Teil 4: Testen und Verifikation

Falk Howar

Softwarekonstruktion WS 2018/19

LS14

Motivation

Software ist omnipräsent in modernen Geräten!

Bugs auch …

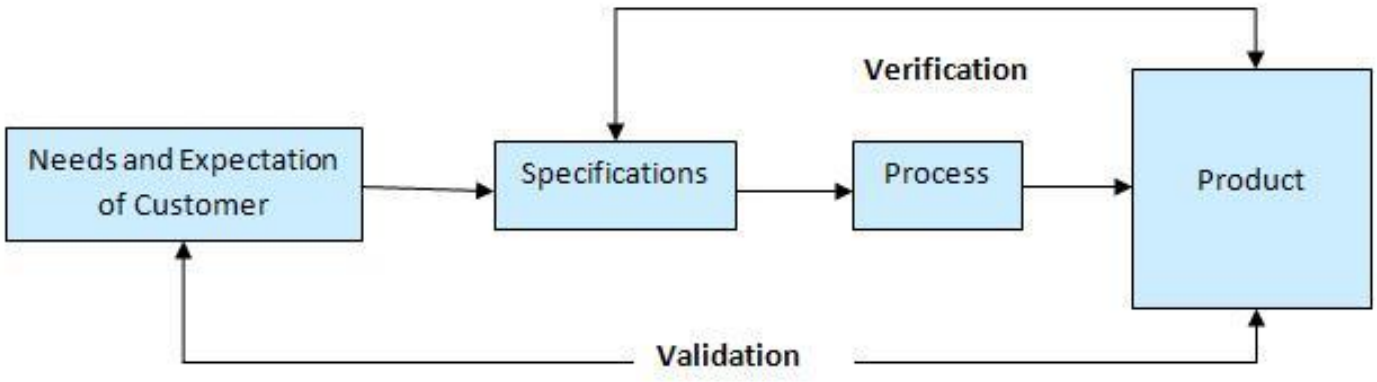
Bugs im Betrieb können wirklich teuer werden!

⇒ Methoden, um Fehler zu finden und zu verhindern

Verifikation vs. Validierung

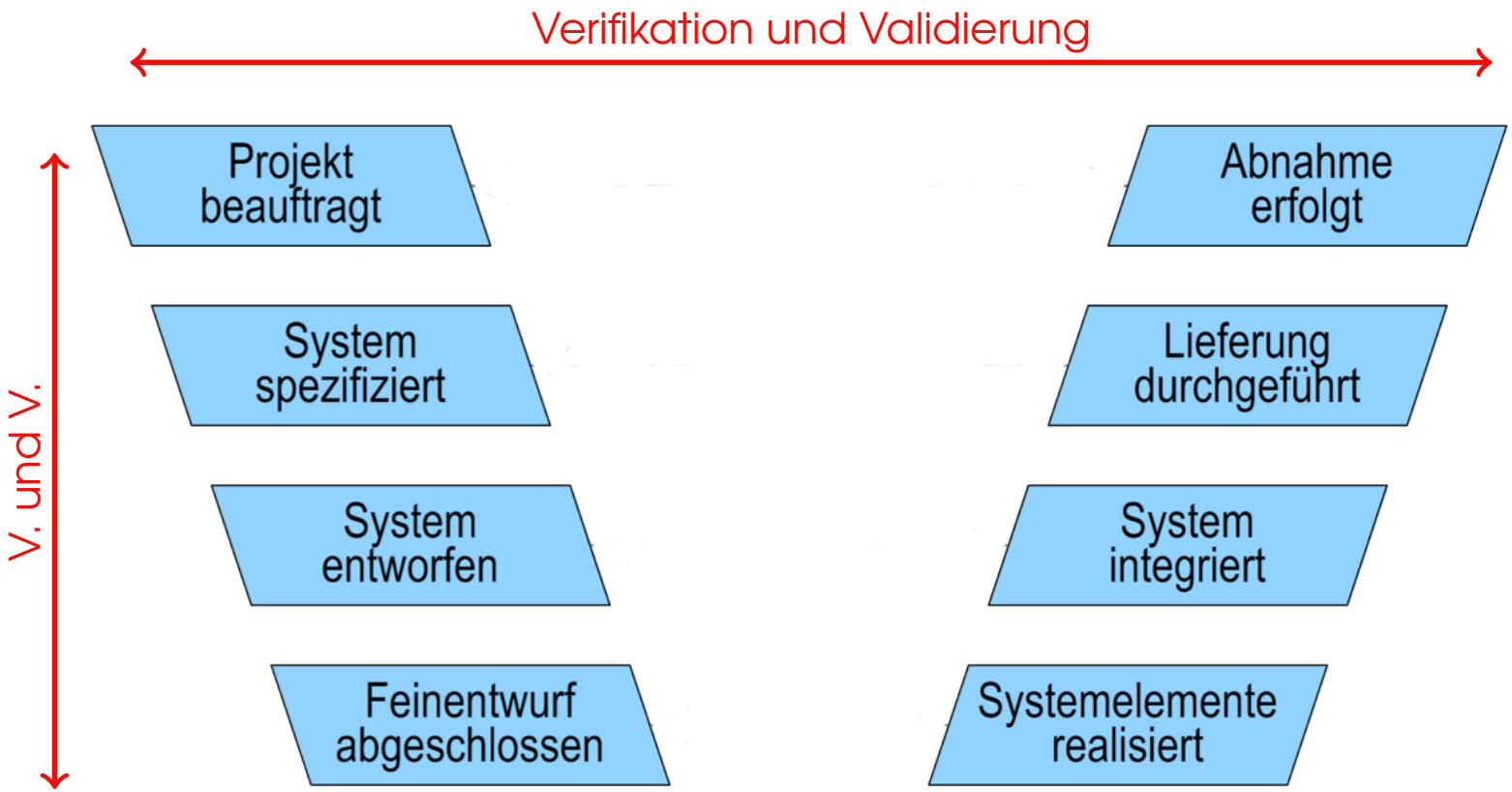
Verifikation: Bauen wir das Produkt richtig?

Validierung: Bauen wir das richtige Produkt?

