```rust
#[test]
fn some_test() {
    let x: i32 = 2;
    let y: i32 = 1;
    let mut point: Point = Point::new(x, y);
    do_something(&mut point);
}
```

Generierte Testsuite

```rust
#[test]
fn some_test() {
    let x: i32 = 2;
    let y: i32 = 1;
    let mut point: Point = Point::new(x, y);
    do_something(&mut point);
}
```

```rust
fn instrument_branch(&self, branch: &mut Block) {
    let stmts : &mut Vec<Stmt> = &mut branch.stmts;
    let id : i32 = self.branch_id;
    let stmt: Stmt = parse_quote! {
        Monitor::trace_branch(#id);
    };
    stmts.insert( index: 0,  element: stmt);
}
```

Alle Funktionen im SUT,
die getestet werden können

Generierte Testsuite

```rust
#[test]
fn some_test() {
    let x : i32 = 2;
    let y : i32 = 1;
    let mut point : Point = Point::new(x, y);
    do_something(&mut point);
}
```

Code instrumentieren

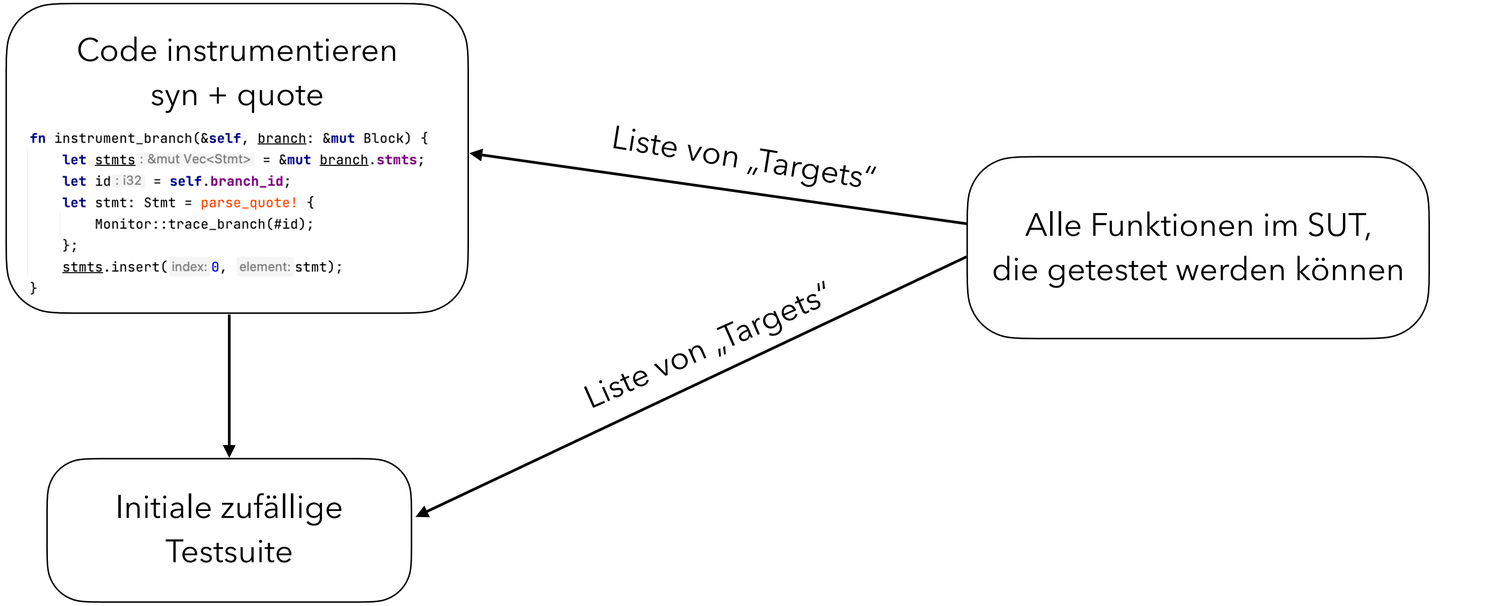
syn + quote

```rust
fn instrument_branch(&self, branch: &mut Block) {
    let stmts : &mut Vec<Stmt> = &mut branch.stmts;
    let id : i32 = self.branch_id;
    let stmt: Stmt = parse_quote! {
        Monitor::trace_branch(#id);
    };
    stmts.insert( index: 0, element: stmt);
}
```

