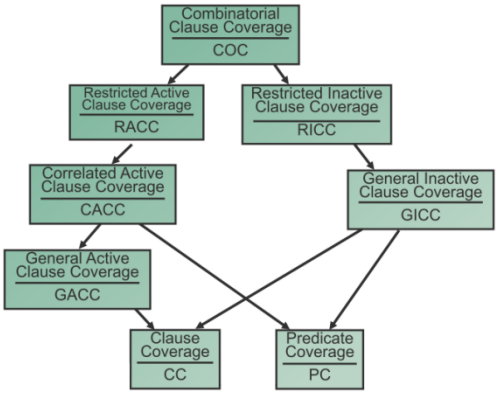
- *Simple Condition Coverage* (C_{3a}): Jeder Ausdruck muss mindestens einmal wahr und falsch sein.
- *Multiple Condition Coverage* (C_{3b}): Jede mögliche Kombination aus Ausdrücken muss zu wahr und falsch ausgewertet werden.
- *Minimal Multiple Condition Coverage* (C_{3c}): Jede atomar, kombinierte und kompletter Boolischer Ausdruck muss zu wahr und falsch ausgewertet werden.
- ergibt Riesige Menge an Testfällen

Weitere Coverage-Metriken

- Call Coverage: Wurde jede Funktion mindestens einmal ausgeführt?
- Class Coverage: Wurde jede Klasse mindestens einmal benützt?
- Race Coverage: Ist Code von verschiedenen Threads gleichzeitig ausgeführt?

Kapitel 7 - Coverage Logic

- P ist eine Menge von Prädikaten, p ein einzelnes Prädikat, C ist die Menge an Klauseln, c ist eine einzelne Klausel
- **Predicate Coverage** (PC): Für jedes Prädikat in P wird gefordert, dass p zu wahr und zu falsch ausgewertet wird.
- **Clause Coverage** (CC): Für jede Klausel in C wird gefordert, dass c zu wahr und zu falsch ausgewertet wird.
- **Combinatorial Coverage** (CoC): Für alle Clauses alle Kombinationen.
- *Determination*: Eine Klausel c_i im Prädikat p wird „**major clause**“ bezeichnet, wenn allein das ändern von c_i eine Änderung von p hervorruft.
- **Active Clause Coverage** (ACC): Für jedes Prädikat in P und jeden Major Clause c_i, wähle die „minor clauses“ c_j, sodass c_i p bestimmt. Für jedes c_i wertet c_i zu wahr und falsch aus.
- **General Acctive Clause Coverage** (GACC): Wie AAC, nur dass die Minor Clauses beim selben c_i nicht gleich sein müssen
- **Restricted Active Clause Coverage** (RACC): Wie AAC, nur dass die Minor Clauses beim selben c_i gleich sein müssen
- **Correlated Active Clause Coverage** (CACC): Wie GACC nur, dass das gesamte Prädikat nun auch wahr und falsch sein muss
- **Inactive Clause Coverage** (ICC): Für jedes Prädikat in P und jeden Major Clause muss folgendes gelten. Wähle die minor clauses so, dass c_i nicht p bestimmt. Insgesamt alle Kombinationen aus p (wahr/falsch) und c_i (wahr/falsch)



- **General Inactive Clause Coverage** (GICC): Im Prinzip ICC
- **Restricted Inactive Clause Coverage** (RICC): Wie ICC, nur es muss nur folgendes gelten: Für alle c_j $c_j(c_i = \text{true}) = c_j(c_i = \text{false})$
- Prädikat bestimmen: $p_n = p_{n=\text{true}}$ XOR $p_{c=\text{false}}$ (Wenn p_n = false, dann ist n irrelevant)

Kapitel 8 – Formal Methods and Automated Deduction

- Statische Methoden führen das Programm nicht unter Test aus, sondern analysieren den Quellcode
- **Partial Correctness**: Wenn P erfüllt ist, dann garantiert jede Ausführung von S Q.
- **Total Correctness**: Garantiert noch zusätzlich die partielle Korrektheit der Termination von S
- **Valid**: Formel ist immer wahr (Tautologie)
- **Invalid/Satisfiable**: Formel kann wahr oder falsch sein
- **Unsatisfiable**: Formel ist immer falsch (Kontradiktion)
- **Predicate Logic** ($\exists x: x = y^2$)
- **Temporal Logic** ("Ich bin hungrig bis ich was gegessen habe")
- **Hoare-Calculi**: Wird benützt um über die Korrektheit eines Programs zu urteilen. (nicht relevant)
- **Deduction Systems**: Basierend auf wahren Vorraussetzungen versucht es Schlüsse zu finden die auch wahr sind. (z.B. Hilbert-System, Andrew's System P)

Automated Proof Methods (mit BDD)

TODO

Kapitel 9 - BMC and SAT Solving (Formal Verification)

Idee: Suche nach einem Gegenbeispiel mit der maximalen Länge k. → erstelle eine Formel die nur wahr ist, wenn es ein Gegenbeispiel existiert

BMC (Bounded Model Checking)

- Findet schnell mit SAT Solving und gleichzeitig das kürzeste Gegenbeispiel
- Analyse ist unvollständig, da Schleifen nur endlich oft durchlaufen werden
- Algorithmus:
  1. Vereinfache den Quellcode (i++ zu i = i + 1, alle Schleifen in while umschreiben, Seiteneffekte entfernen, continue/break zu goto)
  2. Schleifen abhängig von k oft ausrollen (falls k+1 assert werfen)
  3. Zuweisungen zu SSA (Single Static Assignment) umwandeln (nur 1 Def pro Variable)
  4. In Gleichungen umwandeln
  5. Bit-blast
  6. Mit SAT Solver lösen
  7. Gegenbeispiel von SAT konvertieren falls vorhanden

SATisfiability Solving

- Beantwortet ob eine gegebene Formel P (üblicherweise in CNF) erfüllbar ist oder nicht.
- 3-SAT ist NP-vollständig und sogar NP-hart!
- Begriffe:
  - literal: positives/negatives auftreten einer Variable
  - pure literal: eine Variable die nur positiv oder negativ auftritt
  - unit clause: eine Klausel nur einer nicht festgelegten Variable
  - empty clause: eine Klausel bei der alle Variablen fest sind und die zu falsch auswertet
- Algorithmus:

```
DPLL(P) {
    if P is model return true;
    if P contains empty clause return false;
    foreach unit_clause l in P {
        P = unit_propagate(l,P);
    }
    foreach pure l in P {
        P = assign_pure_literal(l,P);
    }
    l = choose literal(P);
    return DPLL(P & l) or DPLL(P & -l);
}
```

Conflict Driven Clause Learning

- Konflikte treten auf, wenn es 2 unit clause gibt die sich widersprechen (z.B. $(x \vee y) \wedge (\bar{x} \vee z)$)
- Da solche Konflikte häufiger auftreten, lohnt es sich weitere Klauseln aufzunehmen, hier zum Beispiel $y \vee z$
- Welche Variable zuerst bestimmt wird, wird durch üblicherweise mit Heuristiken bestimmt

Alternative SMT Solver

- Kombiniert beweisen und lösen
- Versucht mit Lemmas die Formel erst zu vereinfachen
- sind allerdings im Allgemeinen aber unvollständig