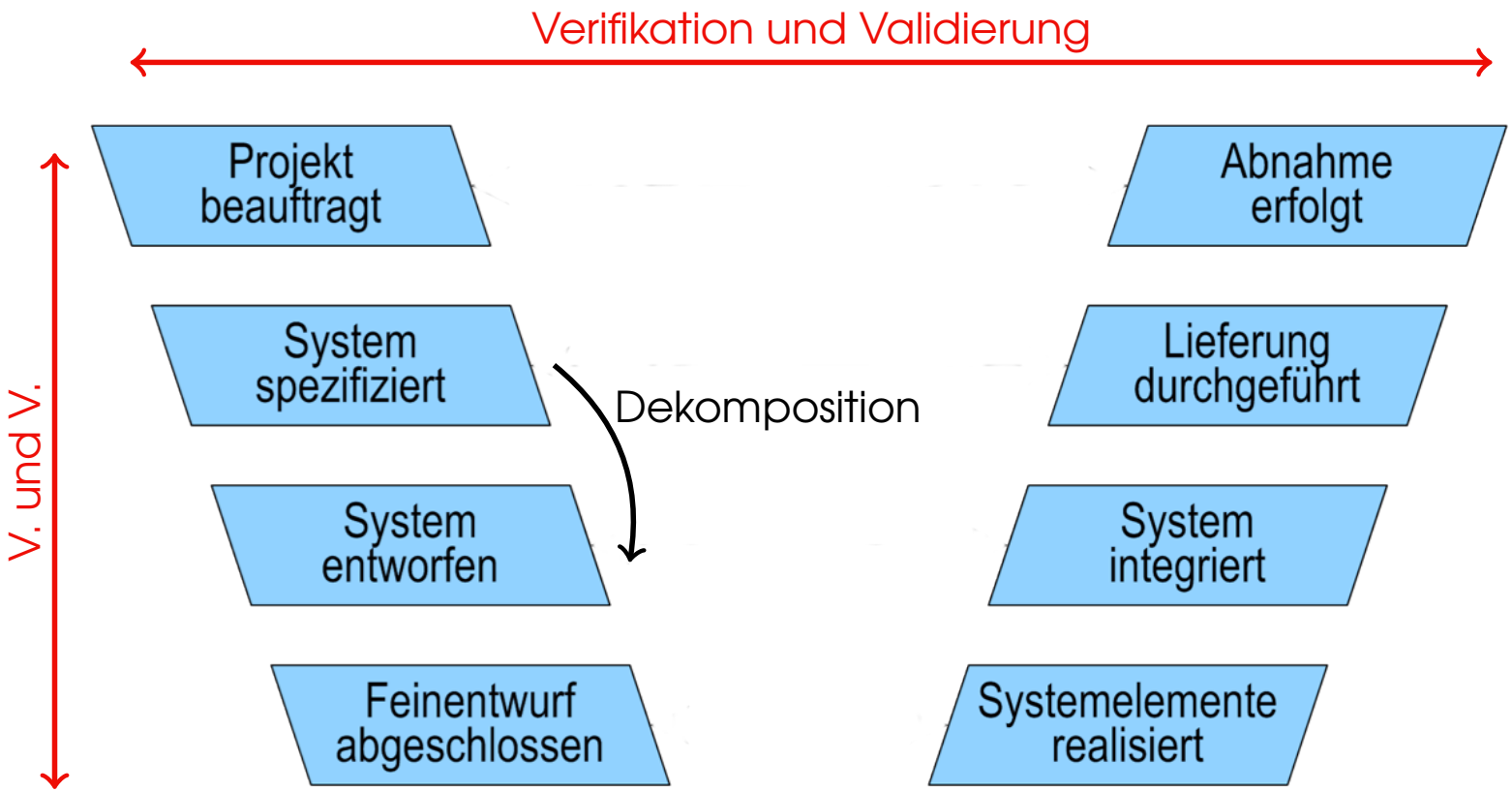


B. Boehm

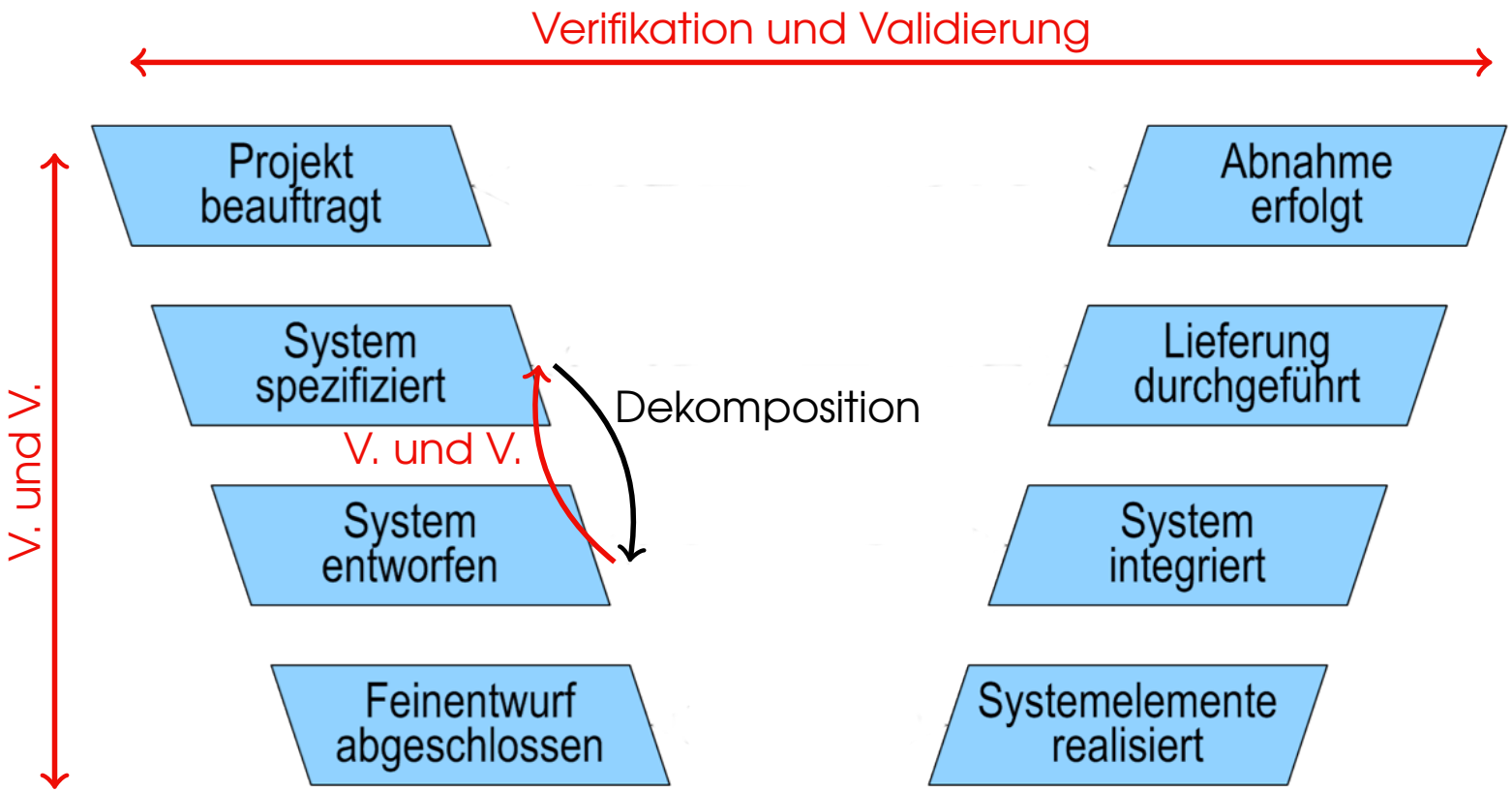
V & V im Entwicklungsprozess



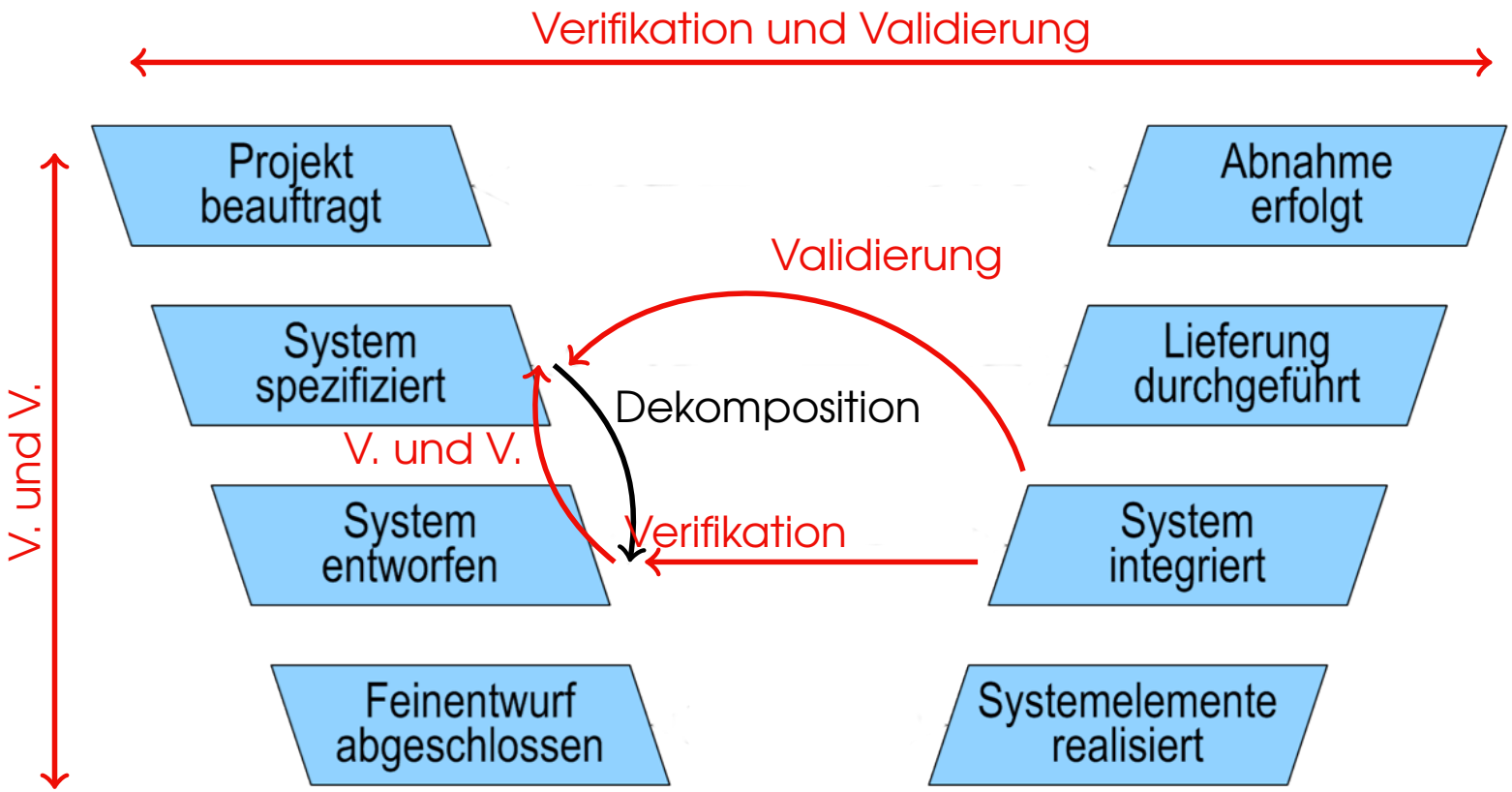
V & V im Entwicklungsprozess



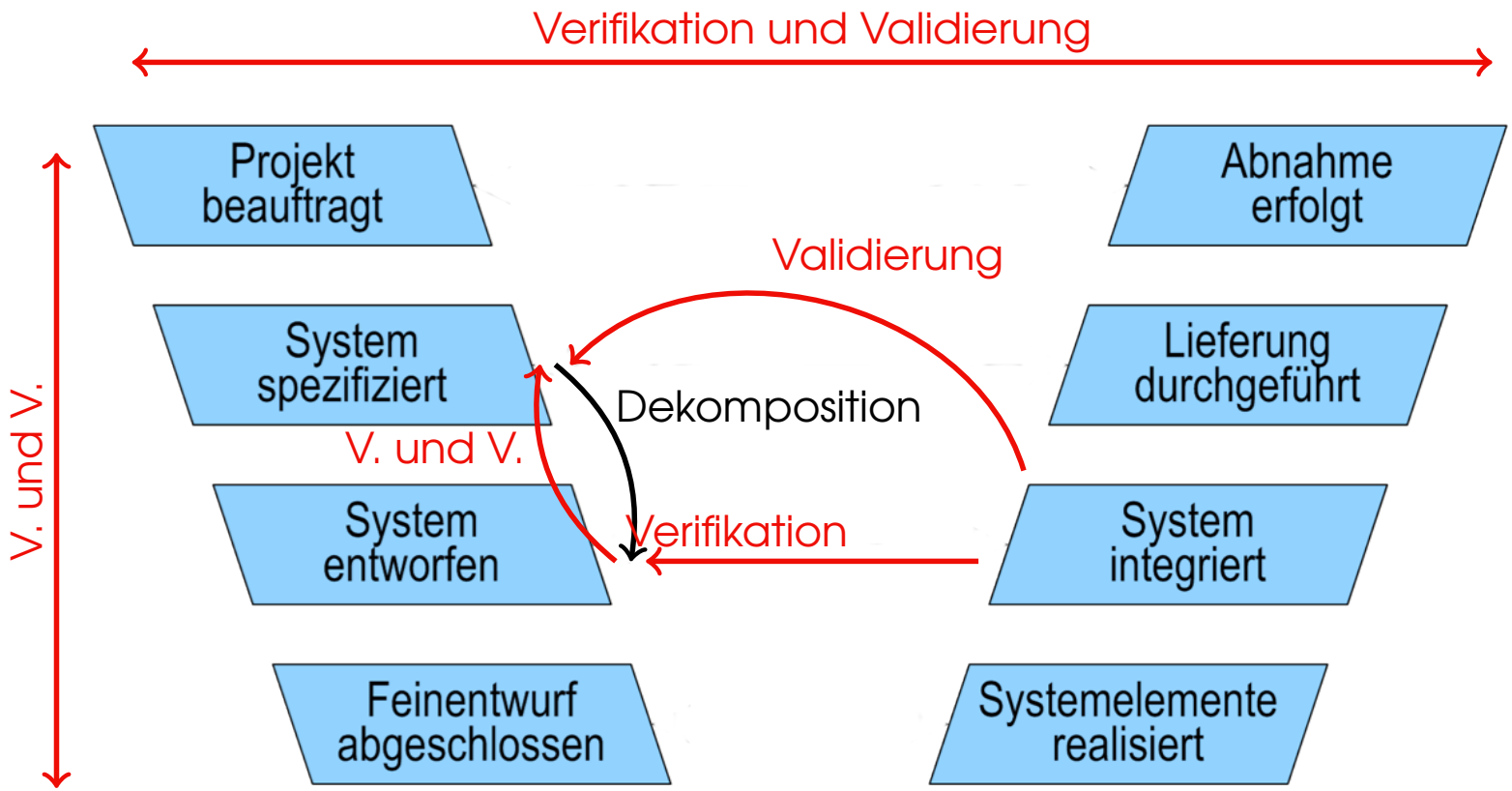
V & V im Entwicklungsprozess



V & V im Entwicklungsprozess



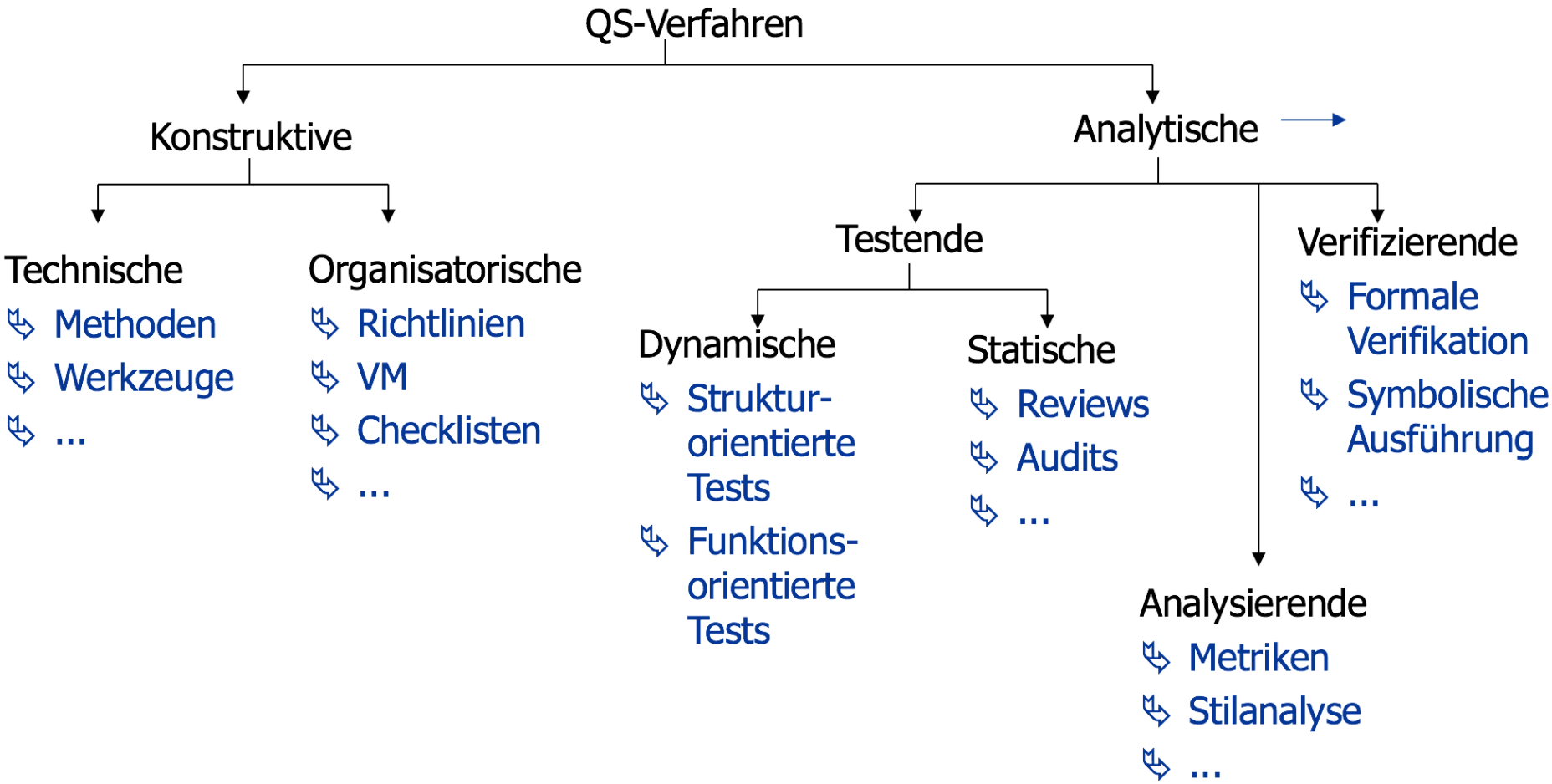
V & V im Entwicklungsprozess



$\Rightarrow$ Wann? In jedem Prozessschritt!

Jedes Artefakt des Entwicklungsprozesses kann verifiziert und validiert werden.

Wie? - Klassifizierung von Ansätzen



P. Liggesmeyer 2002

Genaue Analyse des Verhaltens

Funktioniert das Modul / die Komponente / das Produkt wie erwartet?

- Erfüllt es Ansprüche der Kunden?
- Erfüllt es die Spezifikation?
- Gibt es Ausfälle?

Voraussetzungen

- Analysierbares oder ausführbares System

$\Rightarrow$ Relativ spät im Prozess

$\Rightarrow$ Erkenntnisse über Verhalten des Systems

Semantik einer Programmiersprache

- In frühen Programmiersprachen war die Semantik in natürlicher Sprache definiert
- Konsequenz: Ungenauigkeit, nicht definiertes Verhalten

Beispiel:

```
integer procedure awkward
begin
  x := x+1
  awkward := 3
end awkward
```

- Globale Variable x.
- Was ist der Wert von x+awkward?

Formale Semantik

Idee: Formale Semantik für Programme, die Bedeutung eindeutig festlegt.

Strukturierte Operationelle Semantik:

- Strukturiert: Feste Regeln in strukturierter Darstellung
- Operationell: Beschreibt die Ausführung
- Semantik: Bedeutung der Syntax eines Prgramms

Beispiel: While Sprache.

While: Ausdrücke

$a \in \mathbf{AExp}$ Arithmetische Ausdrücke

$a ::= \; x \mid n \mid a_1 \; op_a \; a_2$

$b \in \mathbf{Bxp}$ Boolesche Ausdrücke

$b ::= \; \texttt{true} \mid \texttt{false} \mid \texttt{not}\; b \mid b_1 \; op_b \; b_2 \mid a_1 \; op_r \; a_2$

$x, y \in \mathbf{Var}$ Variablen

$n \in \mathbf{Num}$ Numerale

$op_a \in \mathbf{Op_a}$ Arithmetische Operatoren: $+, \; -, \; *, \; /, \; \ldots$

$op_b \in \mathbf{Op_b}$ Boolesche Operatoren: $\wedge, \; \vee, \; \ldots$

$op_r \in \mathbf{Op_r}$ Relationale Operatoren: $>, \; <, \; =, \; \ldots$

While: Anweisungen

$$
\begin{aligned}
S ::= \quad & [x := a]^l \\
& \mid [\texttt{skip}]^l \\
& \mid [S_1; S_2]^l \\
& \mid \texttt{if } [b]^l \texttt{ then } S_1 \texttt{ else } S_2 \\
& \mid \texttt{while } [b]^l \texttt{ do } S
\end{aligned}
$$

$\mathbf{Stmt}$: Menge aller syntaktisch korrekten Anweisungen.

Formale Semantik für While

Intuitiv brauchen wir:

- Abstrakte Maschine: Formaler Begriff eines Zustandes
- Ausführung: Übergänge zwischen Zuständen
- Konfiguration: Kombination aus Programm und Zustand

vgl. Markov Algorithmus, Registermaschine

Speicher = Belegung von Variablen

Def.: Belegung

Eine Belegung v von X ist eine Typ-konsistente Zuweisung von Werten zu Variablen in X. v ist eine Funktion und für $x \in X$ ist $v(x)$ der Wert mit dem x in v belegt ist.

- Typ-konsistent: $v(x)$ gehört zu Typ von x.
- Z.B., $v(x) \in \mathbb{Z}$ wenn x von Typ `Nat`.

Sei V_X die Menge aller möglichen typ-konsistenten Belegungen von X.

Update des Zustandes:
Wir schreiben: $v' = v[x \mapsto e]$ für:

$$v'(y) = \begin{cases} v(y) & \text{wenn } y \neq x, \\ e & \text{wenn } y = x. \end{cases}$$

Konfigurationen

Def.: Konfiguration

Eine Konfiguration ist ein Tupel $\langle S, v \rangle$ aus

- einem Programmtext $S \in \mathbf{Stmt}$ und
- einer Belegung $v \in V_X$.

Beispiel:

- Konfiguration $\sigma = \langle \texttt{if } (x > 0) \texttt{ then } x := 0 \texttt{ else skip}, \ [x \mapsto 2] \rangle$

Idee:

- S speichert den noch auszuführenden Teil des Programms
- v speichert den aktuellen Wert aller Variablen

Wir nehmen im folgenden immer an, dass $X \supseteq FV(S_\star)$
wenn wir das While Programm $S_\star$ analysieren.

Auswertung von Ausdrücken

- Set von Variablen X
- Ausdrücke aus $\mathbf{AExp} \cup \mathbf{BExp}$ über Variablen aus X

Auswertung:

- Funktion $\alpha_X : \mathbf{AExp} \times V_X \mapsto \mathbb{Z}$
- Funktion $\beta_X : \mathbf{BExp} \times V_X \mapsto \{0, 1\}$

Beispiel (Negation):

- $\beta_X[(\neg b), v] = 1$ gdw. $\beta_X[b, v] = 0$ (sonst 0)

Übergangsregeln

Wir definieren einige Regeln für das Abarbeiten von Anweisungen

Inferenzregeln:

Format: $\frac{\text{Prämissen}}{\text{Konklusion}}$

Beispiel: $\frac{a \Rightarrow b,\ b \Rightarrow c}{a \Rightarrow c}$

Axiome:

Format: $\frac{}{\text{Axiom}}$ (oder einfach: Axiom)

Beispiel: $n + 0 = n$

- Generell haben Prämisse, Konklusion und Axiome die Form $A \Rightarrow B$.
- Bedeutung: A kann in B überführt werden.
- Dabei sind A und B Konfigurationen oder Zustände

SOS: Zuweisung

Die Regel (Axiom) ASS gibt an, wie eine Zuweisung $x := a$ einen Zustand verändert.

$$[\text{ASS}] \qquad \langle x := a, v \rangle \;\Rightarrow\; v[x \mapsto c] \qquad\qquad \text{für } c = \alpha_X[a, v]$$

- Die Konfiguration $\langle x := a, v \rangle$ wird in den Zustand $v[x \mapsto c]$ überführt.
- Die Regel überführt eine Konfiguration in einen Zustand
- Nach der Überführung verbleibt nichts auszuführen

Beispiel:

$$\langle x := 5 + x,\ [x \mapsto 2] \rangle \;\Rightarrow\; [x \mapsto 7]$$

SOS: Skip

Die Regel (Axiom) $SKIP$ gibt an, wie eine `skip` Anweisung auszuführen ist.

$$[\text{SKIP}] \qquad \langle \texttt{skip}, v \rangle \;\Rightarrow\; v$$

- Die Konfiguration $\langle \texttt{skip}, v \rangle$ wird in den Zustand v überführt.
- Die Regel überführt eine Konfiguration in einen Zustand
- Nach der Überführung verbleibt nichts auszuführen

Beispiel:

$$\langle \texttt{skip}, \; [x \mapsto 2] \rangle \;\Rightarrow\; [x \mapsto 2]$$

SOS: Kontrollstrukturen

Die Regeln (Axiome) IF_T und IF_F geben an, wie eine `if then else` Anweisung auszuführen ist.

$$[\mathrm{IF_T}] \qquad \langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, v\rangle \ \Rightarrow\ \langle S_1, v\rangle \qquad \text{falls } \beta_X[b, v] = 1$$

$$[\mathrm{IF_F}] \qquad \langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, v\rangle \ \Rightarrow\ \langle S_2, v\rangle \qquad \text{falls } \beta_X[b, v] = 0$$

- Die Konfiguration $\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, v\rangle$ wird in die Konfiguration $\langle S_1, v\rangle$ überführt falls in s die Bedingung b gilt; falls b nicht gilt wird nach $\langle S_2, v\rangle$ überführt
- Die Regeln überführen eine Konfiguration in eine Konfiguration
- Nach der Überführung verbleibt etwas auszuführen

Beispiel:

$$\langle \texttt{if}\ (x > 0)\ \texttt{then}\ x := 0\ \texttt{else}\ \texttt{skip},\ [x \mapsto 2]\rangle \ \Rightarrow\ \langle x := 0,\ [x \mapsto 2]\rangle$$

SOS: Schleifen

Die Regeln (Axiome) WH_T und WH_F geben an, wie eine Schleife auszuführen ist. \ Die Schleife wird durch die Regeln einmal "abgerollt".

$$[\mathrm{WH_T}] \quad \langle \texttt{while}\ b\ \texttt{do}\ S, v\rangle \Rightarrow \langle S; \texttt{while}\ b\ \texttt{do}\ S, v\rangle \qquad \text{falls } \beta_X[b, v] = 1$$

$$[\mathrm{WH_F}] \quad \langle \texttt{while}\ b\ \texttt{do}\ S, v\rangle \Rightarrow v \qquad \text{falls } \beta_X[b, v] = 0$$

- Die Regel WH_T überführt eine Konfiguration in eine Konfiguration. Nach der Überführung verbleibt etwas auszuführen.
- Die Regel WH_F überführt eine Konfiguration in einen Zustand. Nach der Überführung verbleibt nichts auszuführen.

Beispiel:

$$\langle \texttt{while}\ (x > 0)\ \texttt{do}\ x := x - 1,\ [x \mapsto 2]\rangle \Rightarrow$$
$$\langle x := x - 1;\ \texttt{while}\ (x > 0)\ \texttt{do}\ x := x - 1,\ [x \mapsto 2]\rangle$$

SOS: Komposition

Kompositionsregeln (Inferenzregeln) geben an, wie Sequenzen von Anweisungen abzuarbeiten sind. Diese Regeln müssen immer mit einem Axiom kombiniert werden.

$$[\text{COMP}_1] \qquad \frac{\langle S_1, v\rangle \;\Rightarrow\; \langle S_1', v'\rangle}{\langle S_1; S_2, v\rangle \;\Rightarrow\; \langle S_1'; S_2, v'\rangle}$$

$$[\text{COMP}_2] \qquad \frac{\langle S_1, v\rangle \;\Rightarrow\; v'}{\langle S_1; S_2, v\rangle \;\Rightarrow\; \langle S_2, v'\rangle}$$

- Regel 1 funktioniert mit Regeln IF_T, IF_F und WH_T
- Regel 2 funktioniert mit ASS und $SKIP$ und WH_F
- Nach der Überführung verbleibt etwas auszuführen

Beispiel:

$$\frac{\langle x := 1,\ [x \mapsto 2]\rangle \;\Rightarrow\; [x \mapsto 1]}{\langle x := 1; \texttt{skip},\ [x \mapsto 2]\rangle \;\Rightarrow\; \langle \texttt{skip},\ [x \mapsto 1]\rangle}$$

Beispiel auf Papier

Transitionssysteme

Def.: Transitionssystem

Ein Transitionssystem $\mathcal{T}$ ist ein Tupel $\mathcal{T} = \langle \Gamma, \rightarrow, F \rangle$ aus

- möglichen Konfigurationen $\Gamma = (\mathbf{Stmt} \times V_X) \cup V_X$,
- Übergängen $\rightarrow \subseteq \Gamma \times \Gamma$, definiert durch Übergangsregeln (wie auf den vorangegangenen Folien definiert), und
- terminierenden Konfigurationen $F = V_X$.

Wir schreiben $\langle S, v \rangle \Rightarrow \langle S', v' \rangle$ wenn $(\langle S, v \rangle, \langle S', v' \rangle) \in \rightarrow$

- Eine Berechnung ist Sequenz von Konfigurationen $\langle S, v \rangle \Rightarrow \langle S_1, v_1 \rangle \Rightarrow \ldots$
- Eine Berechnung terminiert wenn irgendwann eine terminierende Konfiguration erreicht wird $\langle S, v \rangle \Rightarrow \ldots \Rightarrow v'$

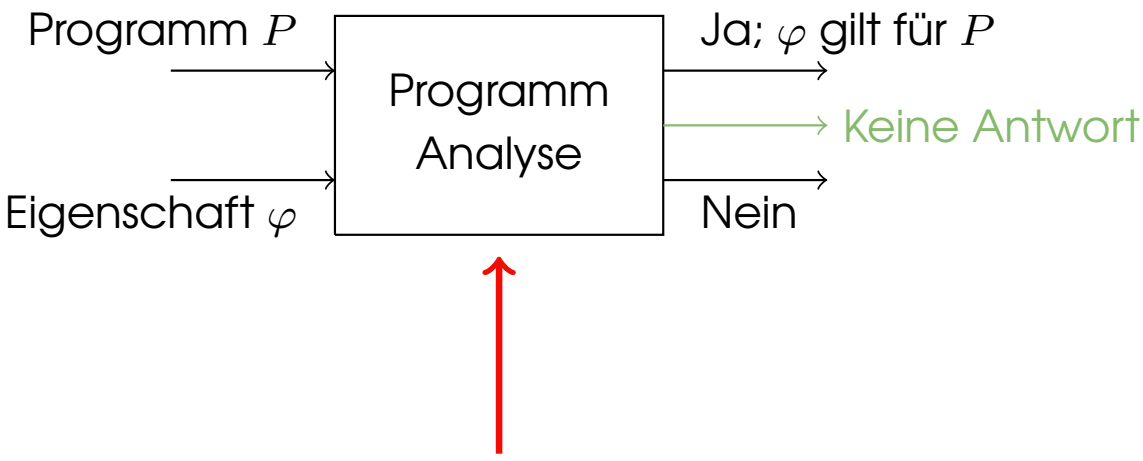
$\mathcal{T}$ beschreibt die Berechnungen
aller möglichen While Programme auf X

Semantik von While Programmen

- Die Semantik eines While Programms $S_\star$, bezeichnet durch $[\![S_\star]\!]$, ist die Menge der Berechnungen in $\mathcal{T}$, die in einer Konfiguration $\langle S_\star, v\rangle$ mit $v \in V_X$ beginnen.

- Ein Programm $S_\star$ terminiert immer wenn alle Berechnungen in $[\![S_\star]\!]$ terminieren

Programmanalyse



Beispiele bis jetzt:

- Alle Klassennamen beginnen mit Großbuchstaben
- Alle Variablen werden initialisiert
- Anweisung `L10` kann nicht erreicht werden

Eigenschaften von While Programmen

Eine Eigenschaft φ von Programm $S_\star$ ist ein Boolescher Ausdruck über Variablen aus X.

$[y := x]^1;$
$[z := 1]^2;$
`while` $[y > 1]^3$ `do`
  $[z := x * y]^4;$
  $[y := y - 1]^5$
$[y := 0]^6;$

Beispiel:

$$\varphi := \Big((z \geq 0) \ \wedge \ (y \neq 0)\Big)$$

Beobachtung:
Wir können in jeder Konfiguration $\langle S, v\rangle \in \mathcal{T}$ bzw. $v \in \mathcal{T}$ testen, ob φ gilt: $\beta_X[\varphi, v] = 1$?

Vorbedingungen, Nachbedingungen und Invarianten

Annahmen und Erreichbare Konfigurationen:

- Eine Vorbedingung $Init$ ist ein Boolescher Ausdruck über Variablen aus X und

$$[\![Init]\!] \;=\; \{\, \langle S_\star, v\rangle \in \Gamma \;:\; \beta_X[Init, v] = 1\}$$

- Erreichbare Konfigurationen $[\![Reach]\!]$ von $[\![Init]\!]$ aus: Konfigurationen in Γ, die durch Berechnungen erreicht werden, die in $\langle S_\star, v\rangle \in [\![Init]\!]$ starten

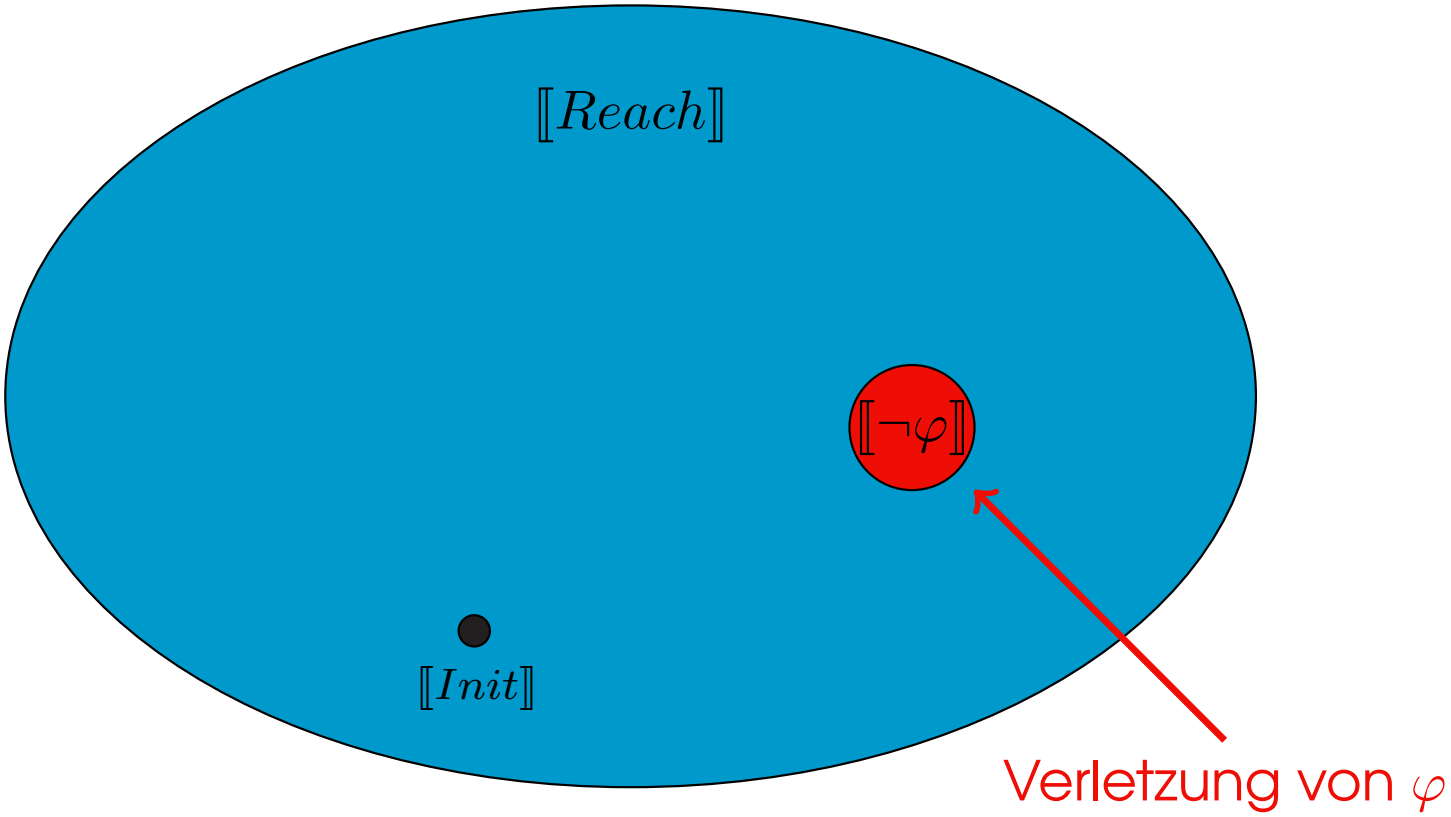
Invariante:

- Eine Invariante φ für $S_\star$ ist eine Eigenschaft von $S_\star$ die in allen erreichbaren Konfigurationen in $[\![Reach]\!]$ erfüllt ist.