Alle Funktionen im SUT,
die getestet werden können

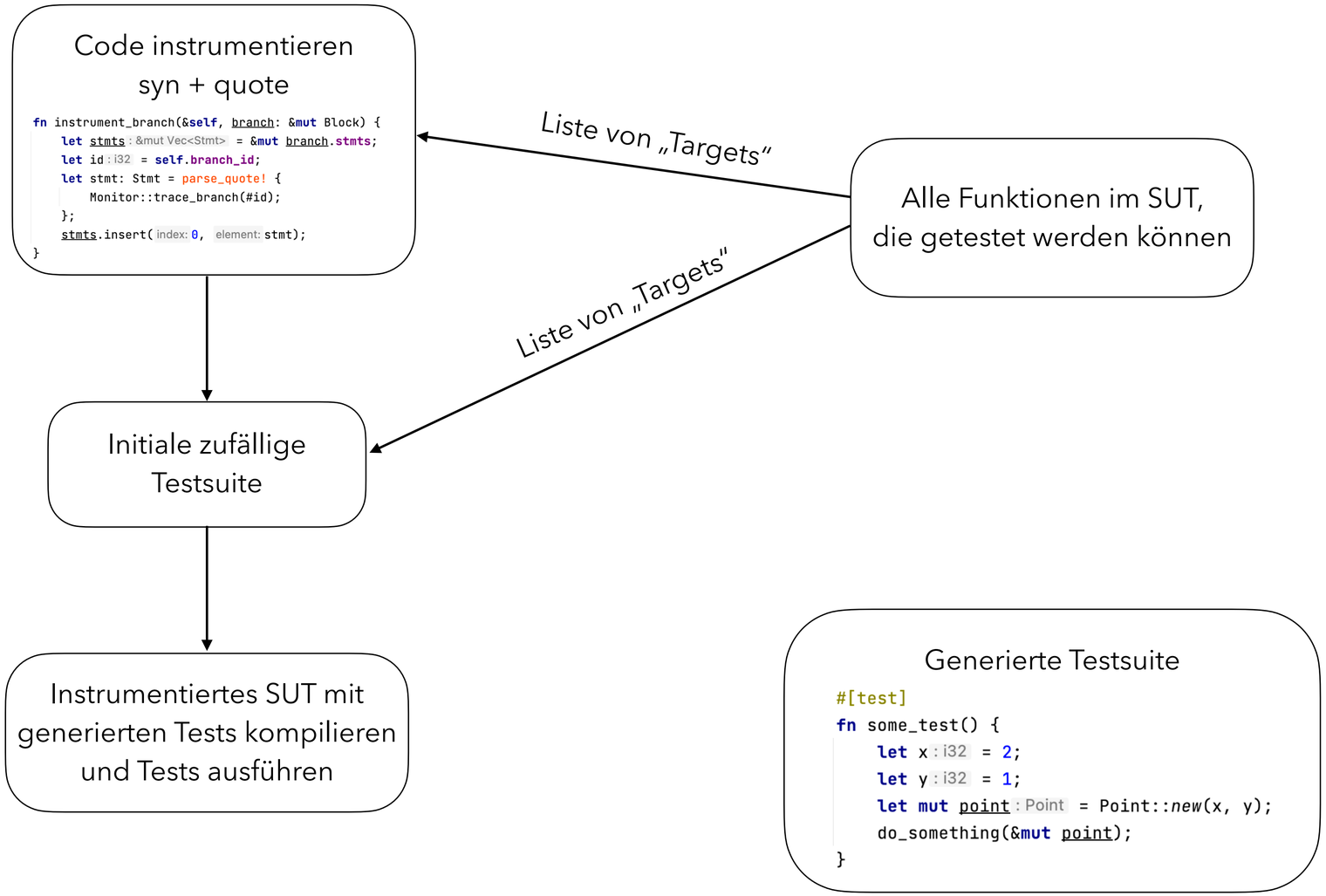
Generierte Testsuite

```rust
#[test]
fn some_test() {
    let x : i32 = 2;
    let y : i32 = 1;
    let mut point : Point = Point::new(x, y);
    do_something(&mut point);
}
```