Nachbedingung:

- Eine Nachbedingung φ für $S_\star$ ist eine Eigenschaft von $S_\star$ die in allen erreichbaren terminierenden Konfigurationen in $[\![Reach]\!]$ erfüllt ist.

Zustandsraum und Eigenschaften



Fehler von Analysen

- False Positive: Berichteter vermeintlicher Fehler
- False Negative: Übersehener echter Fehler

Ab jetzt auf englisch

Software Model Checking

Enumerative Seach

Enumerative Search

For some property φ, enumerative search

- traverses all reachable states of a transition system $\mathcal{T}$ (i.e., configurations σ in $[\![Reach]\!]$), and
- checks for every traversed configuration q if φ is satisfied in q.

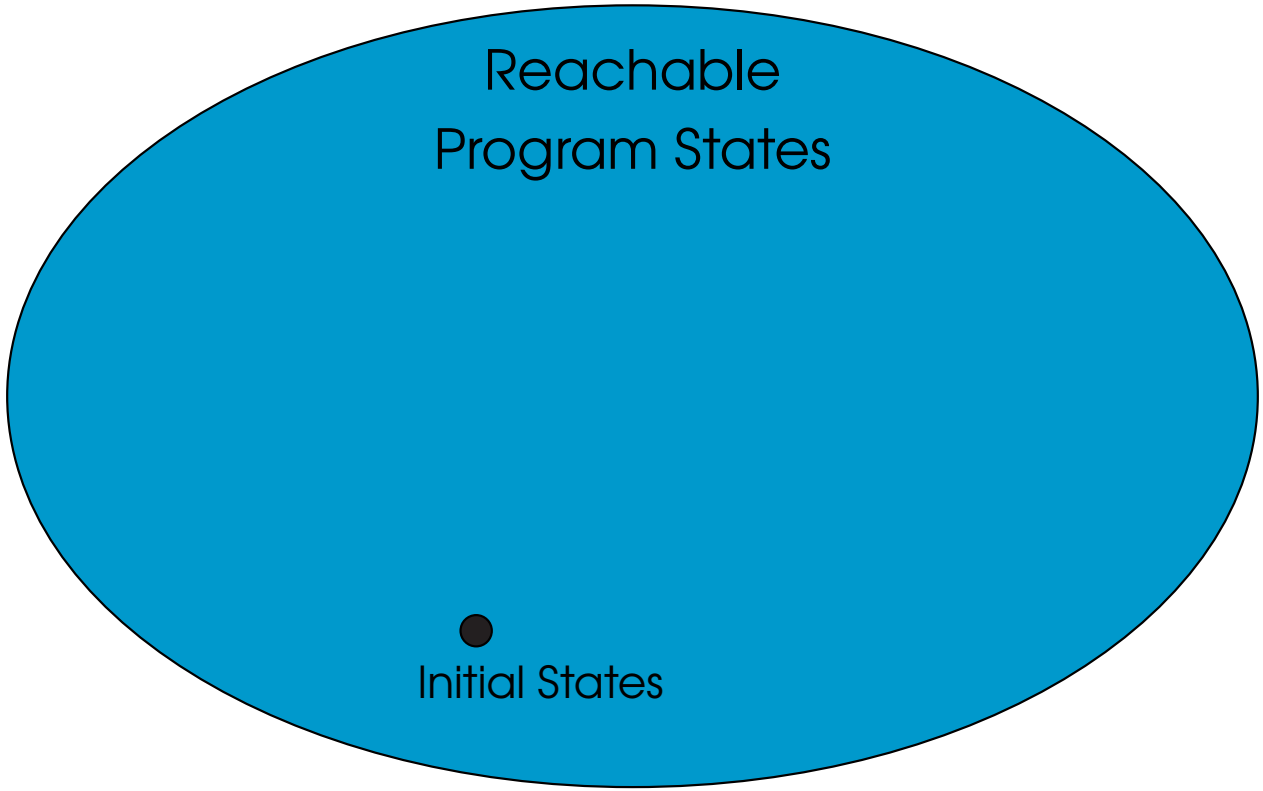
Search Strategies:

- DFS, BFS, ...
- Backtracking
- Challenge: Enumerating all next steps!

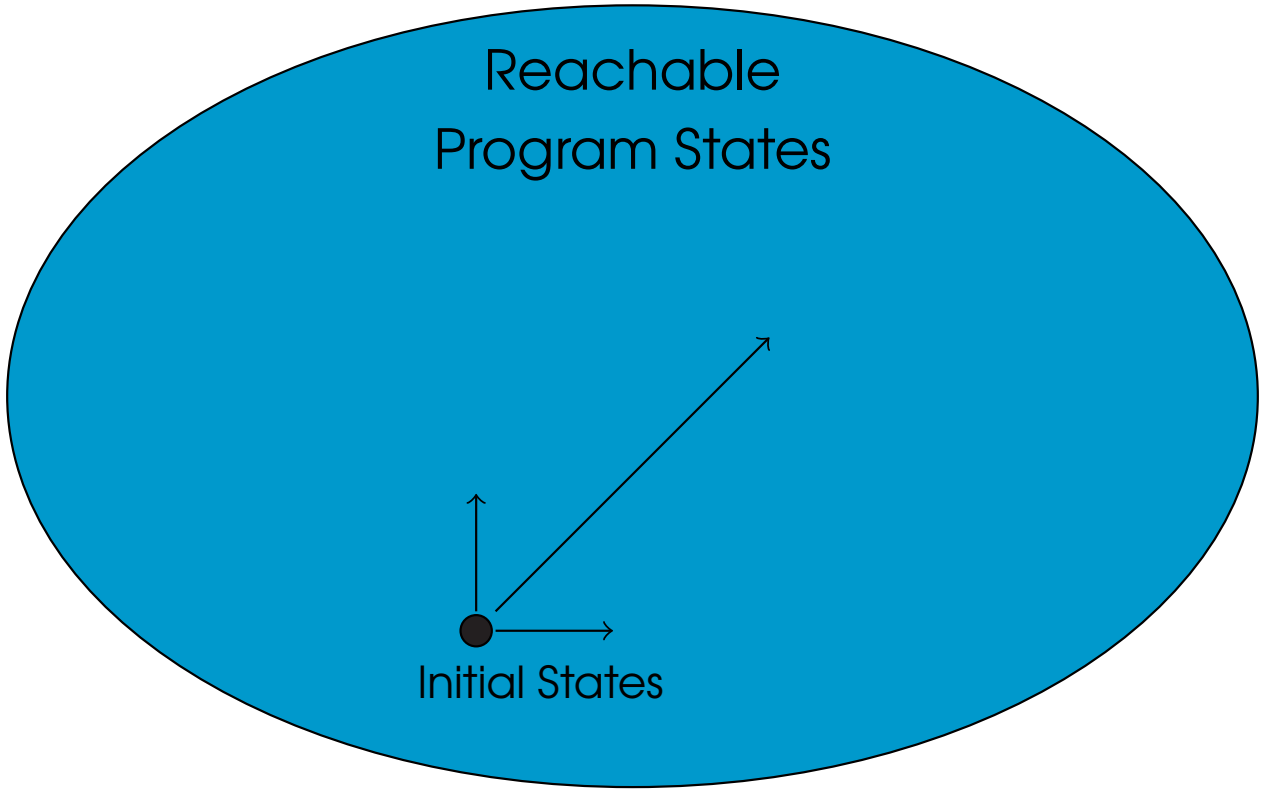
Challenges / Optimizations:

- Huge state space $\rightarrow$ Partial Order Reduction (POR)
- Big states $\rightarrow$ State Compression

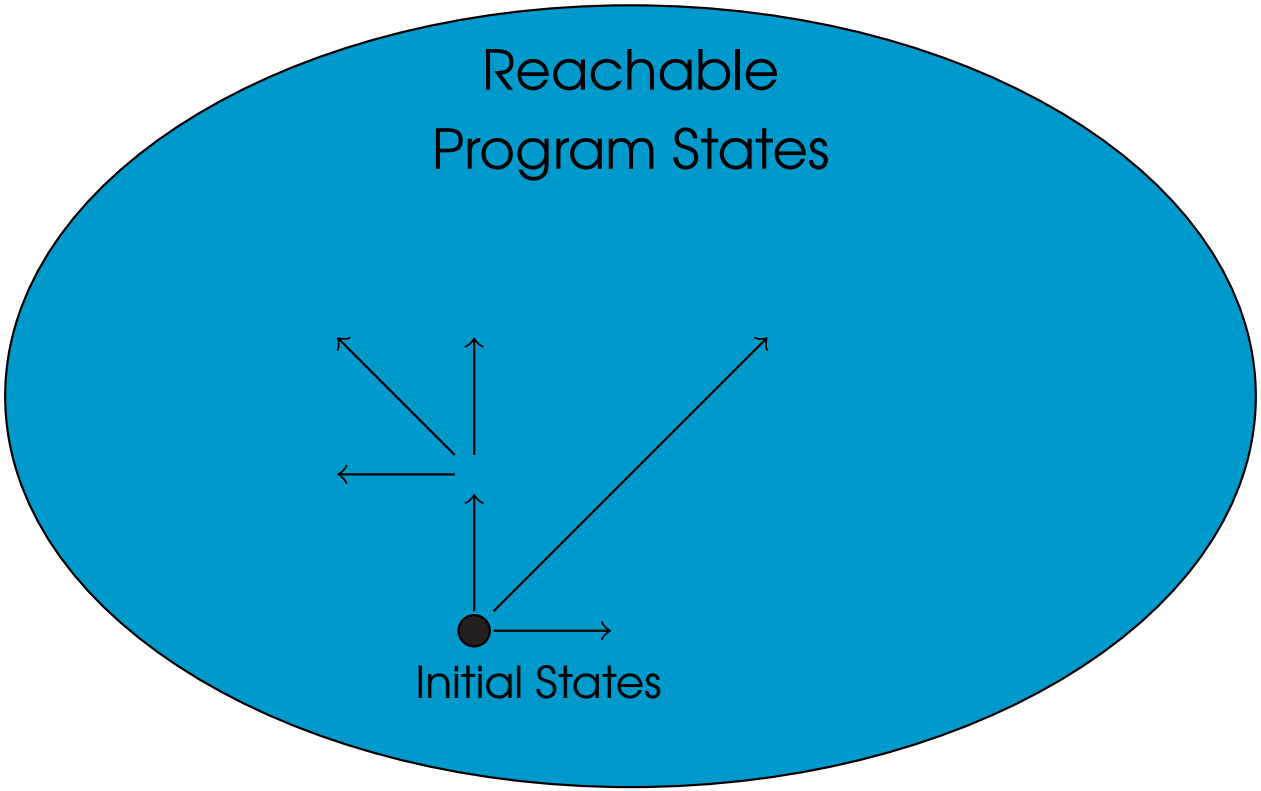
Enumerative Seach: Intuition



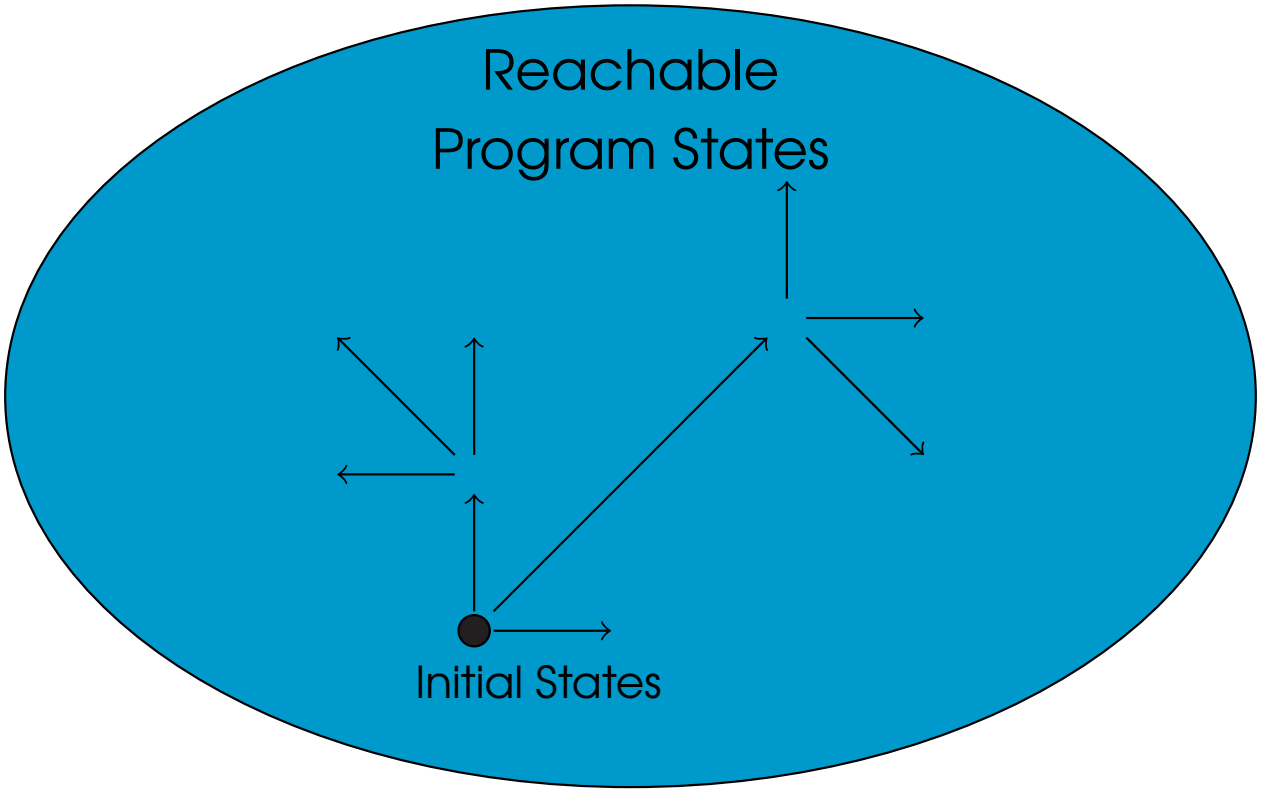
Enumerative Seach: Intuition



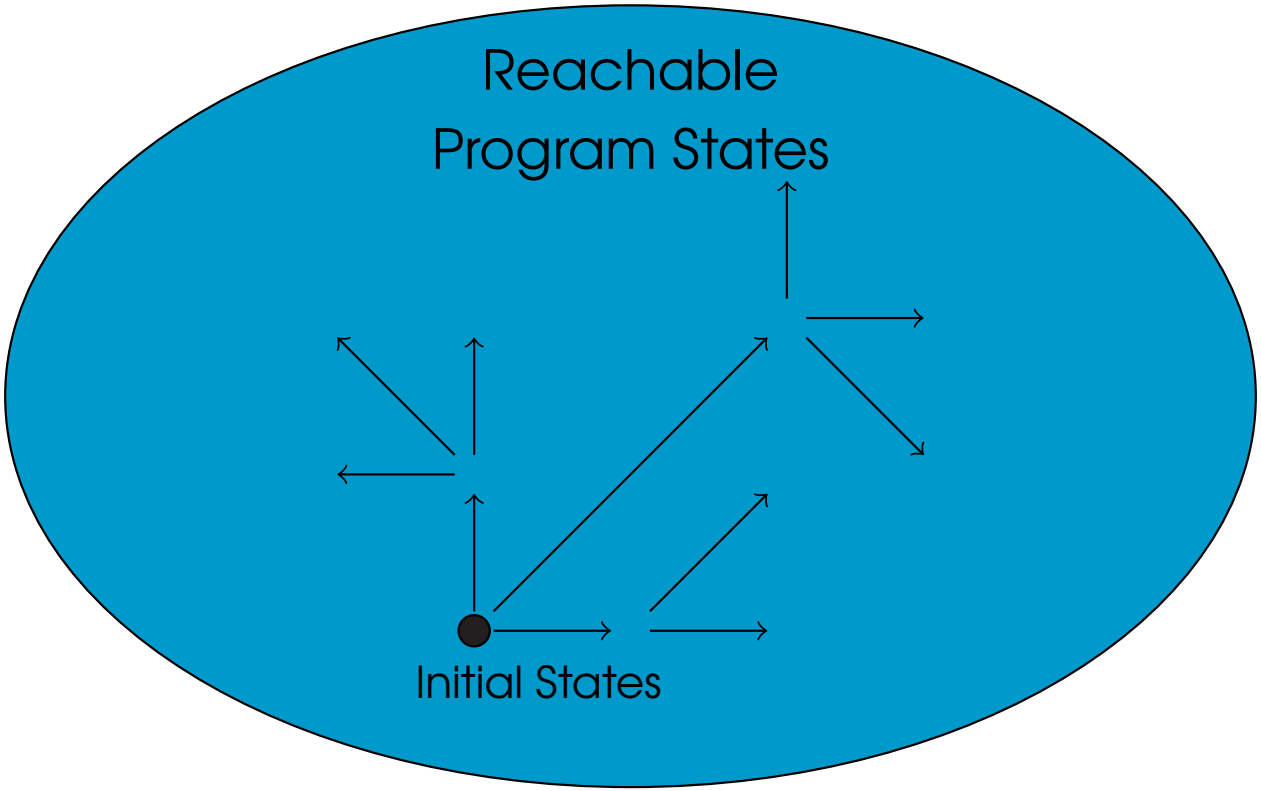
Enumerative Seach: Intuition



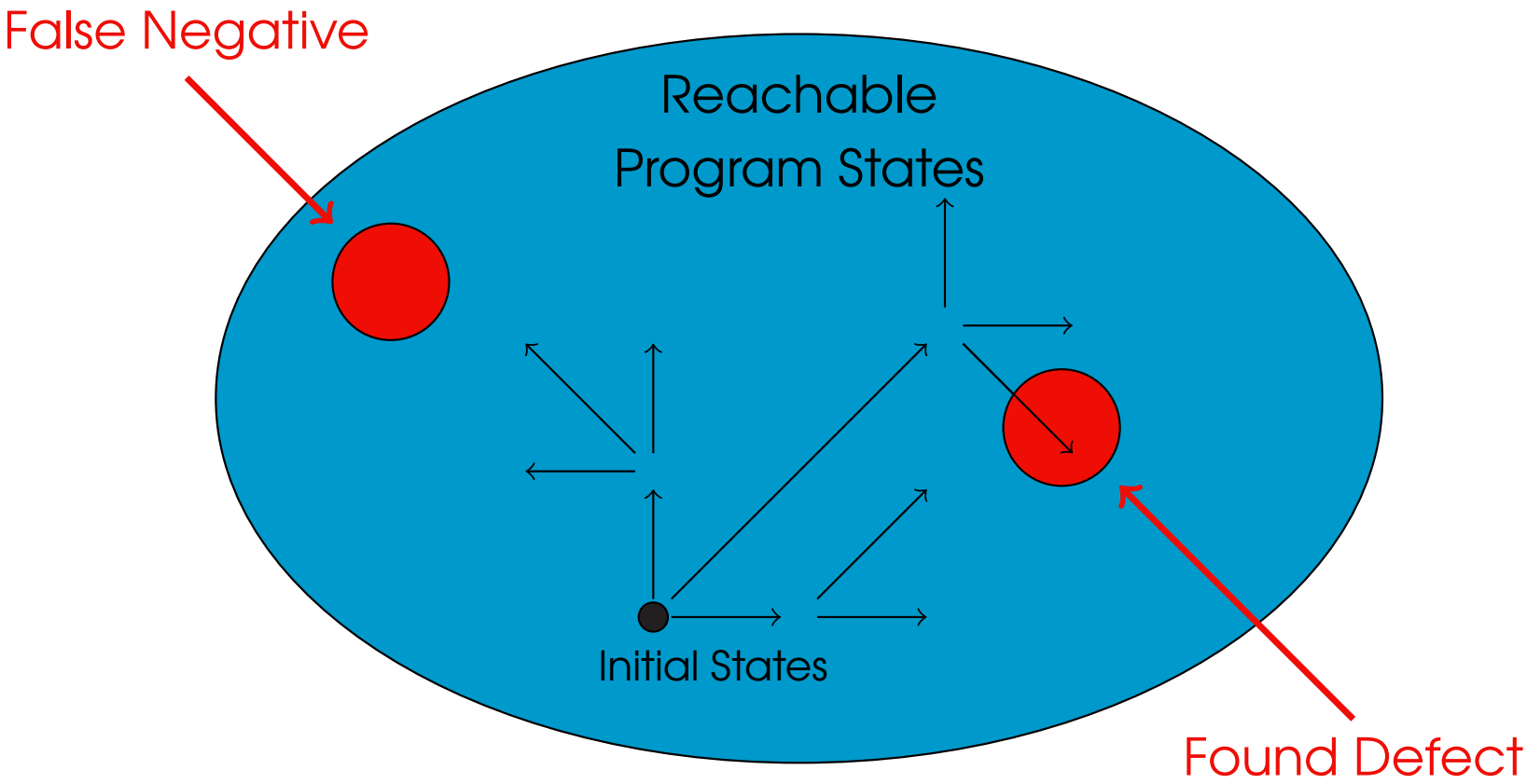
Enumerative Seach: Intuition



Enumerative Seach: Intuition



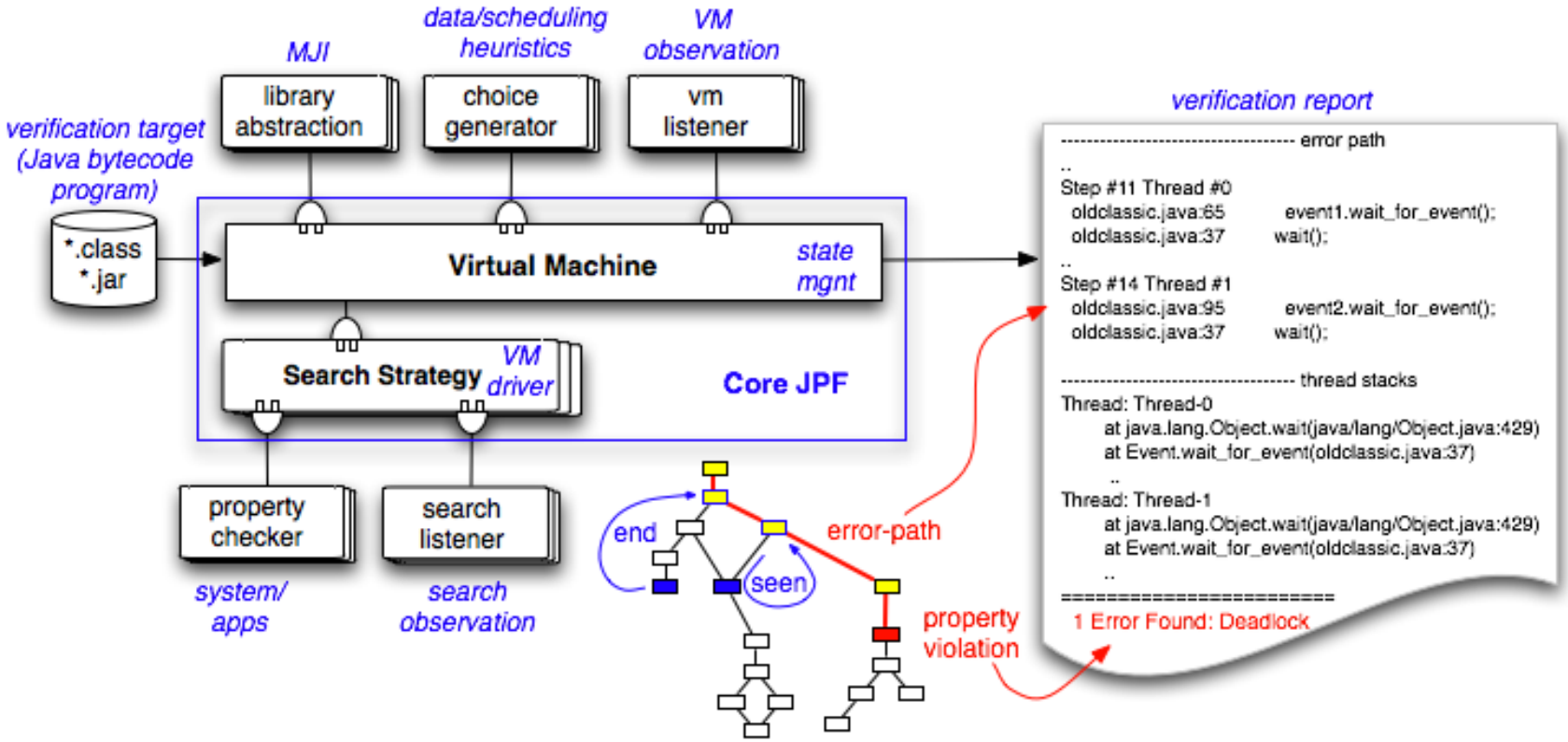
Enumerative Seach: Intuition



Enumerative Search: Properties

- False Positives? No: Explores reachable state space.
- False Negatives? Yes: Exploration not exhaustive.

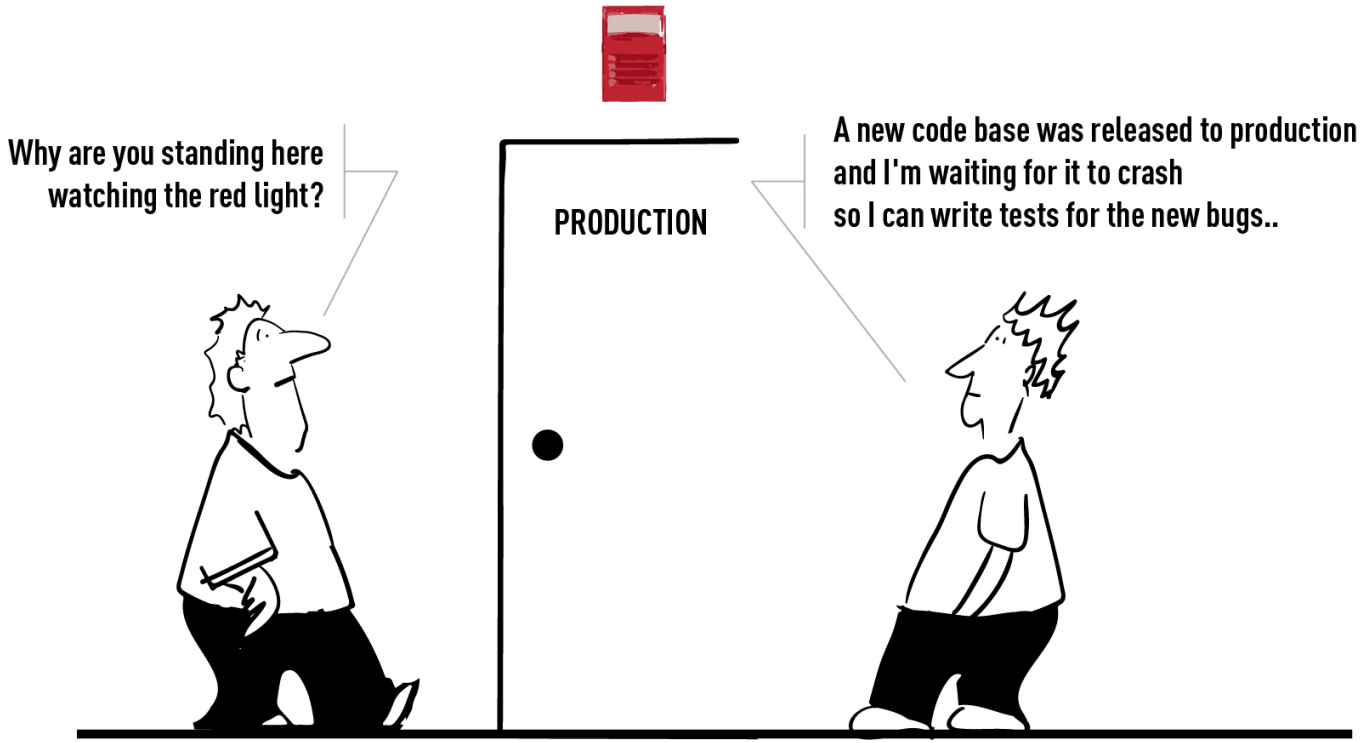
Java PathFinder



- Software Model Checker for Java
- Enumerative Search
- Extensions for Symbolic Analyses

Image from: `http://javapathfinder.sourceforge.net`

Testing



http://www.genrocket.com/software-testing-reactive-vs-proactive/

Program testing can be
a very effective way to show the presence of bugs,
but is hopelessly inadequate for showing their absence

Edsger Dijkstra

Test Case

Test Case

A test case is a scenario for which a tester verifies if a system works as expected. A test case consists of

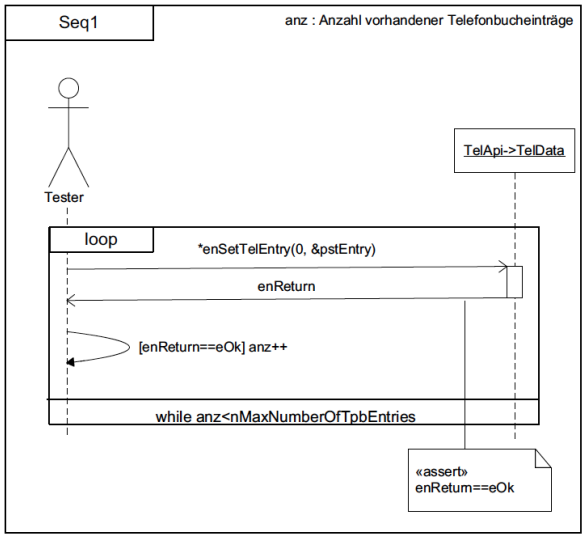
- a precondition, describing the state of the system before the test,
- an input, submitted to the system,
- a postcondition, expressing the expected state of the system after the test.

- May be text, or code
- May be formal or informal
- Postcondition sometimes referred to as test oracle

Example

Prüffall 1a

Szenario	Das Telefonbuch wird initialisiert und Einträge werden hinzugefügt ohne Überschreitung der Maximalanzahl der Einträge.
Eingangszustand	Es befindet sich eine bestimmte, von der Ausgangssituation (vgl. Kap. 3) abhängige Anzahl an Einträgen im Telefonbuch.
Durchzuführende Aktion	Das Telefonbuch wird mit Daten gefüllt. Die Anzahl möglicher Telefonbucheinträge wird nicht überschritten. Es werden so viele Telefonbucheinträge hinzugefügt, bis das Telefonbuch gefüllt ist.
Erwartetes Ergebnis	Als Rückgabe wird jeweils `eOk` als Element der Enumeration `tenError` geliefert.



Scenario

Precondition

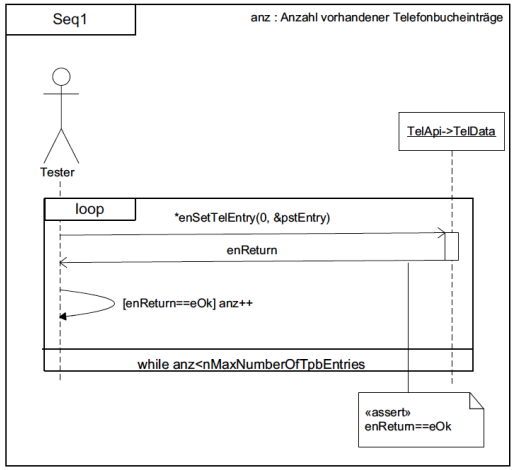
Input

Postcondition

Test Suite

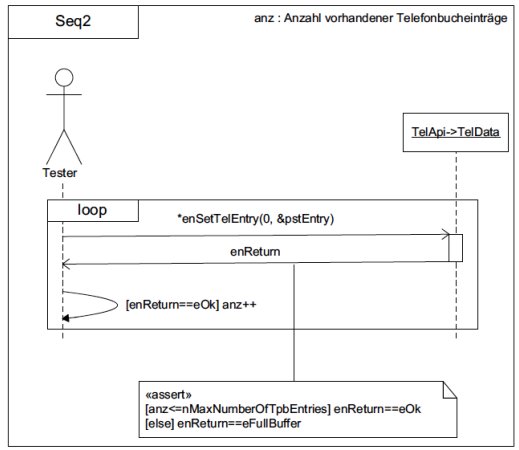
Prüffall 1a

Szenario	Das Telefonbuch wird initialisiert und Einträge werden hinzugefügt ohne Überschreitung der Maximalanzahl der Einträge.
Eingangszustand	Es befindet sich eine bestimmte, von der Ausgangssituation (vgl. Kap. 3) abhängige Anzahl an Einträgen im Telefonbuch.
Durchzuführende Aktion	Das Telefonbuch wird mit Daten gefüllt. Die Anzahl möglicher Telefonbucheinträge wird nicht überschritten. Es werden so viele Telefonbucheinträge hinzugefügt, bis das Telefonbuch gefüllt ist.
Erwartetes Ergebnis	Als Rückgabe wird jeweils `eOk` als Element der Enumeration `tenError` geliefert.



Prüffall 1b

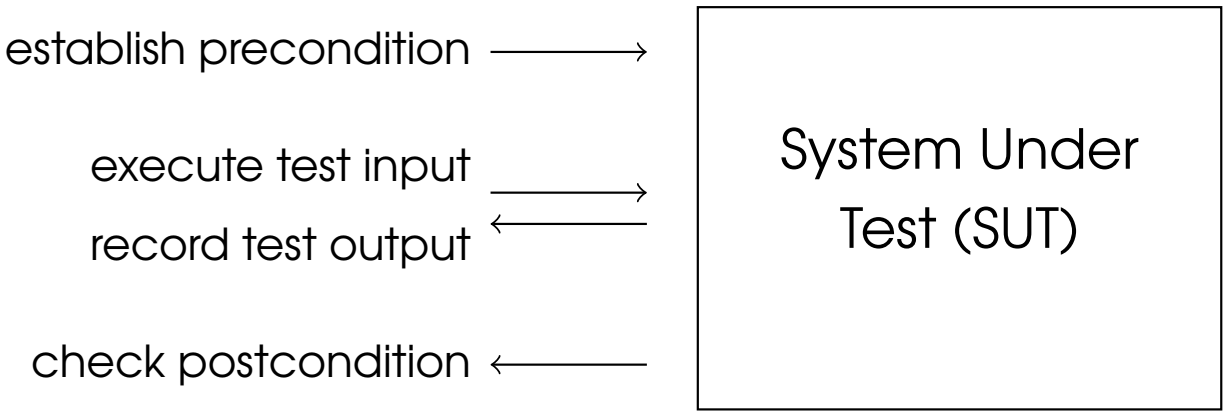
Szenario	Das Telefonbuch wird initialisiert und Einträge werden hinzugefügt. Hierbei wird die Maximalanzahl der Einträge überschritten.
Eingangszustand	Es befindet sich eine bestimmte, von der Ausgangssituation (vgl. Kap. 3) abhängige Anzahl an Einträgen im Telefonbuch.
Durchzuführende Aktion	Das Telefonbuch wird mit Daten gefüllt. Die Anzahl möglicher Telefonbucheinträge wird überschritten. Es werden so viele Telefonbucheinträge hinzugefügt, bis die maximale Anzahl möglicher Einträge überschritten ist.
Erwartetes Ergebnis	Beim Überschreiten der möglichen Anzahl an Telefonbucheinträgen wird `eFullBuffer` als Element der Enumeration `tenError` zurück gegeben.



...

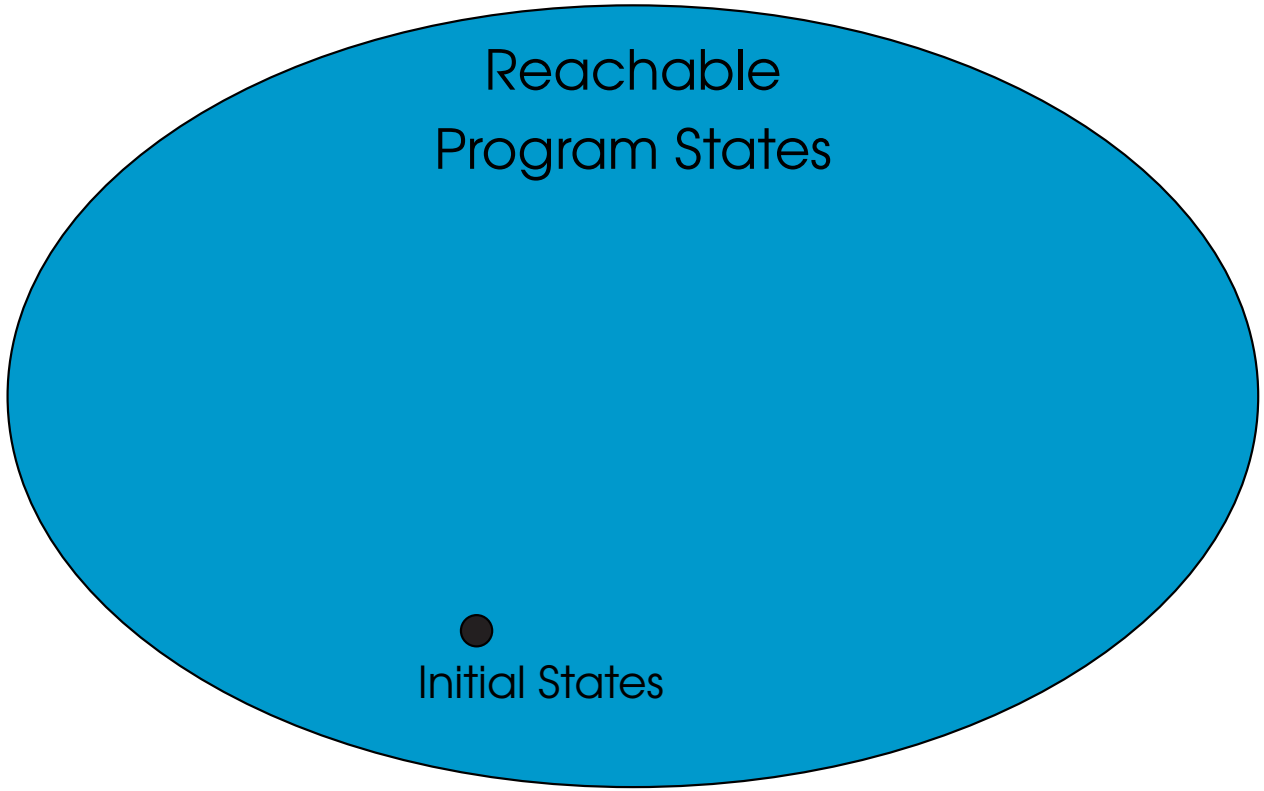
A test suite is a collection of test cases

Executable Test Cases

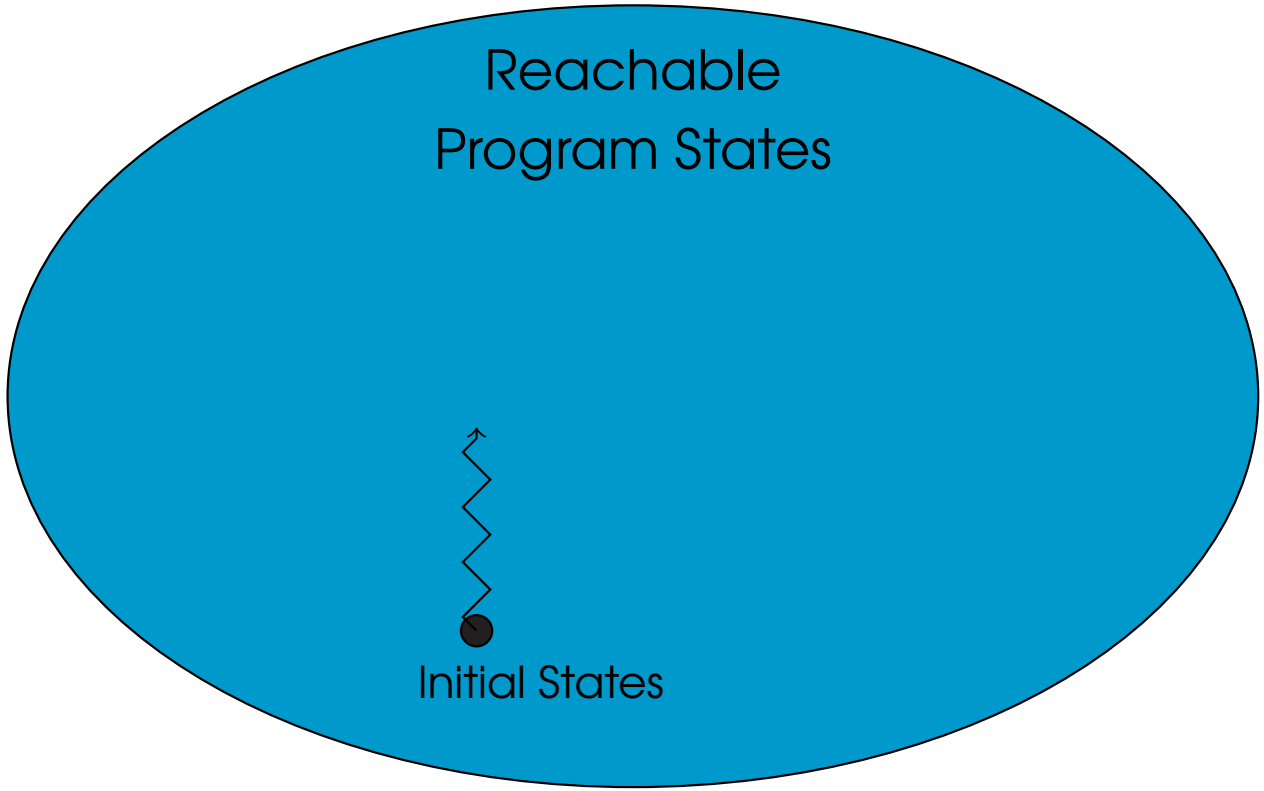



```
@Test
public void testGetEntries() throws Exception {
    ... // init
    sut.setEntry(entryA);
    sut.setEntry(entryC);

    List result = sut.getEntries(0, 2);

    assertEquals(2, result.size());
}
```