Generierte Testsuite

```
#[test]
fn some_test() {
    let x : i32 = 2;
    let y : i32 = 1;
    let mut point : Point = Point::new(x, y);
    do_something(&mut point);
}
```


Code instrumentieren
syn + quote
fn instrument_branch(&self, branch: &mut Block) {
let stmts : &mut Vec<Stmt> = &mut branch.stmts;
let id : i32 = self.branch_id;
let stmt: Stmt = parse_quote! {
Monitor::trace_branch(#id);
};
stmts.insert(index: 0, element: stmt);
}
Liste von „Targets“
Alle Funktionen im SUT,
die getestet werden können
Liste von „Targets“
Initiale zufällige
Testsuite
Instrumentiertes SUT mit
generierten Tests kompilieren
und Tests ausführen
Generierte Testsuite
#[test]
fn some_test() {
let x : i32 = 2;
let y : i32 = 1;
let mut point : Point = Point::new(x, y);
do_something(&mut point);
}

Code instrumentieren
syn + quote

```
fn instrument_branch(&self, branch: &mut Block) {
    let stmts : &mut Vec<Stmt> = &mut branch.stmts;
    let id : i32 = self.branch_id;
    let stmt: Stmt = parse_quote! {
        Monitor::trace_branch(#id);
    };
    stmts.insert( index: 0, element: stmt);
}
```

Liste von „Targets“

Alle Funktionen im SUT,
die getestet werden können

Liste von „Targets“

Initiale zufällige
Testsuite

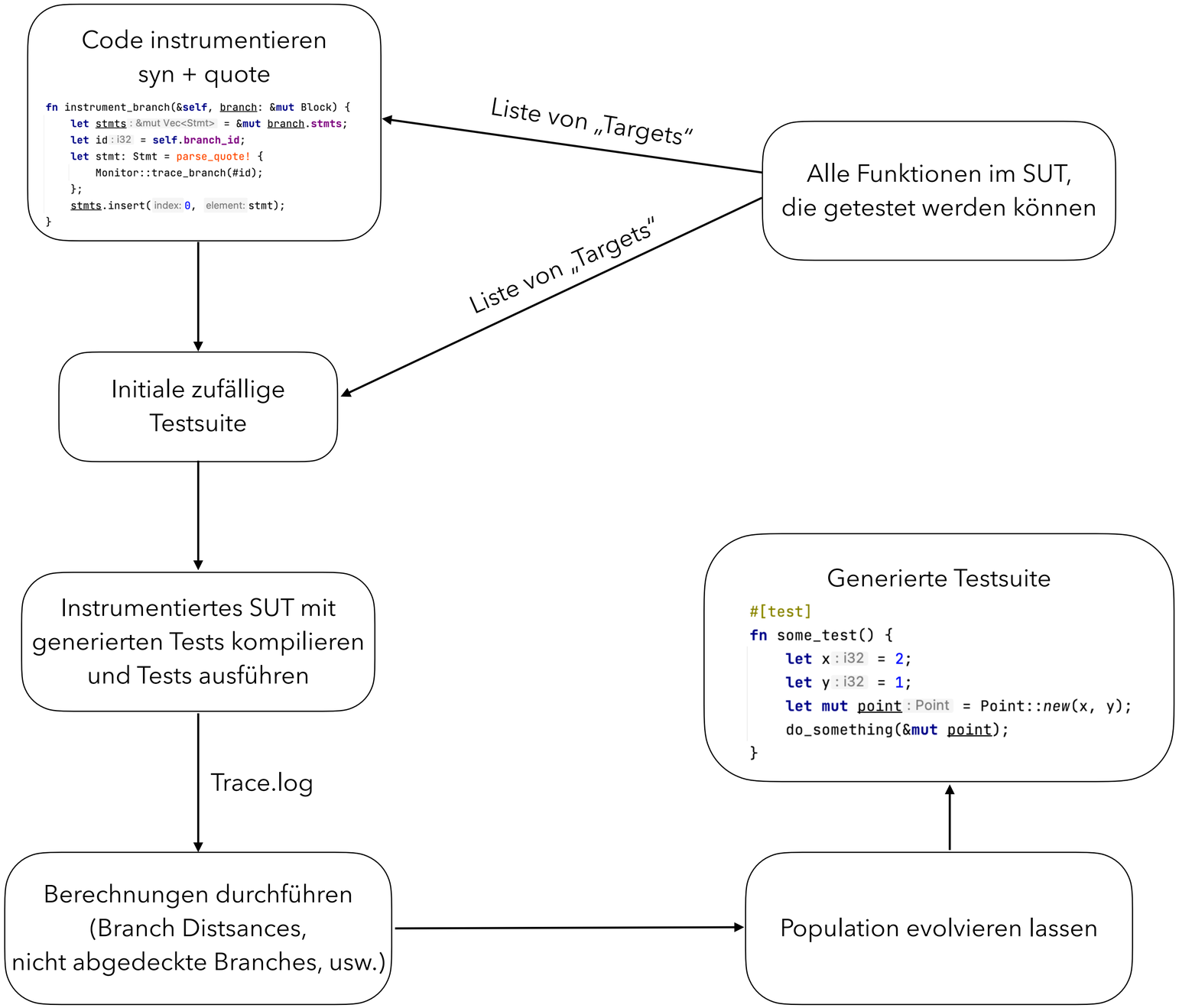
Instrumentiertes SUT mit
generierten Tests kompilieren
und Tests ausführen