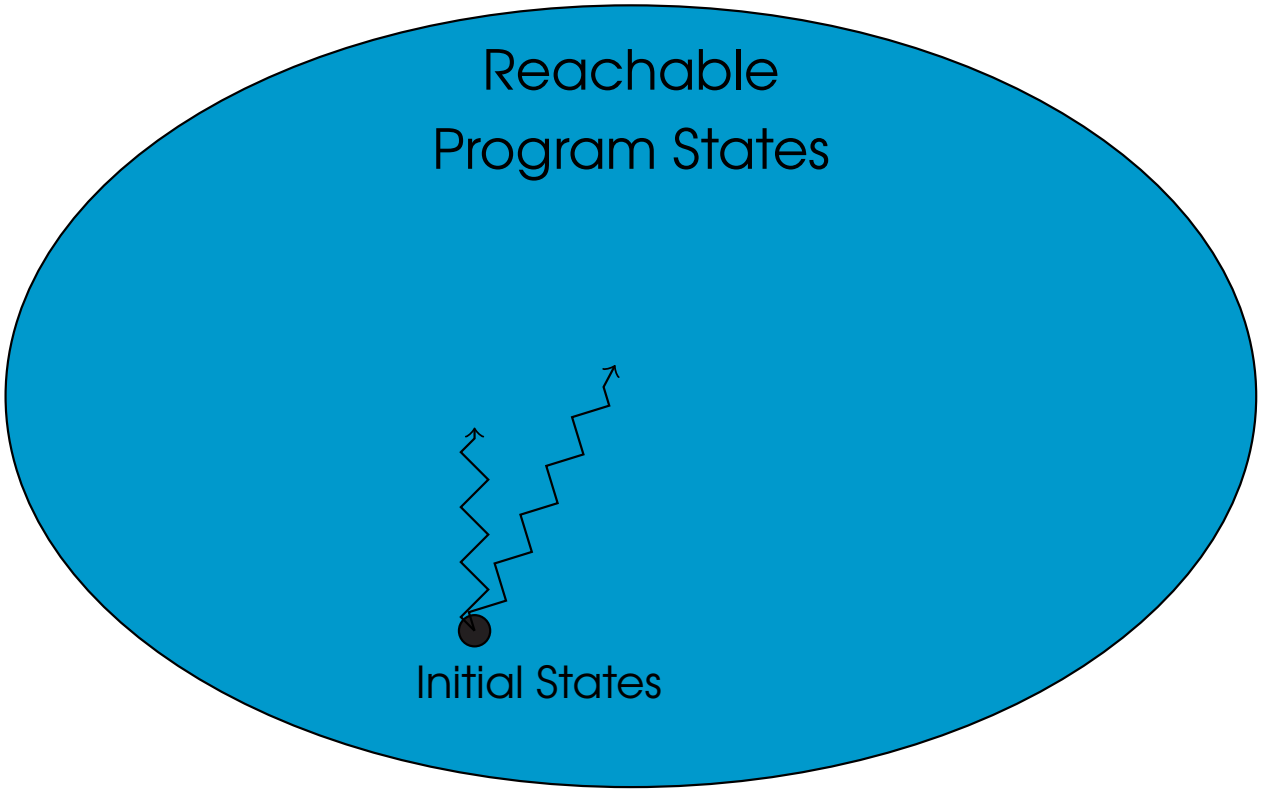
Testing: Intuition



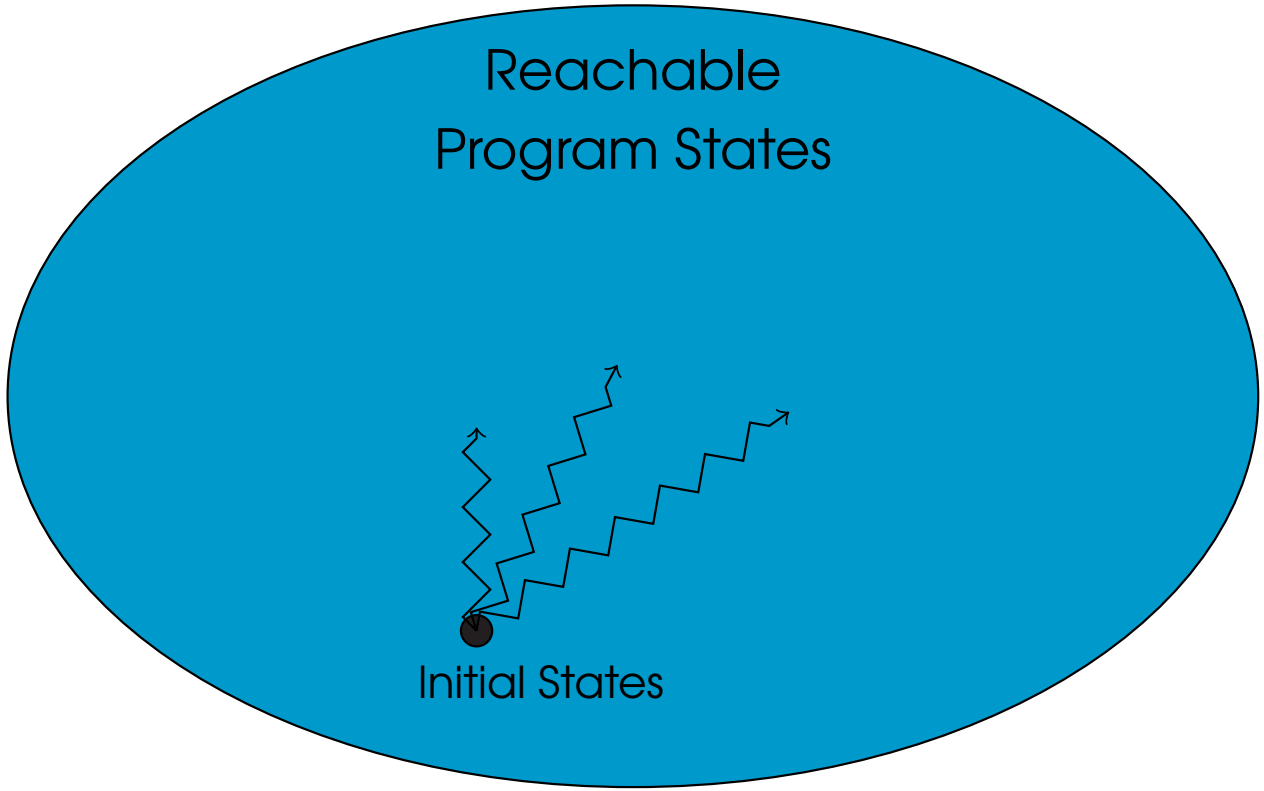
Testing: Intuition



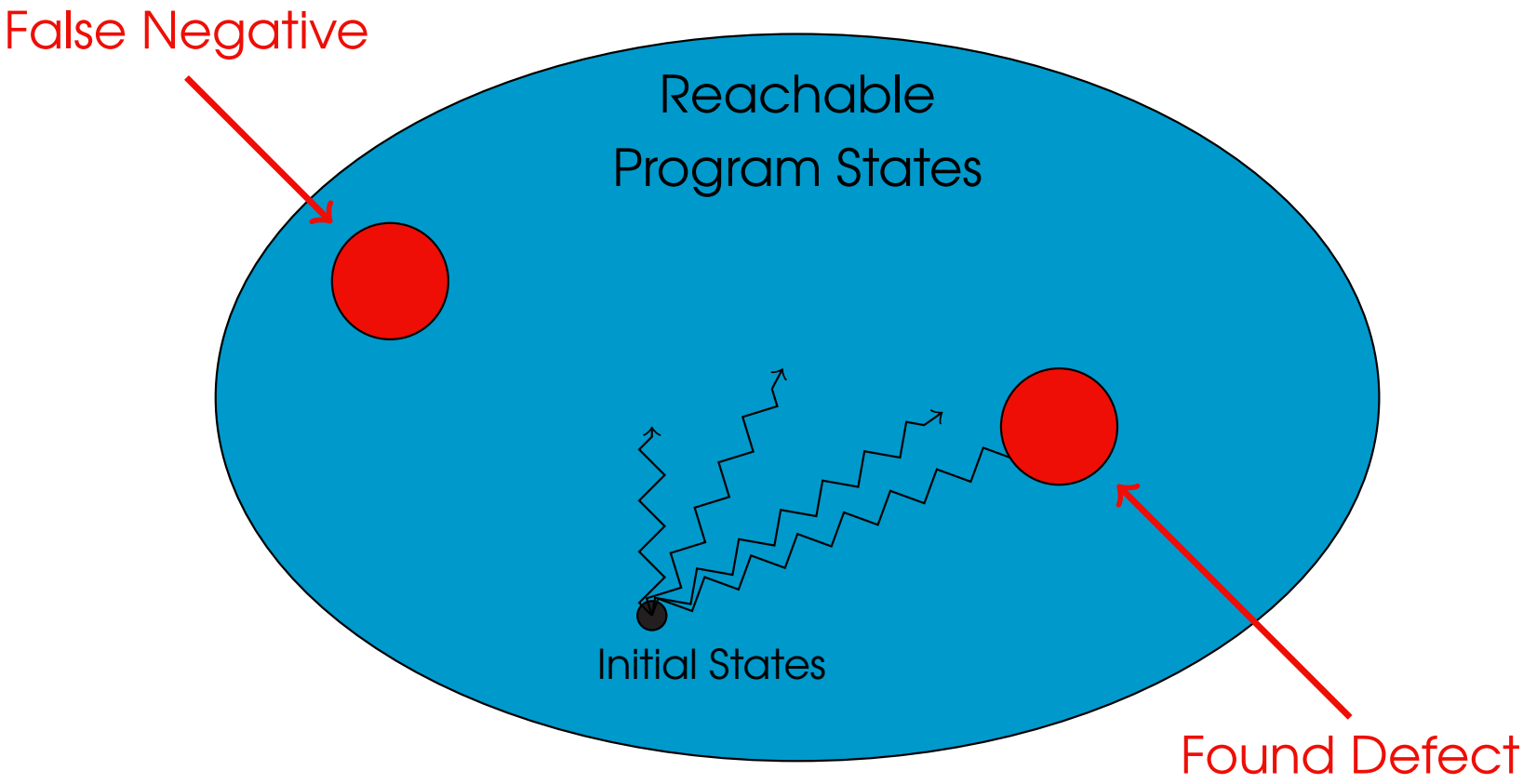
Testing: Intuition



Testing: Intuition



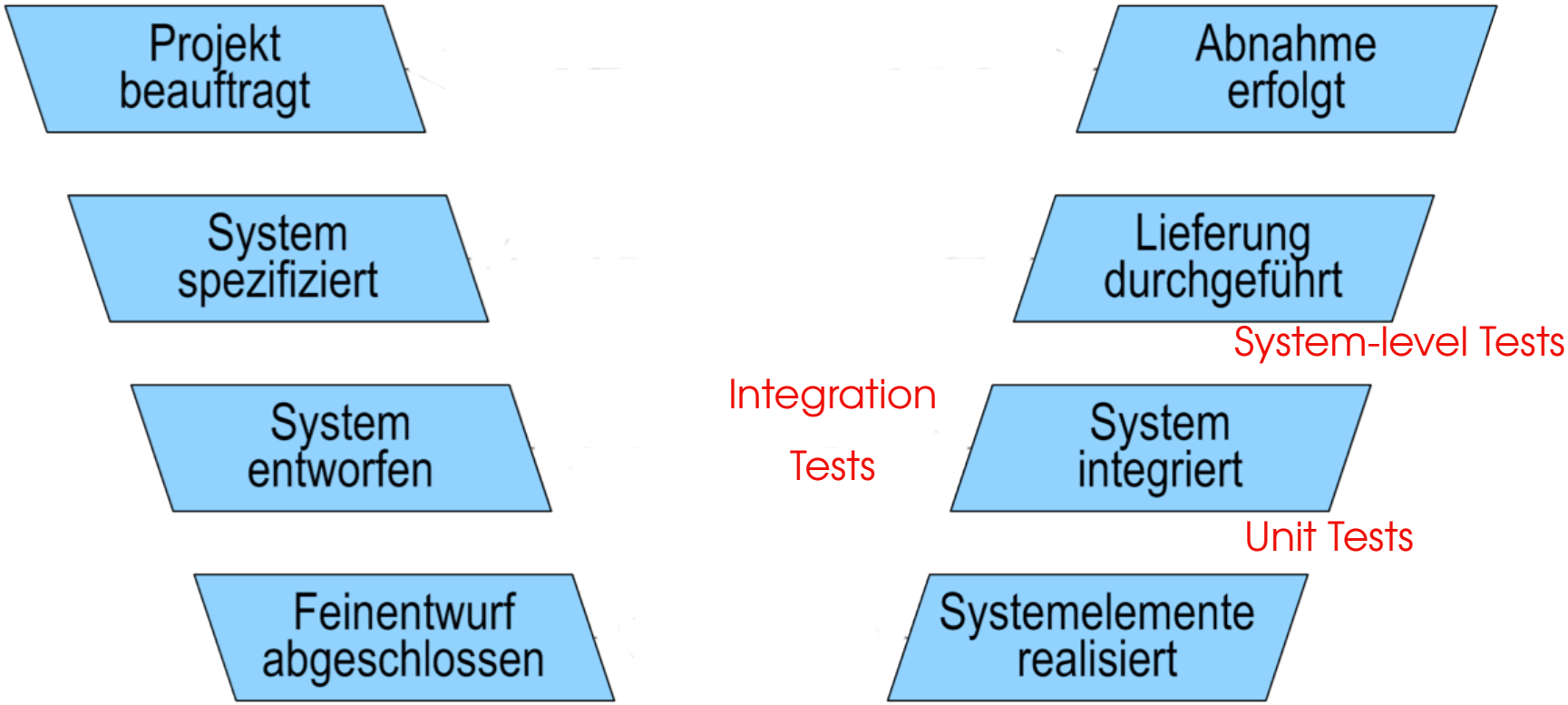
Testing: Intuition



Testing: Properties

- False Positives? No: Tests are always within reachable state space.
- False Negatives? Yes: Exploration not exhaustive.

Testing in the process



Unit Tests

Unit tests test small units of code: individual methods, classes, or modules

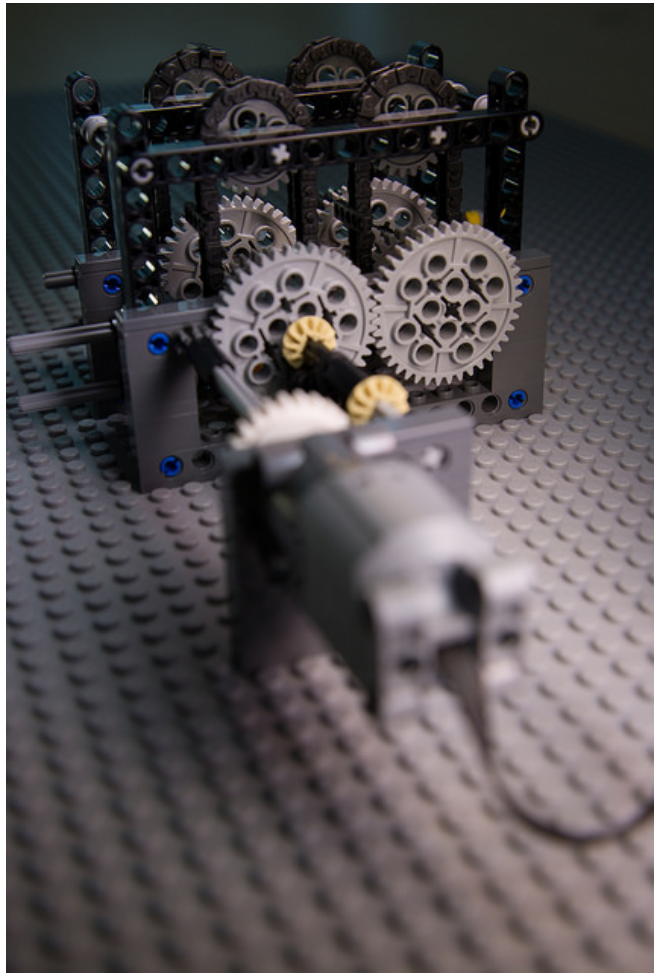
- No external resources (other components, database, etc.)
- External resources are "mocked" (replaced by mock-objects)
- Often written by developer
- Test-driven development: Write unit tests before actual code
- Generation and execution can be automated quite well

Rule of thumb: Code that is not easy to unit test is designed poorly!

Integration Tests

Integration tests check if the parts of a system work together correctly

- Integration of components, or tiers
- Check protocols between components
- Often harder to realize and slower than unit tests (more things needed to run)
- Based on potential realistic interactions
- Hard to automate generation of integration tests (not all interaction scenarios are meaningful)
- Execution can be automated



Unit Testing vs. Integration Testing

https://twitter.com/ThePracticalDev/status/687672086152753152?lang=fil&lang=fil

System Tests

System tests or functional tests target the complete application or system

- From a user's perspective
- Often based on use cases
- High Complexity $\Rightarrow$ hard to write and maintain
- Test Oracle based on user expectations / user experience.
- Real system / production-like environment
- Execution can be automated; oracle may be difficult
- Often designed by dedicated expert / team

Application Example:
Test booking a flight for your web application

Tool Example:
Selenium WebDriver

Summary

- Testing is dynamic analysis (executes actual units, components, and/or systems)
- Test cases: precondition, input, postcondition
- Late in the process
- Individual executions

- How to come up with good tests?
- How to assess the quality of tests?

Test Coverage (SWT)

- Statement Coverage
- Branch Coverage
- Condition Coverage
- …

Essentially: Heuristics …

Code Coverage Example

$[x := a]^1;$

$[y := b]^2;$

if $[x \geq 0]^3$ **then**

  if $[x \geq 10 \wedge y = 0]^4$ **then**

    $[\texttt{skip}]^5;$

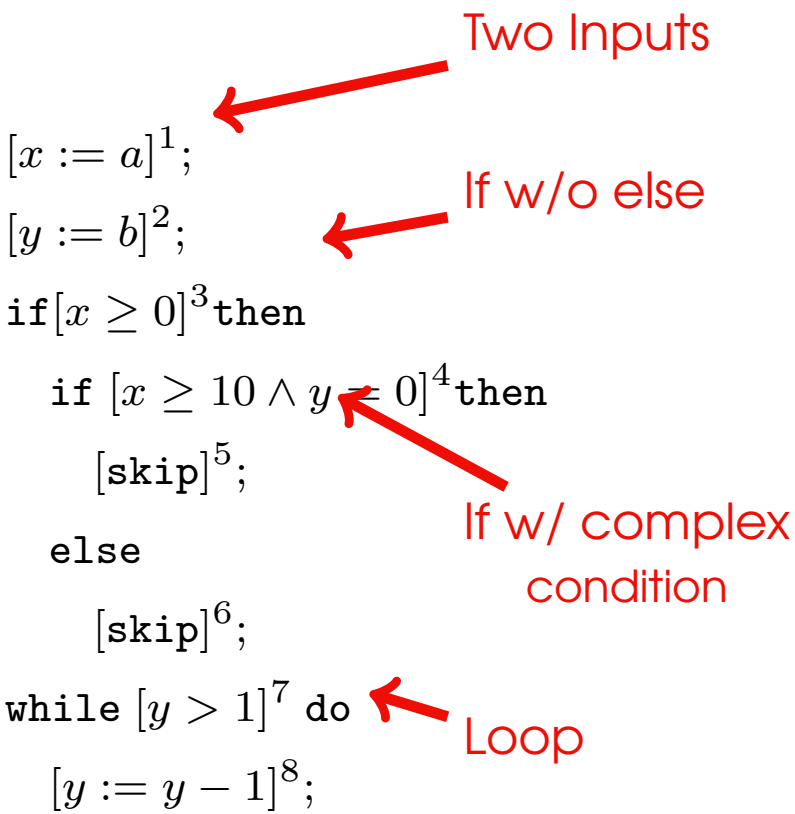
  else

    $[\texttt{skip}]^6;$

while $[y > 1]^7$ **do**

  $[y := y - 1]^8;$

Code Coverage Example



Code Coverage Example

Two Inputs

If w/o else

If w/ complex condition

Loop

$[x := a]^1;$

$[y := b]^2;$

if $[x \geq 0]^3$ **then**

  if $[x \geq 10 \wedge y = 0]^4$ **then**

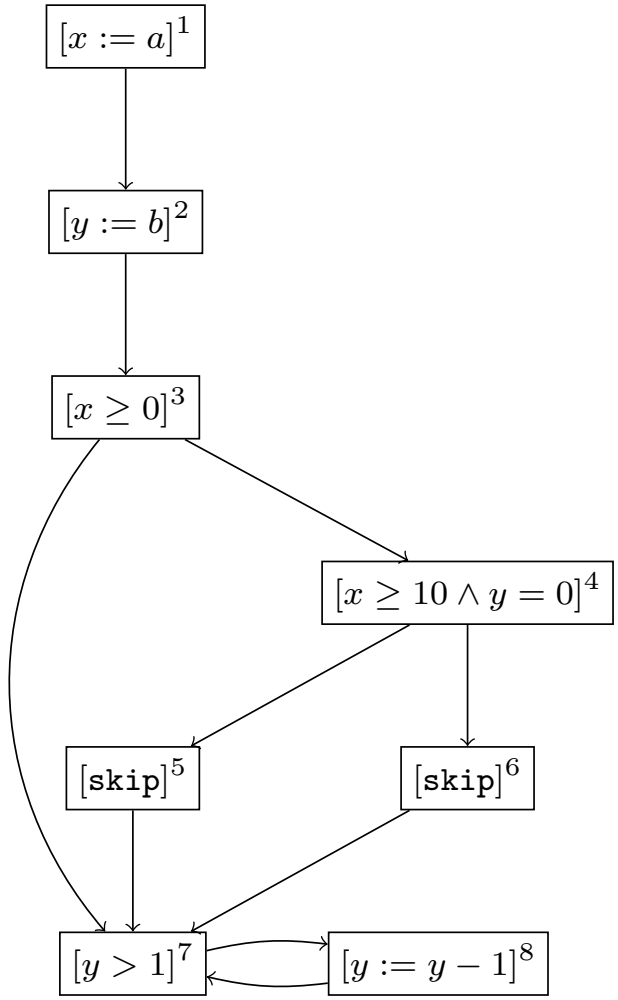
    $[\texttt{skip}]^5;$

  else

    $[\texttt{skip}]^6;$

while $[y > 1]^7$ **do**

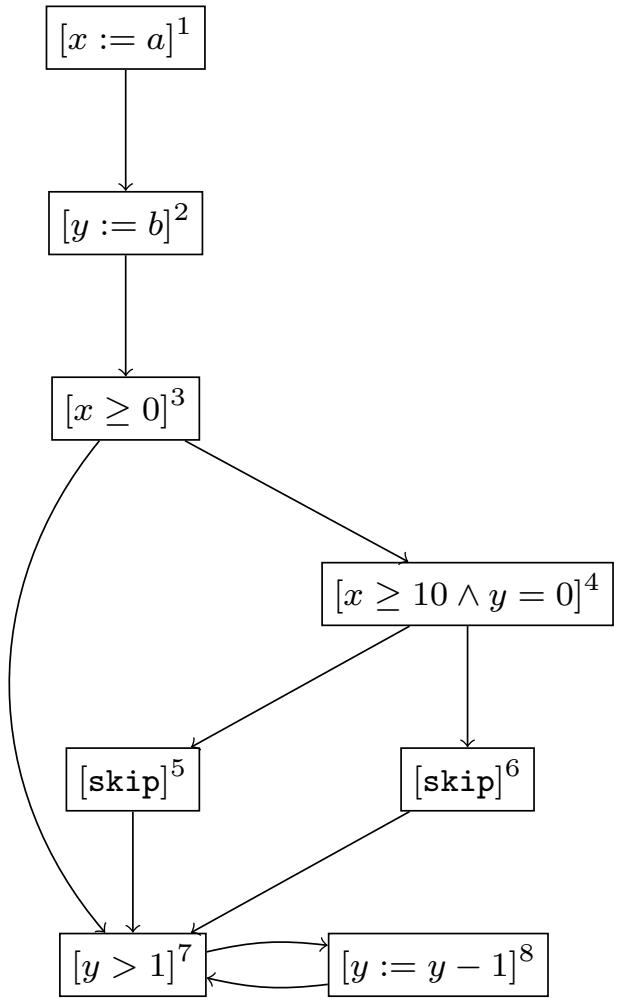
  $[y := y - 1]^8;$



Statement Coverage

Criterion:

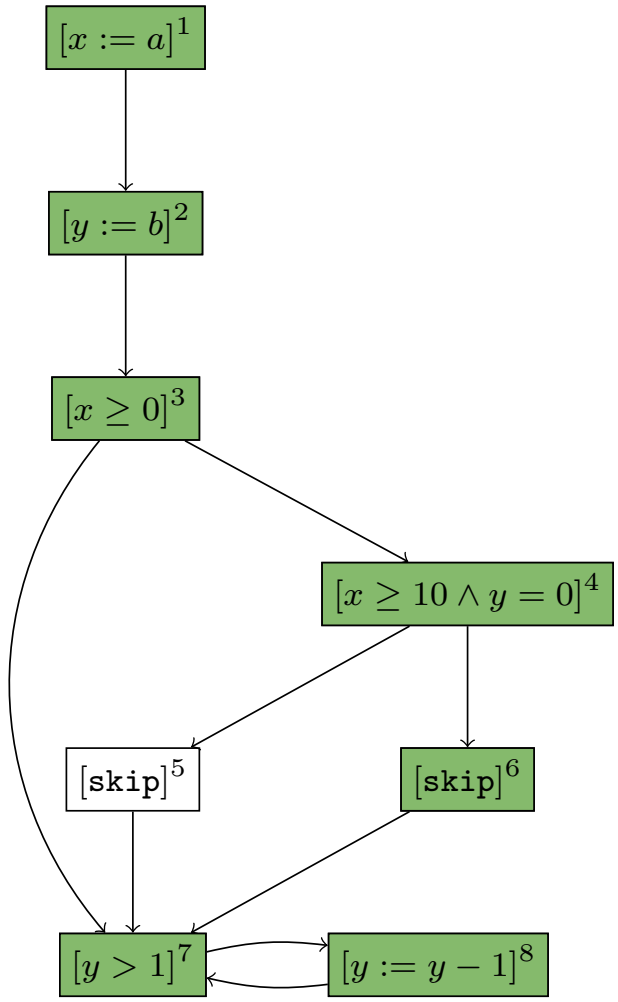
Number of statements covered by a test case or test suite.



Statement Coverage

Criterion:
Number of statements covered by a test case or test suite.

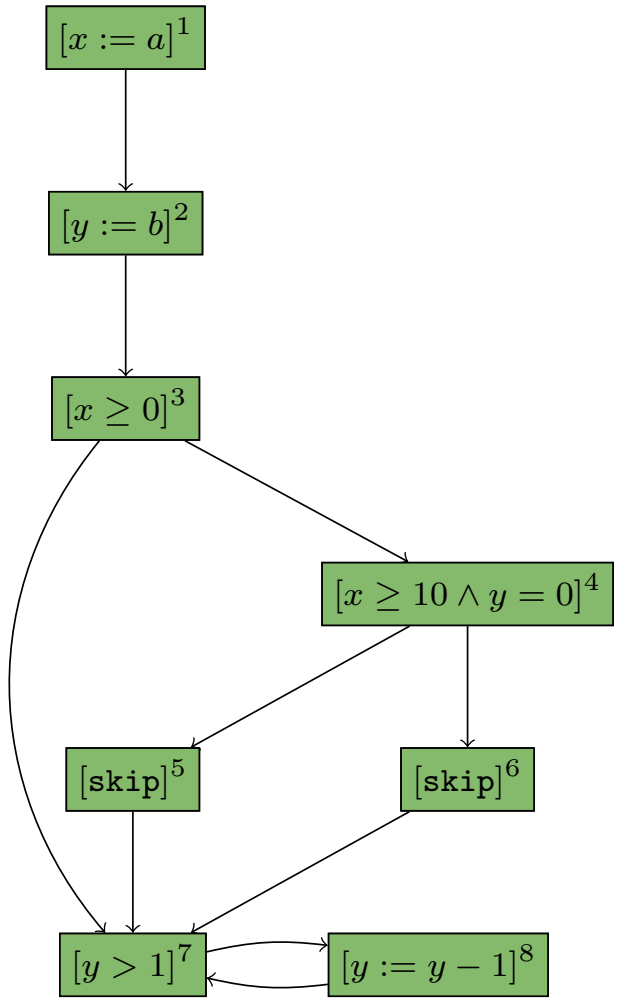
Example:
$(a := 1, b := 2)$



Statement Coverage

Criterion:
Number of statements covered by a test case or test suite.

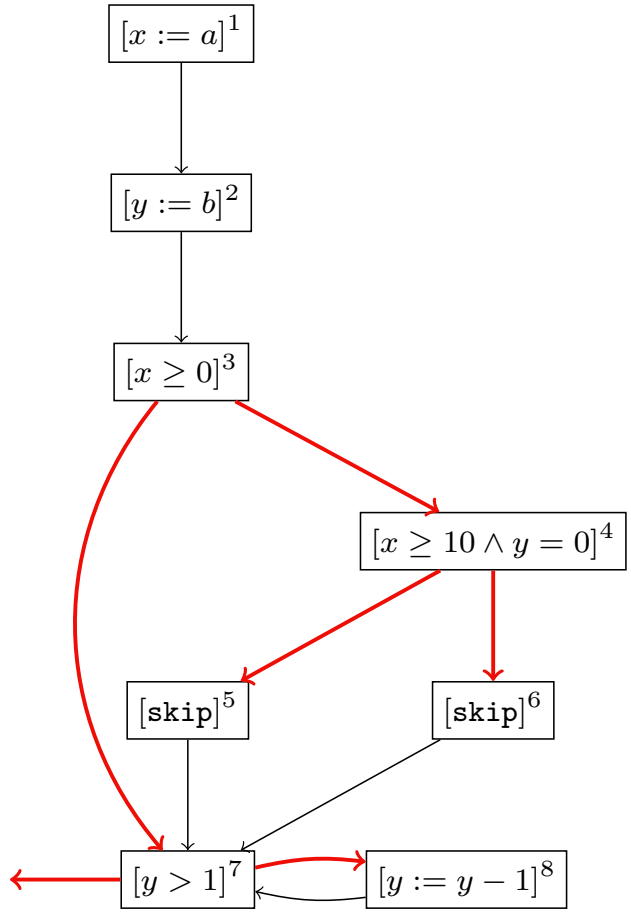
Example:
$(a := 1, b := 2)$
$(a := 10, b := 0)$



Branch Coverage

Criterion:
Number of branches covered by a test case or test suite.

Branch: Conditional exits of if or while statements

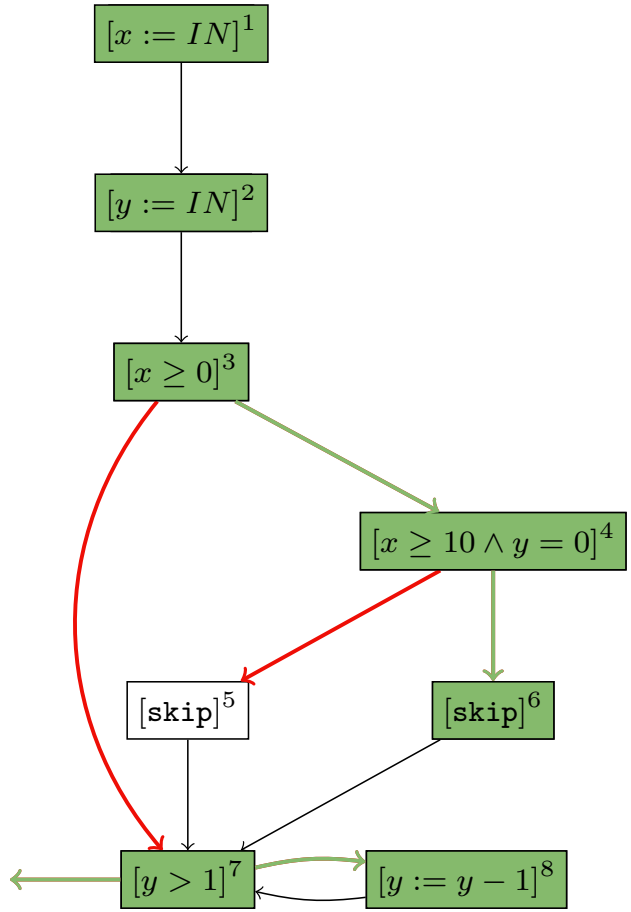


Branch Coverage

Criterion:
Number of branches covered by a test case or test suite.

Branch: Conditional exits of if or while statements

Example: $(a := 1, b := 2)$

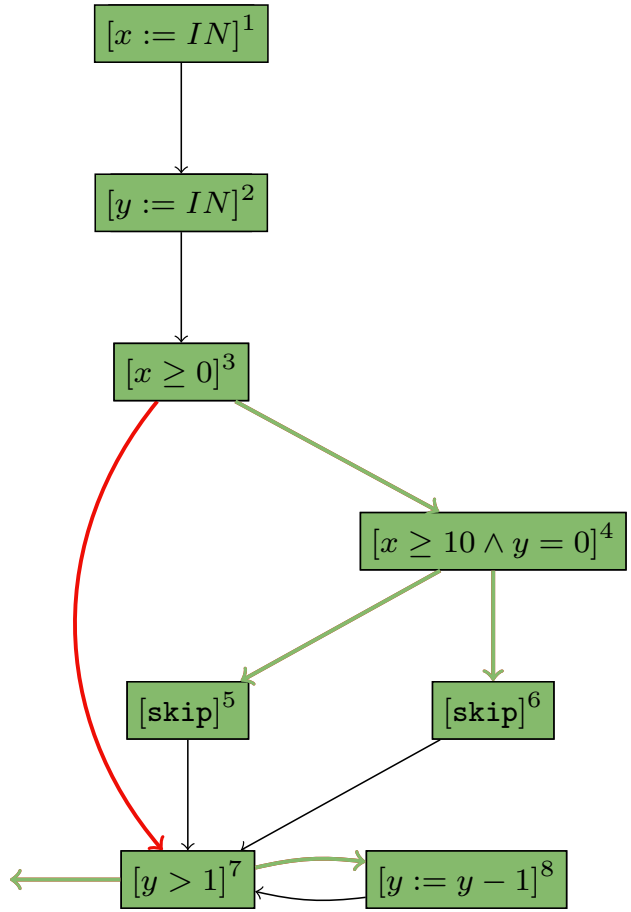


Branch Coverage

Criterion:
Number of branches covered by a test case or test suite.

Branch: Conditional exits of if or while statements

Example: $(a := 1, b := 2)$
$(a := 10, b := 0)$

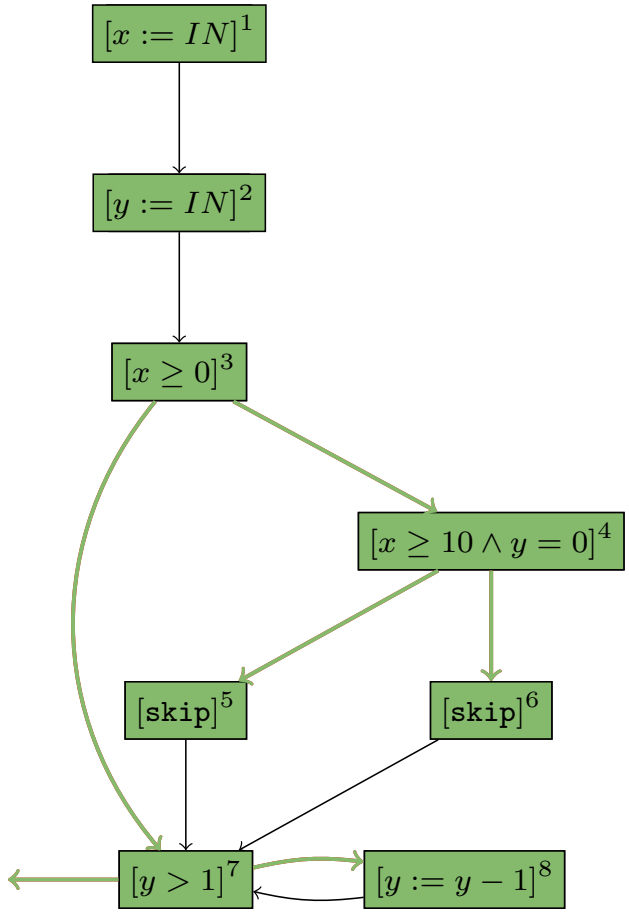


Branch Coverage

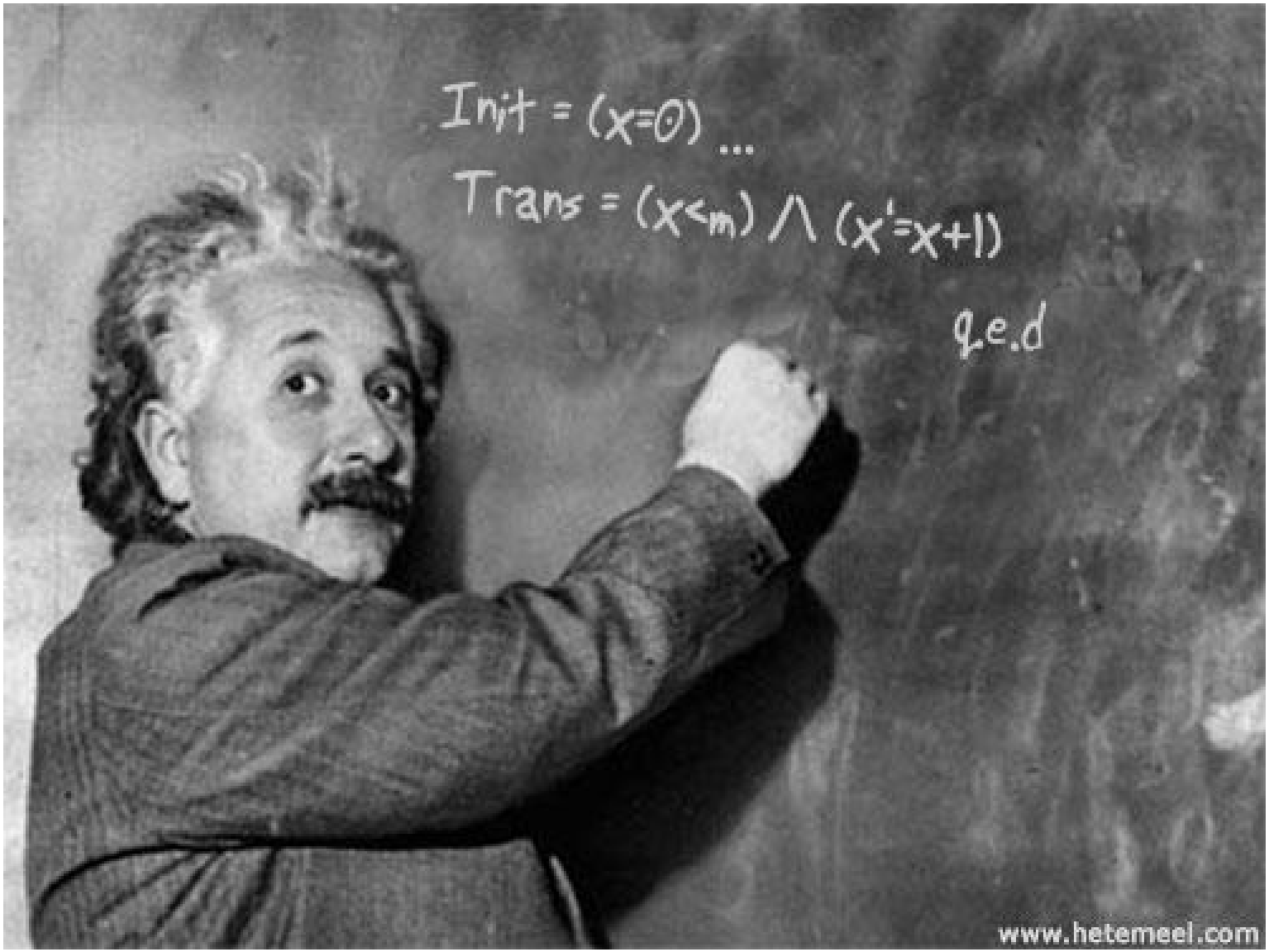
Criterion:
Number of branches covered by a test case or test suite.

Branch: Conditional exits of if or while statements

Example: $(a := 1, b := 2)$
$(a := 10, b := 0)$
$(a := -1, b := 0)$



Symbolic Analysis



White-box Testing

Idea: Generate just enough tests to cover all behaviors of a program.

Symbolic Execution

Symbolic Execution

Symbolic execution is a static program analysis that executes a program with symbolic variables instead of concrete inputs.

- All operations are performed symbolically.
- At branching instructions, both branches are explored and the conditions are tracked as path conditions.

Example:

Concrete: $i = 1$

```
int y = i + 100; // y = 101
if (y > 200) {
  ...
} else {
  ...              // here
}
```

Symbolic: $i = X$

```
int y = i + 100; // y = X+100
if (y > 200) {
  ...      // X+100 > 200
} else {
  ...      // X+100 <= 200
}
```

Symbolic Execution for While (Definitions)

Symbolic Valuations:

A symbolic valuation v_s of a set of variables X is a mapping from X to the set of expressions over X.

Term Replacement:

- We write $e[y/x]$ for the term that results from replacing all occurences of x in e by y.
- We write $v_s[x \mapsto e]$ for the symbolic valuation that is equal to v_s on all $y \in X$ with $y \neq x$ and that maps x to e.

Satisfiability:

- A Boolean expression φ is satisfiable if there is at least one valuation $v \in V_X$ for which $\beta_X[\varphi, v] = 1$.

Automated Proof Assistants

Satisfiability (SAT):

- Prominent NP-complete problem
- DPLL algorithm: worst-case complexity $O(2^n)$
- Often efficient in practice

$\Rightarrow$ Modern SAT solvers really efficient in practice!

Sat. modulo Theories (SMT):

- Extend SAT solver with theories (arithmetic, arrays, functions, ...)
- Different approaches to integration of SAT and theory

$\Rightarrow$ We can use SMT solvers to prove (some) invariants

Satisfiability Modulo Theories. CW Barrett, R Sebastiani, SA Seshia, C Tinelli, Handbook of satisfiability, 2009.

SMT Solvers

Let $\varphi[\mathbf{x}]$ be a formula in some logic fragment (e.g., QFLIA - quantifier free linear integer arithmetic)

SMT Solver:
Decides if $\varphi[\mathbf{x}]$ is satisfiable (SAT) or not satisfiable (UNSAT).