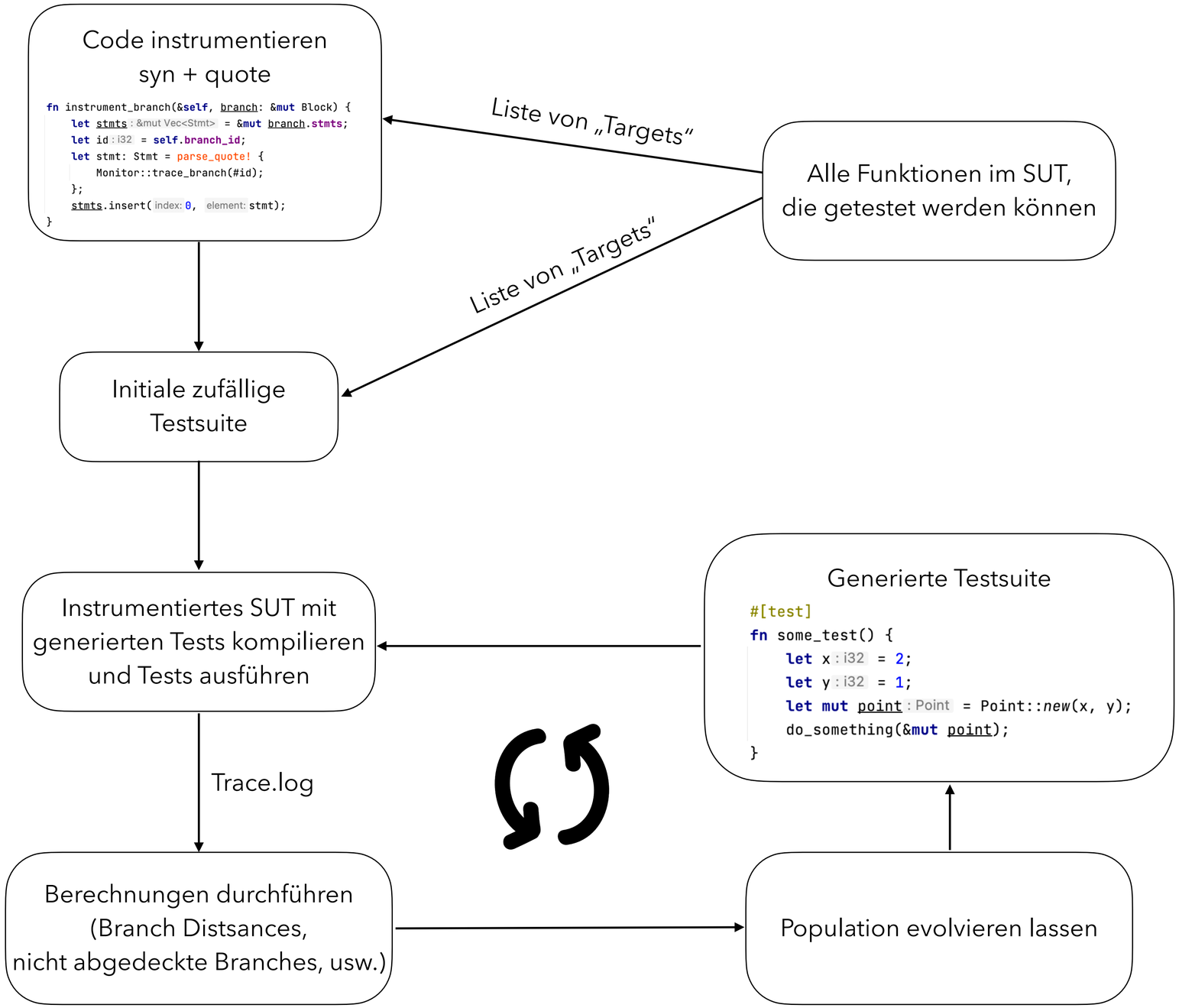
Trace.log

Berechnungen durchführen
(Branch Distsances,
nicht abgedeckte Branches, usw.)

Generierte Testsuite

```
#[test]
fn some_test() {
    let x : i32 = 2;
    let y : i32 = 1;
    let mut point : Point = Point::new(x, y);
    do_something(&mut point);
}
```


Code instrumentieren
syn + quote
fn instrument_branch(&self, branch: &mut Block) {
let stmts : &mut Vec<Stmt> = &mut branch.stmts;
let id : i32 = self.branch_id;
let stmt: Stmt = parse_quote! {
Monitor::trace_branch(#id);
};
stmts.insert(index: 0, element: stmt);
}
Liste von „Targets“
Alle Funktionen im SUT,
die getestet werden können
Liste von „Targets“
Initiale zufällige
Testsuite
Instrumentiertes SUT mit
generierten Tests kompilieren
und Tests ausführen
Trace.log
Berechnungen durchführen
(Branch Distsances,
nicht abgedeckte Branches, usw.)
Population evolvieren lassen
Generierte Testsuite
#[test]
fn some_test() {
let x : i32 = 2;
let y : i32 = 1;
let mut point : Point = Point::new(x, y);
do_something(&mut point);
}


Code instrumentieren
syn + quote
fn instrument_branch(&self, branch: &mut Block) {
let stmts : &mut Vec<Stmt> = &mut branch.stmts;
let id : i32 = self.branch_id;
let stmt: Stmt = parse_quote! {
Monitor::trace_branch(#id);
};
stmts.insert(index: 0, element: stmt);
}
Liste von „Targets“
Alle Funktionen im SUT,
die getestet werden können
Liste von „Targets“
Initiale zufällige
Testsuite
Instrumentiertes SUT mit
generierten Tests kompilieren
und Tests ausführen
Generierte Testsuite
#[test]
fn some_test() {
let x : i32 = 2;
let y : i32 = 1;
let mut point : Point = Point::new(x, y);
do_something(&mut point);
}
Trace.log
Berechnungen durchführen
(Branch Distsances,
nicht abgedeckte Branches, usw.)
Population evolvieren lassen

Chromosome

```rust
#[test]
fn some_test() {
    let x: i32 = 2;
    let y: i32 = 1;
    let mut point: Point = Point::new(x, y);
    do_something(&mut point);
}
```