E.g., Z3: `http://rise4fun.com/z3`

Finding valuations with a constraint solver

Demo

Symbolic Execution for While

- We will define symbolic execution on the control flow graph
- We slightly extend the earlier definition of CFGs and label outgoing edges of decisions

- Program variables: x, y, z
- Symbolic values: X, Y, Z

- Idea: unrolling control flow graph, simulating effect of execution through
  - term replacement
  - recording of path constraints
  - satisfiability checking

Symbolic Execution for While CFG

sym_exec($i,\ pc,\ v_s$)

Require: Node i, path constraint pc, sym. val. v_S

1: **if** pc not satisfiable **then**
2: output pc as $UNSAT$
3: **return**
4: **end if**
5: **if** node i is decision b **then**
6: $pc_b \leftarrow b[v_s(x)/x]$ for $x \in X$
7: Let $j,\ k$ be then/do- and else-successor of i
8: sym_exec($j,\ (pc\ \wedge\ pc_b),\ v_s$)
9: sym_exec($k,\ (pc\ \wedge\ \neg pc_b),\ v_s$)
10: **else**
11: **if** node i is assigment $y := e$ **then**
12: $e_s \leftarrow e[v_s(x)/x]$ for $x \in X$
13: $v'_s \leftarrow v_s[y \mapsto e_s]$
14: **end if**
15: **if** node i has successor j **then**
16: sym_exec($j,\ pc,\ v'_s$)
17: **else**
18: output pc as SAT and model for pc
19: **end if**
20: **end if**

Symbolic Execution for While CFG

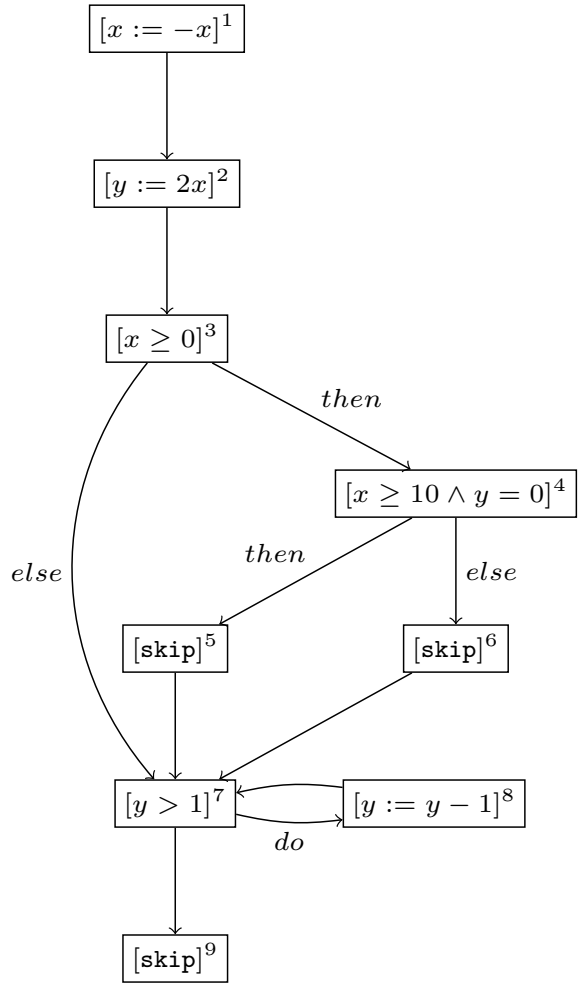
sym_exec($i,\ pc,\ v_s$)

Require: Node i, path constraint pc, sym. val. v_S

1: **if** pc not satisfiable **then**
2: output pc as $UNSAT$
3: **return**
4: **end if**
5: **if** node i is decision b **then**
6: $pc_b \leftarrow b[v_s(x)/x]$ for $x \in X$
7: Let $j,\ k$ be then/do- and else-successor of i
8: sym_exec($j,\ (pc\ \wedge\ pc_b),\ v_s$)
9: sym_exec($k,\ (pc\ \wedge\ \neg pc_b),\ v_s$)
10: **else**
11: **if** node i is assigment $y := e$ **then**
12: $e_s \leftarrow e[v_s(x)/x]$ for $x \in X$
13: $v'_s \leftarrow v_s[y \mapsto e_s]$
14: **end if**
15: **if** node i has successor j **then**
16: sym_exec($j,\ pc,\ v'_s$)
17: **else**
18: output pc as SAT and model for pc
19: **end if**
20: **end if**

Example:

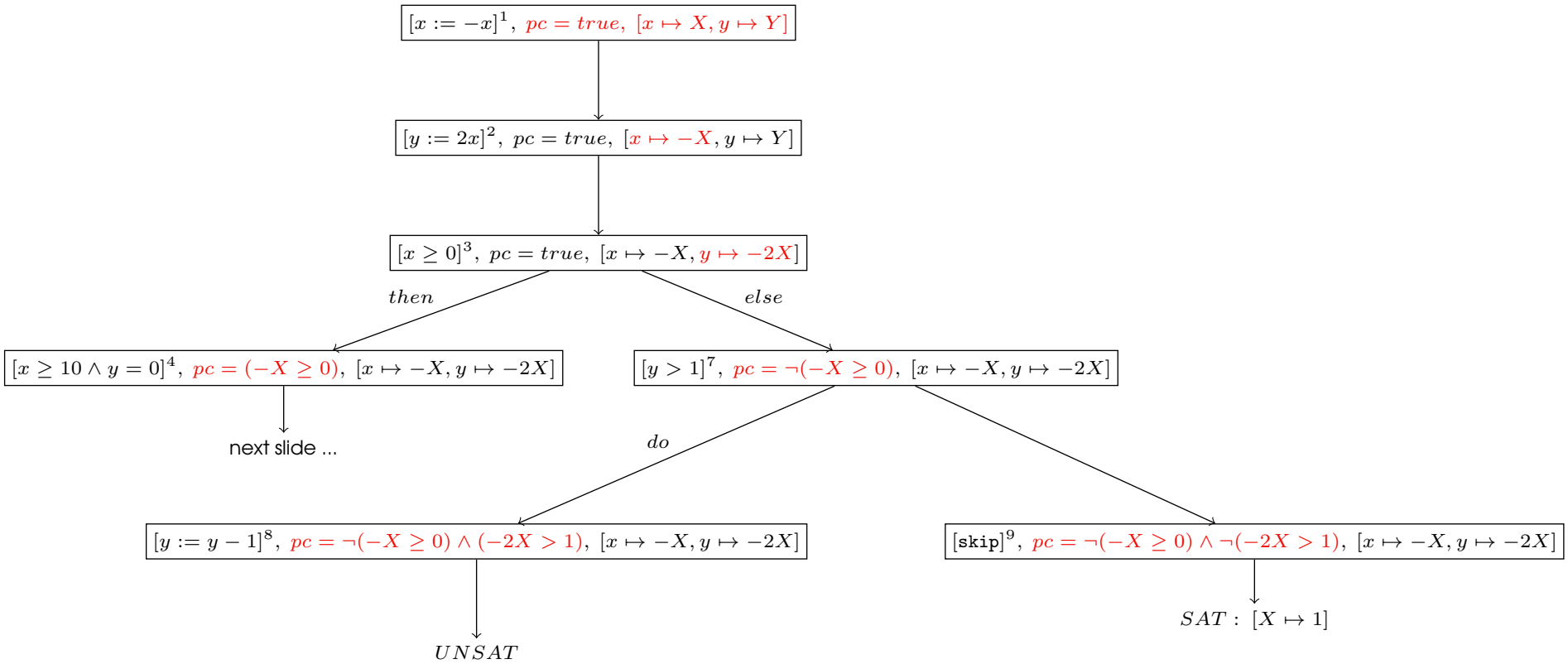
sym_exec($1,\ true,\ [x \mapsto X, y \mapsto Y]$)



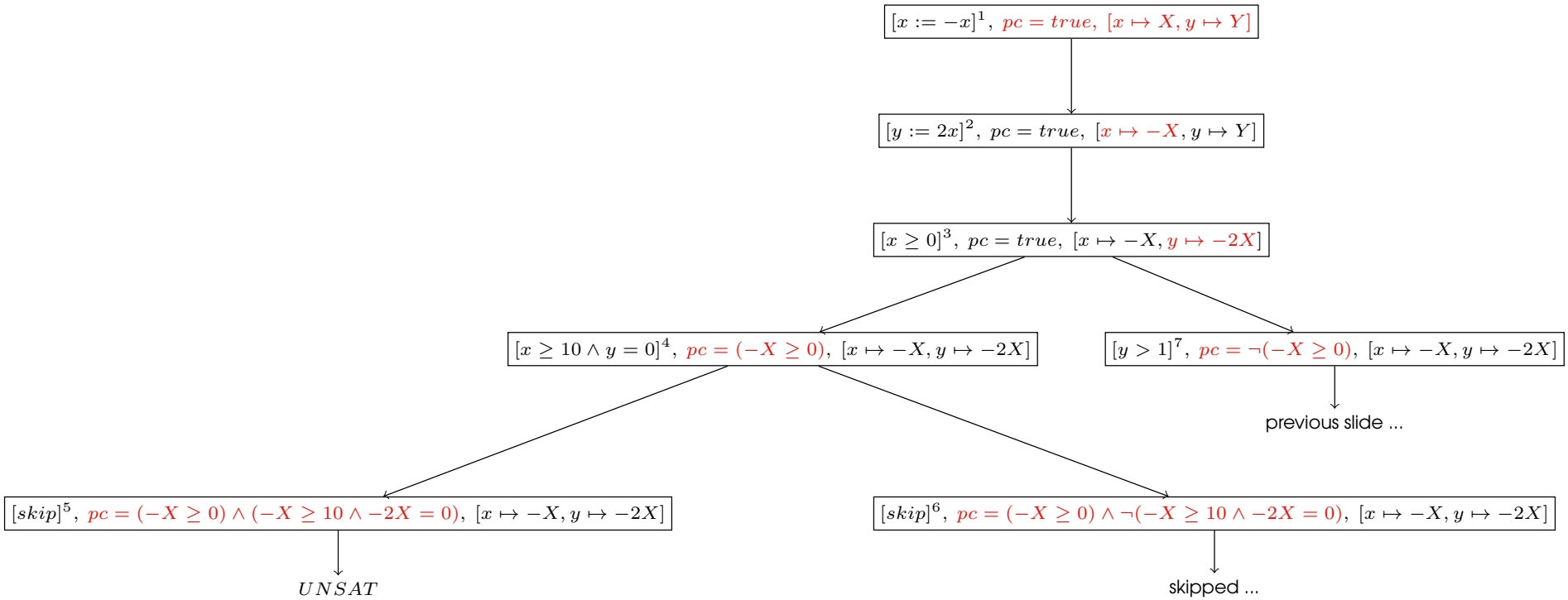
Symbolic Execution for While (Discussion)

- Semi algorithm (unrolling of loops may be infinite)

Symbolic Decision Tree for While CFG (Example)



Symbolic Decision Tree for While CFG (Example)



Variant: Dynamic Symbolic Execution

Variant of symbolic execution

- Execution with concrete values
- Recording of path constraints
- Negation of path constraints for finding new concrete values

$\Rightarrow$ Only one call to SAT solver per path

Dynamic Symbolic Execution

```
private int cruiseMph = 0;

public void set(int speed) {

  if (speed < 30) {
    warning("...");
  }

  cruiseMph = speed;

  assert (cruiseMph >= 20);
}

public static void main(String[] a)
{
  @Symbolic int X = 0;
  set(X);
}
```

Symbolic method parameters

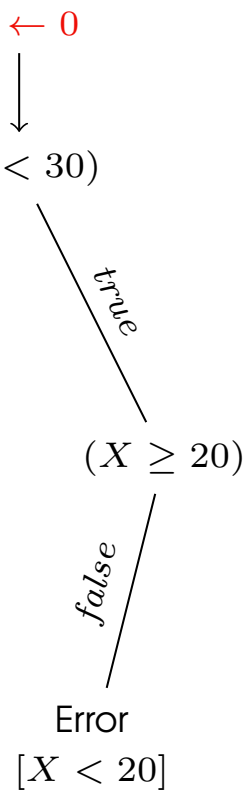
Dynamic Symbolic Execution

```
private int cruiseMph = 0;

public void set(int speed) {

  if (speed < 30) {
    warning("...");
  }

  cruiseMph = speed;

  assert (cruiseMph >= 20);
}

public static void main(String[] a)
{
  @Symbolic int X = 0;
  set(X);
}
```

Execute with concrete values

Record symbolic constraints

Symbolic method parameters

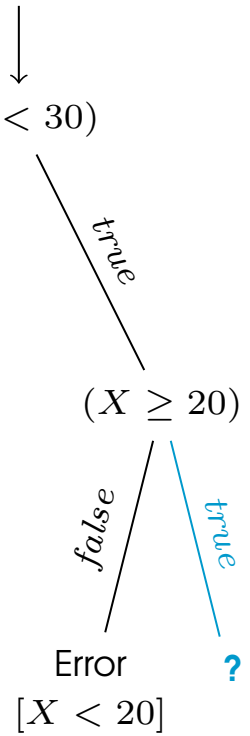


Dynamic Symbolic Execution

```
private int cruiseMph = 0;

public void set(int speed) {

  if (speed < 30) {
    warning("...");
  }

  cruiseMph = speed;

  assert (cruiseMph >= 20);
}

public static void main(String[] a)
{
  @Symbolic int X = 0;
  set(X);
}
```

Execute with concrete values

Record symbolic constraints



Find X for: $(X < 30) \wedge (X \geq 20)$

Symbolic method parameters

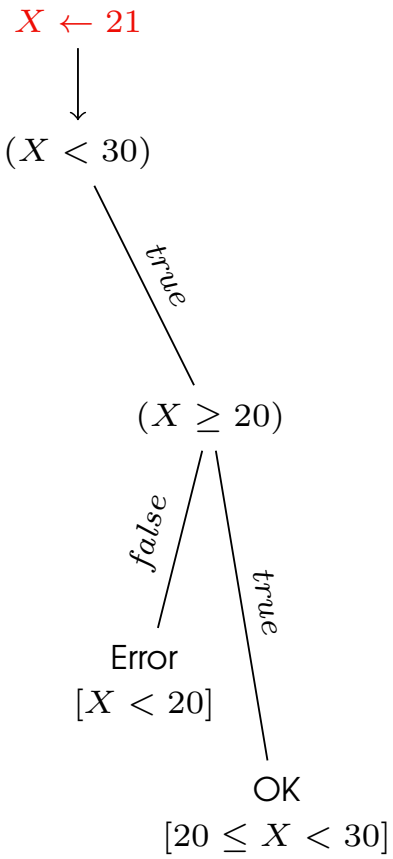
Dynamic Symbolic Execution

```
private int cruiseMph = 0;

public void set(int speed) {

  if (speed < 30) {
    warning("...");
  }

  cruiseMph = speed;

  assert (cruiseMph >= 20);
}

public static void main(String[] a)
{
  @Symbolic int X = 0;
  set(X);
}
```

Execute with concrete values

Record symbolic constraints

Symbolic method parameters



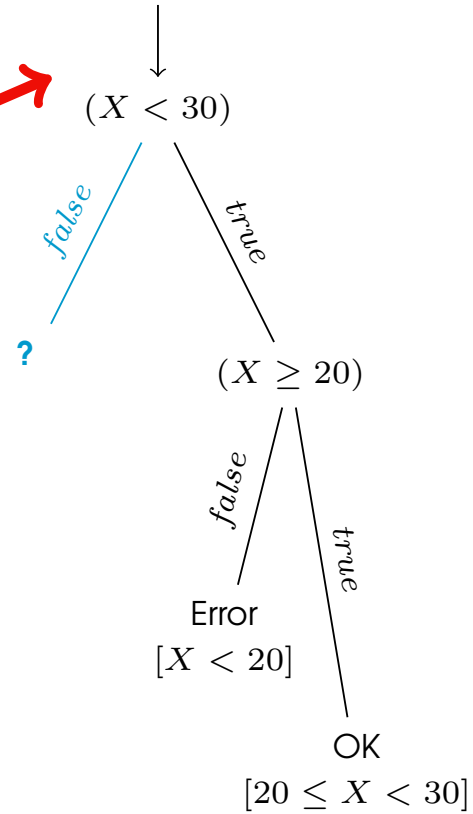
Dynamic Symbolic Execution

```
private int cruiseMph = 0;

public void set(int speed) {

  if (speed < 30) {
    warning("...");
  }

  cruiseMph = speed;

  assert (cruiseMph >= 20);
}

public static void main(String[] a)
{
  @Symbolic int X = 0;
  set(X);
}
```

Execute with concrete values

Record symbolic constraints

Symbolic method parameters



Find X for: $\neg(X < 30)$

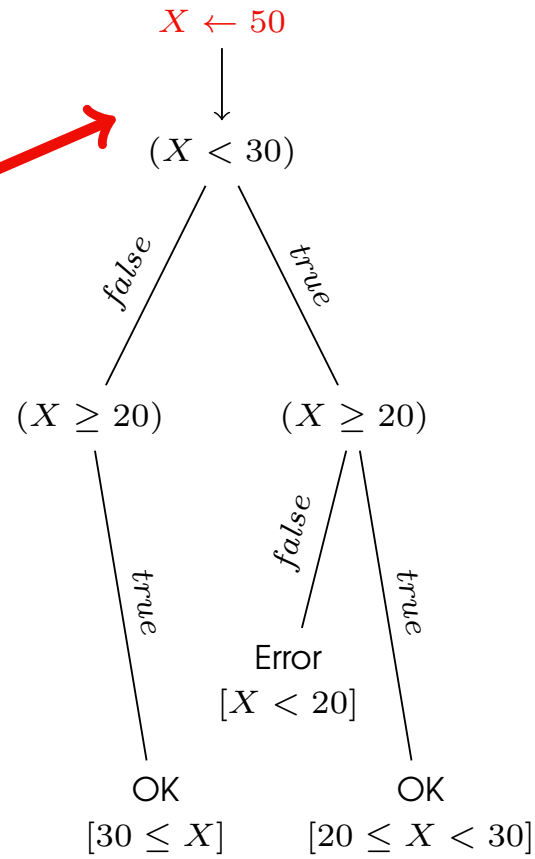
Dynamic Symbolic Execution

```
private int cruiseMph = 0;

public void set(int speed) {

  if (speed < 30) {
    warning("...");
  }

  cruiseMph = speed;

  assert (cruiseMph >= 20);
}

public static void main(String[] a)
{
  @Symbolic int X = 0;
  set(X);
}
```

Execute with concrete values

Record symbolic constraints

Symbolic method parameters

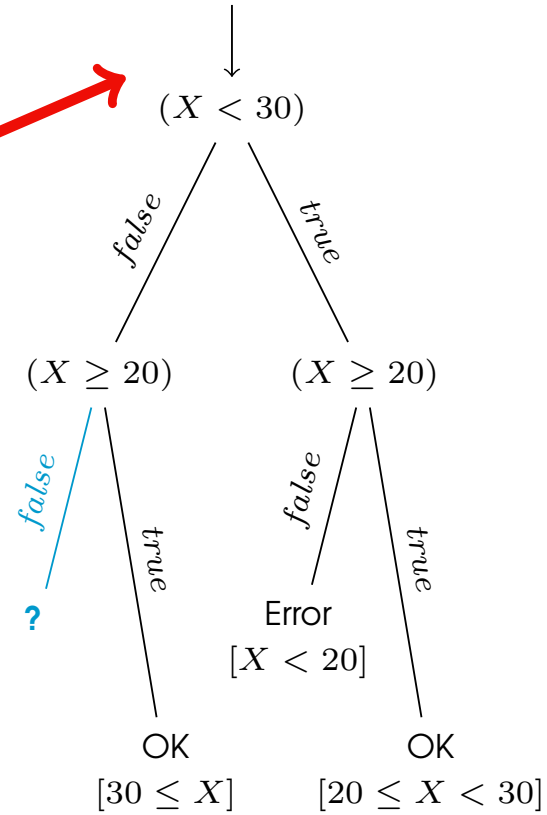


Dynamic Symbolic Execution

```
private int cruiseMph = 0;

public void set(int speed) {

  if (speed < 30) {
    warning("...");
  }

  cruiseMph = speed;

  assert (cruiseMph >= 20);
}

public static void main(String[] a)
{
  @Symbolic int X = 0;
  set(X);
}
```

Execute with concrete values

Record symbolic constraints



Find X for: $\neg(X < 30) \wedge \neg(x \geq 20)$

Symbolic method parameters

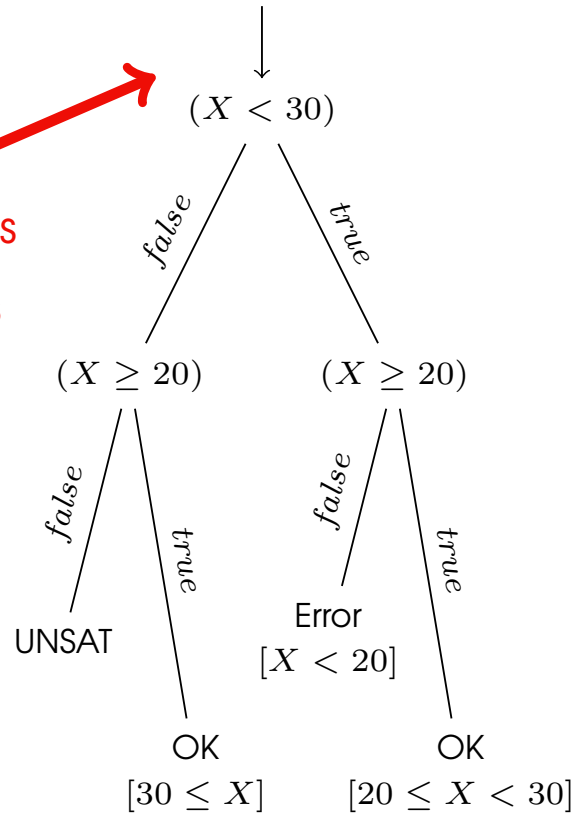
Dynamic Symbolic Execution

```
private int cruiseMph = 0;

public void set(int speed) {

  if (speed < 30) {
    warning("...");
  }

  cruiseMph = speed;

  assert (cruiseMph >= 20);
}

public static void main(String[] a)
{
  @Symbolic int X = 0;
  set(X);
}
```

Execute with concrete values

Record symbolic constraints

Symbolic method parameters

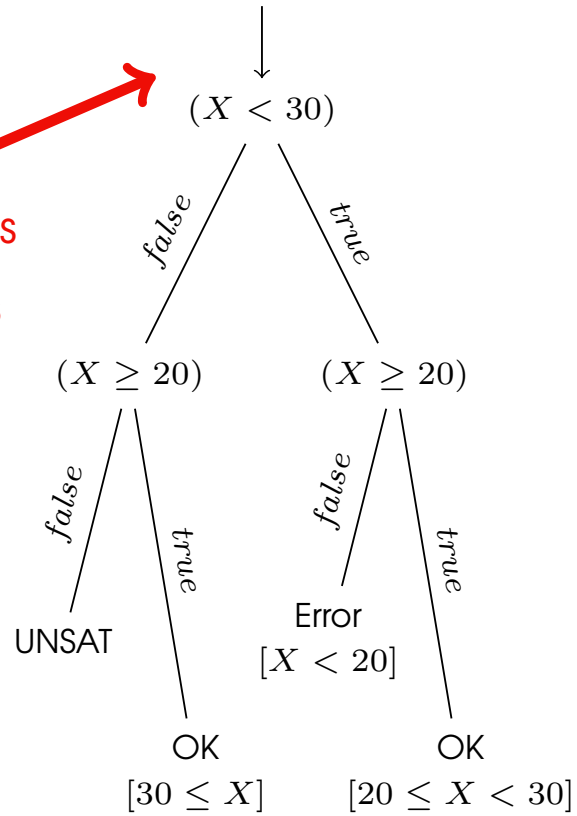


Dynamic Symbolic Execution

```
private int cruiseMph = 0;

public void set(int speed) {

  if (speed < 30) {
    warning("...");
  }

  cruiseMph = speed;

  assert (cruiseMph >= 20);
}

public static void main(String[] a)
{
  @Symbolic int X = 0;
  set(X);
}
```

Execute with concrete values

Record symbolic constraints

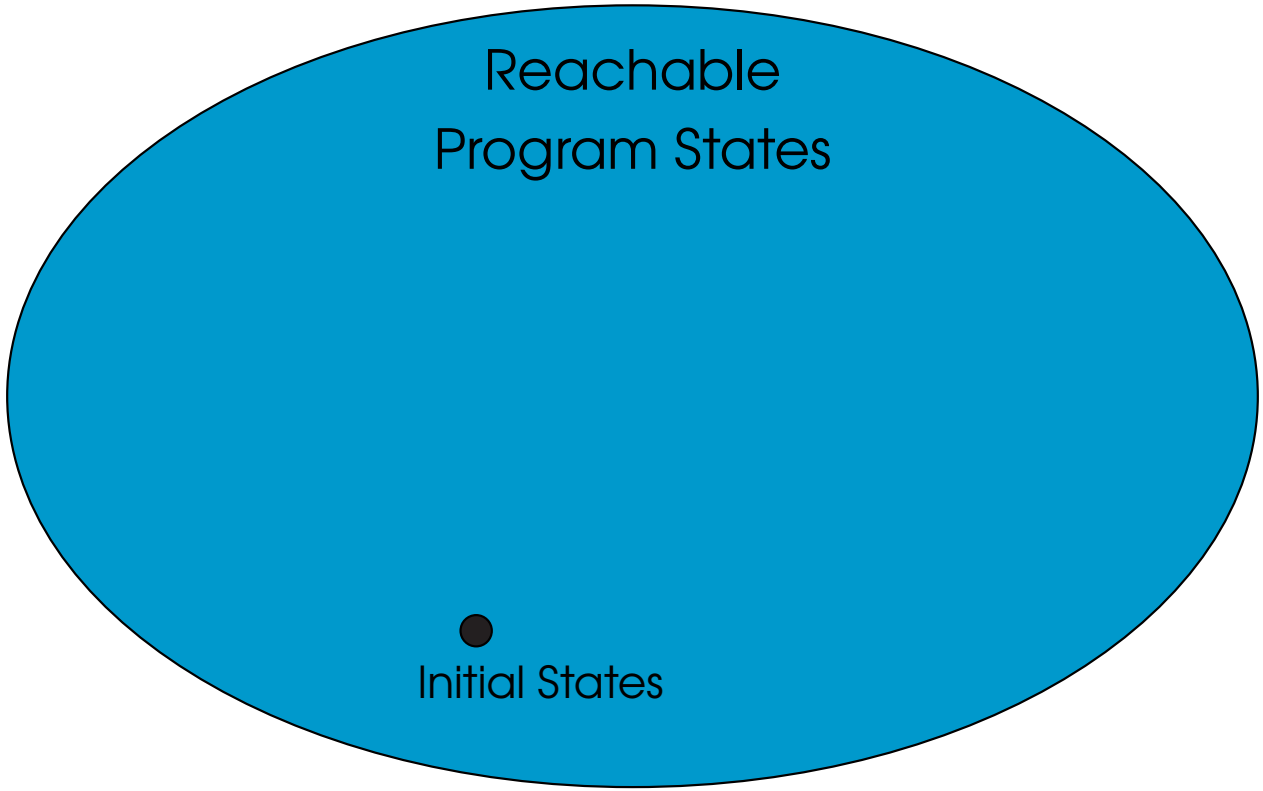


Symbolic method parameters

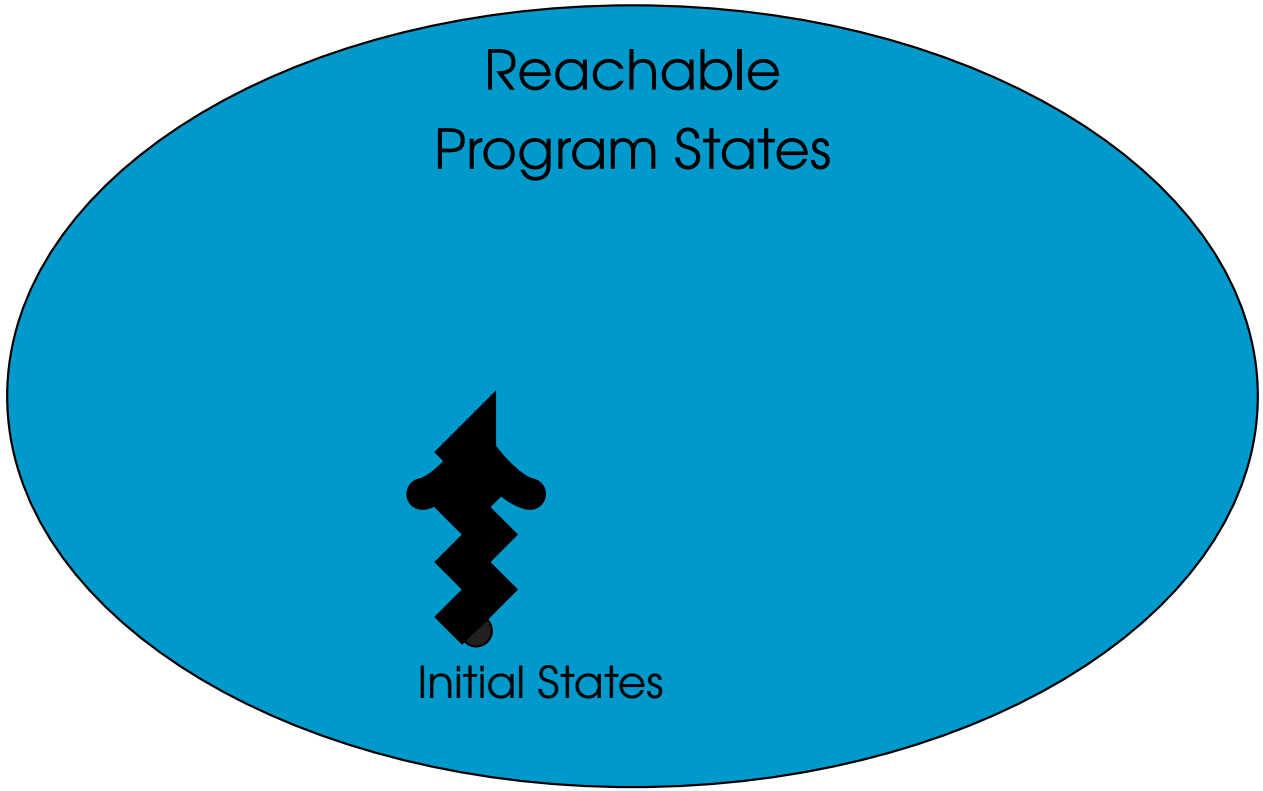
E.g. Luckow et al., TACAS 2016

https://github.com/psycopaths/jdart

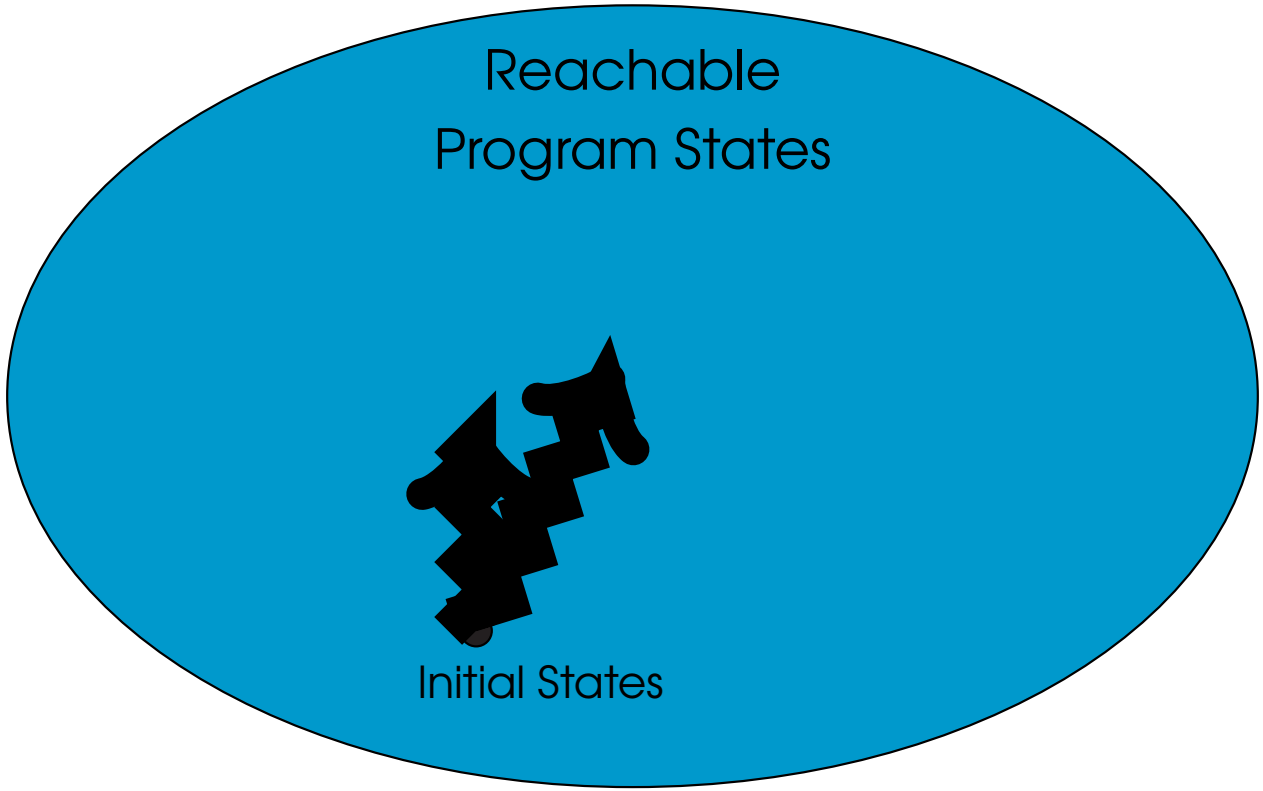
Symbolic Execution: Intuition



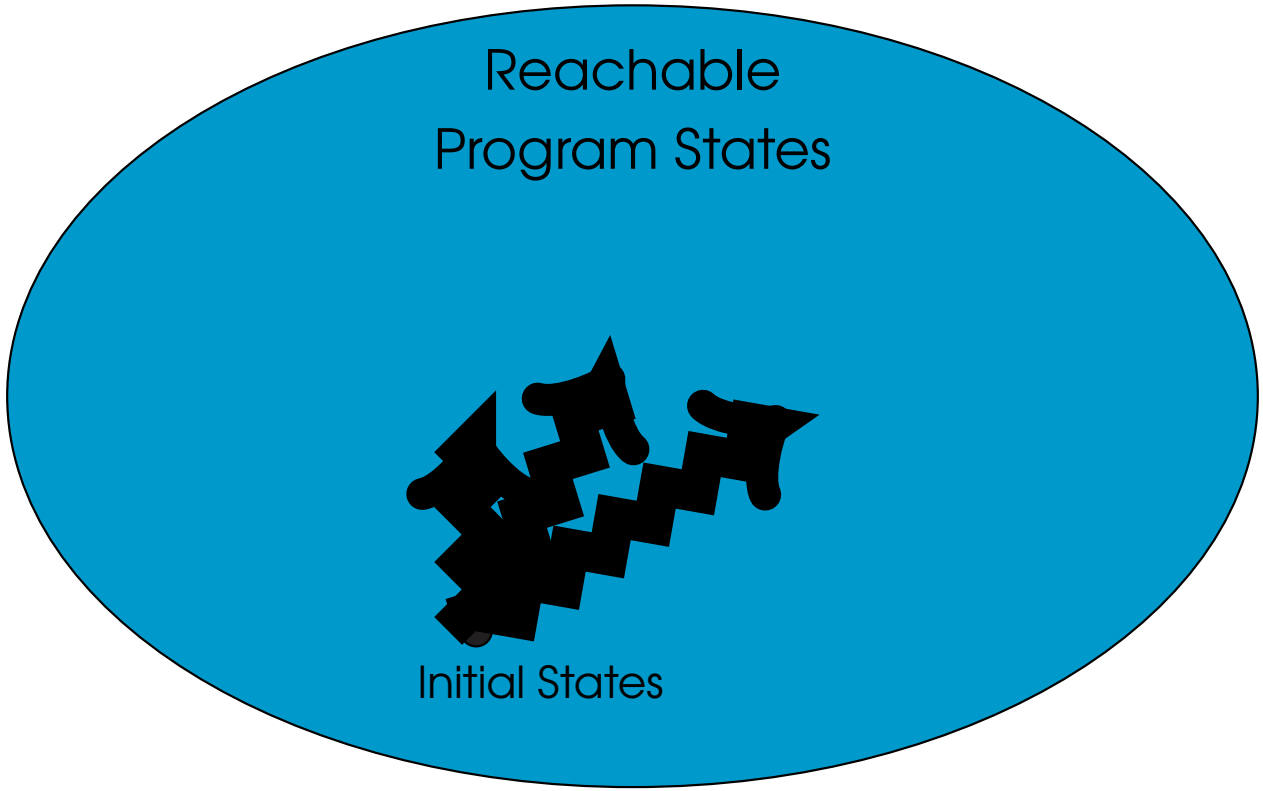
Symbolic Execution: Intuition



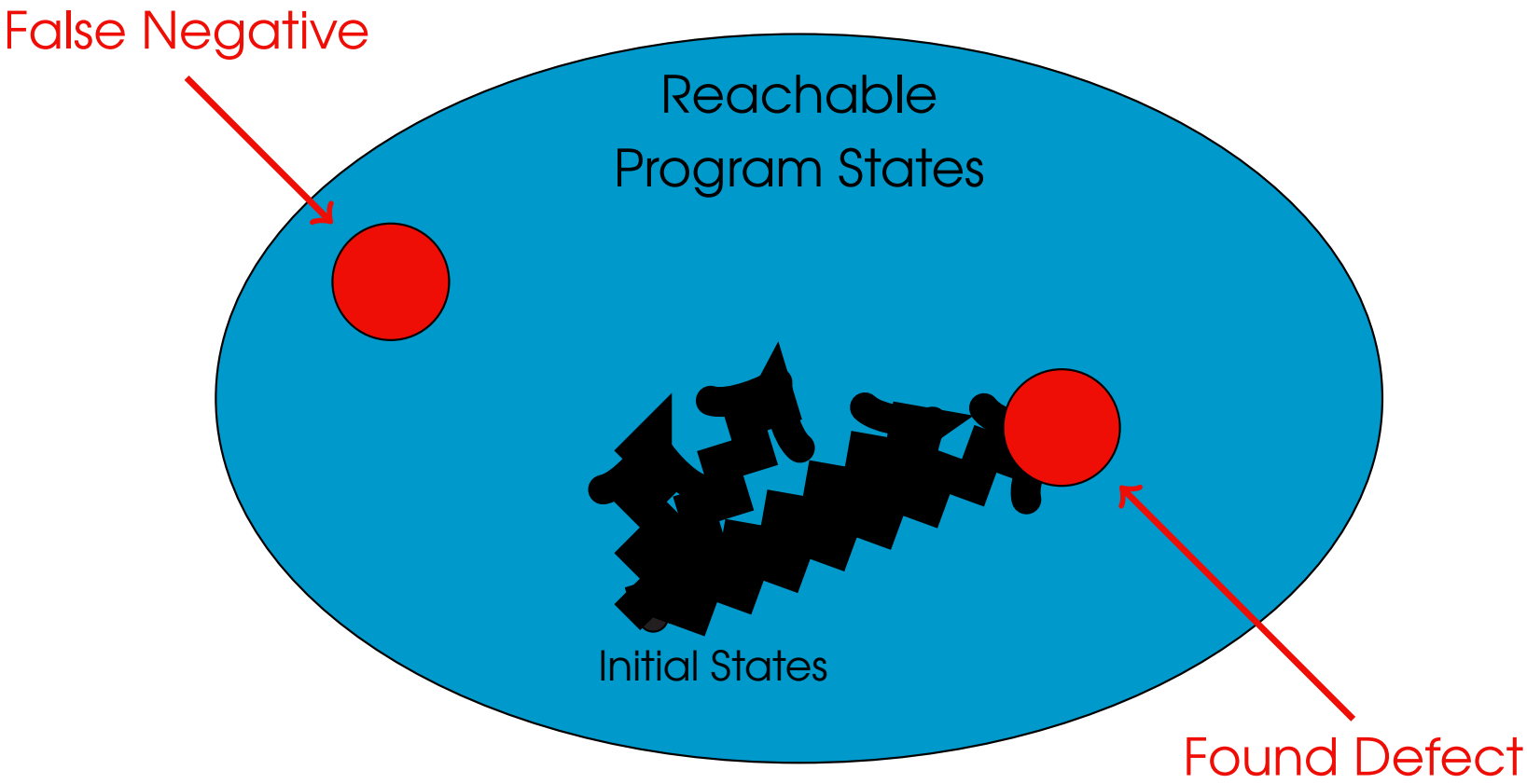
Symbolic Execution: Intuition



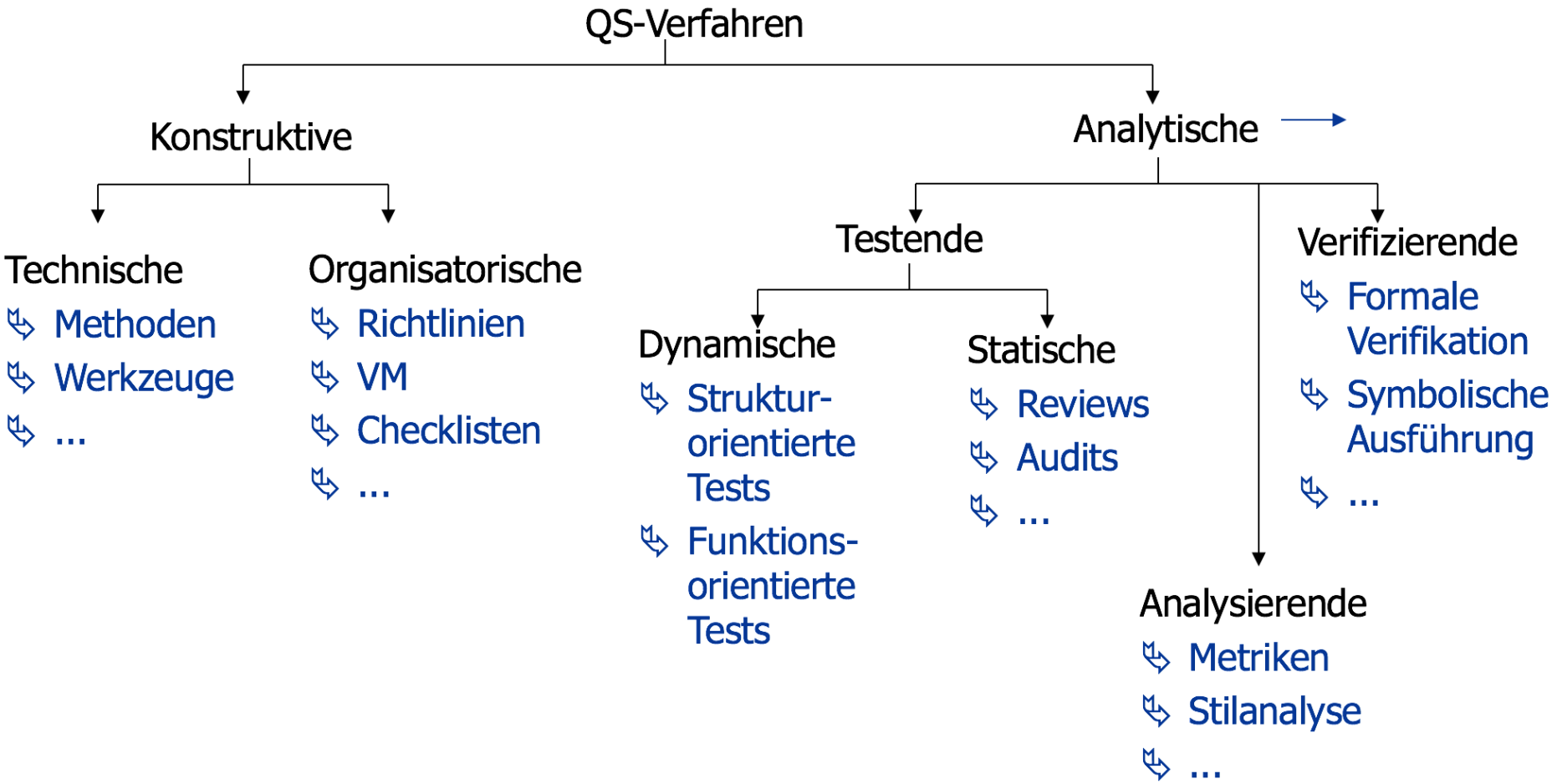
Symbolic Execution: Intuition



Symbolic Execution: Intuition



Zusammenfassung



P. Liggesmeyer 2002

Zusammenfassung

- Formale Definition von Semantik (SOS)
- Analyse von Verhalten
- False positives vs. false negatives

Beispiele für Analysen:

- Testen: einzelne konkrete Ausführungen
- Model Checking: Suche im Zustandsraum
- Symbolic Execution: Symbolische Analyse aller Ausführungspfade

Weitere Ansätze:

- Hoare Beweise
- Symbolische Suche
- Induktion