Chromosome

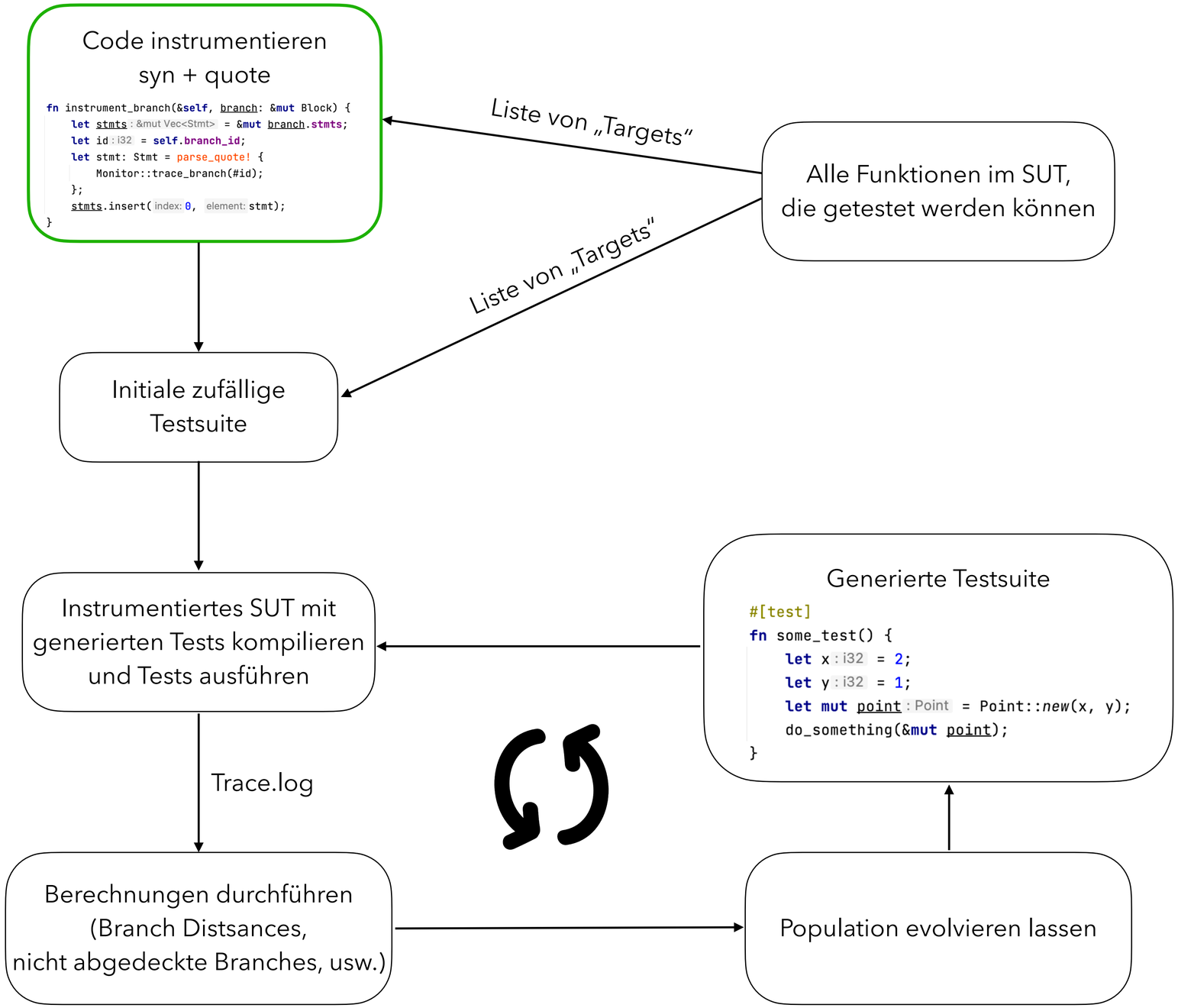
```rust
#[test]
fn some_test() {
    let x: i32 = 2;
    let y: i32 = 1;
    let mut point: Point = Point::new(x, y);
    do_something(&mut point);
}
```

Chromosome

```rust
#[test]
fn some_test() {
    let x: i32 = 2;
    let y: i32 = 1;
    let mut point: Point = Point::new(x, y);
    do_something(&mut point);
}
```

Chromosome

```rust
#[test]
fn some_test() {
    let x: i32 = 2;
    let y: i32 = 1;
    let mut point: Point = Point::new(x, y);
    let mut point2: Point = Point::new(x: y, y: x);
    do_something(&mut point);
}
```


Code instrumentieren
syn + quote
fn instrument_branch(&self, branch: &mut Block) {
let stmts : &mut Vec<Stmt> = &mut branch.stmts;
let id : i32 = self.branch_id;
let stmt: Stmt = parse_quote! {
Monitor::trace_branch(#id);
};
stmts.insert(index: 0, element: stmt);
}
Liste von „Targets“
Alle Funktionen im SUT,
die getestet werden können
Liste von „Targets“
Initiale zufällige
Testsuite
Instrumentiertes SUT mit
generierten Tests kompilieren
und Tests ausführen
Generierte Testsuite
#[test]
fn some_test() {
let x : i32 = 2;
let y : i32 = 1;
let mut point : Point = Point::new(x, y);
do_something(&mut point);
}
Trace.log
Berechnungen durchführen
(Branch Distsances,
nicht abgedeckte Branches, usw.)
Population evolvieren lassen

Code instrumentieren

syn + quote

```
fn instrument_branch(&self, branch: &mut Block) {
    let stmts : &mut Vec<Stmt> = &mut branch.stmts;
    let id : i32 = self.branch_id;
    let stmt: Stmt = parse_quote! {
        Monitor::trace_branch(#id);
    };
    stmts.insert( index: 0, element: stmt);
}
```

Testability Transformation

```
fn div(a: i32, b: i32) -> f64 {
    a as f64 / b as f64
}
```

Code instrumentieren

syn + quote

```
fn instrument_branch(&self, branch: &mut Block) {
    let stmts : &mut Vec<Stmt> = &mut branch.stmts;
    let id : i32 = self.branch_id;
    let stmt: Stmt = parse_quote! {
        Monitor::trace_branch(#id);
    };
    stmts.insert( index: 0, element: stmt);
}
```

Testability Transformation

```
fn div(a: i32, b: i32) -> f64 {
    if b == 0 {
        panic!("Dividing by zero")
    }
    a as f64 / b as f64
}
```

Code instrumentieren

syn + quote

```
fn instrument_branch(&self, branch: &mut Block) {
    let stmts : &mut Vec<Stmt> = &mut branch.stmts;
    let id : i32 = self.branch_id;
    let stmt: Stmt = parse_quote! {
        Monitor::trace_branch(#id);
    };
    stmts.insert( index: 0, element: stmt);
}
```

Testability Transformation

```
fn foo(&self, i: usize) -> i32 {
    self.list[i]
}
```

Code instrumentieren

syn + quote

```
fn instrument_branch(&self, branch: &mut Block) {
    let stmts : &mut Vec<Stmt> = &mut branch.stmts;
    let id : i32 = self.branch_id;
    let stmt: Stmt = parse_quote! {
        Monitor::trace_branch(#id);
    };
    stmts.insert( index: 0, element: stmt);
}
```

Testability Transformation

```
fn foo(&self, i: usize) -> i32 {
    let len : usize = self.list.len();
    if i < 0 {
        panic!("Negative array index");
    } else if i >= len {
        panic!("Index out of bounds");
    }
    self.list[i]
}
```

Code instrumentieren

syn + quote

```
fn instrument_branch(&self, branch: &mut Block) {
    let stmts : &mut Vec<Stmt> = &mut branch.stmts;
    let id : i32 = self.branch_id;
    let stmt: Stmt = parse_quote! {
        Monitor::trace_branch(#id);
    };
    stmts.insert( index: 0, element: stmt);
}
```

Testability Transformation

```
fn add(a: i32, b: i32) -> i32 {
    a + b
}
```

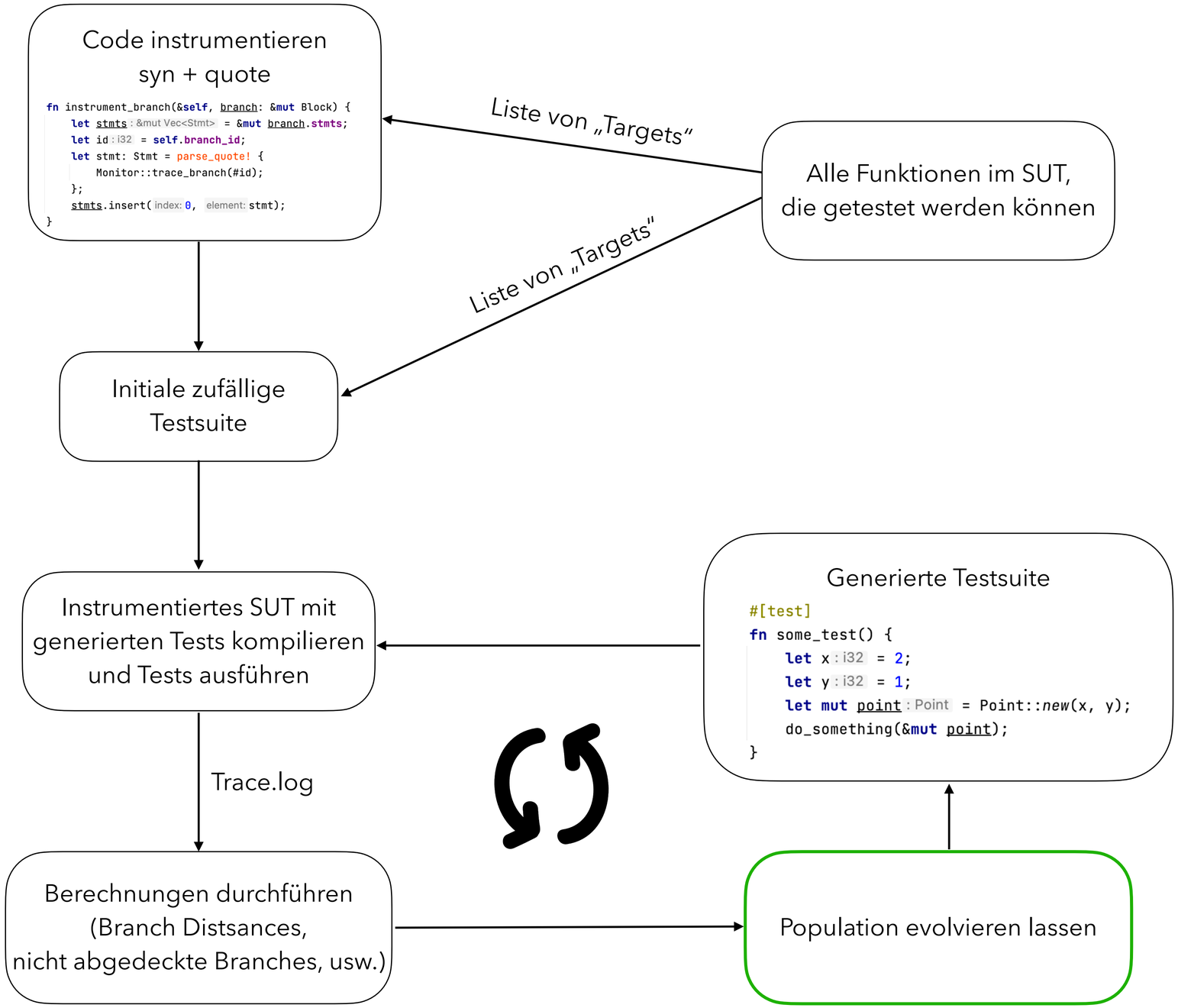
Code instrumentieren

syn + quote

```
fn instrument_branch(&self, branch: &mut Block) {
    let stmts : &mut Vec<Stmt> = &mut branch.stmts;
    let id : i32 = self.branch_id;
    let stmt: Stmt = parse_quote! {
        Monitor::trace_branch(#id);
    };
    stmts.insert( index: 0, element: stmt);
}
```

Testability Transformation

```
fn add(a: i32, b: i32) -> i32 {
    if checkUnderflow(x, y, ADD) < 0 {
        // Panik wird beim Addieren geschoben
    } else if checkOverflow(x, y, ADD) < 0 {
        // Panik wird beim Addieren geschoben
    }
    a + b
}
```


Code instrumentieren
syn + quote
fn instrument_branch(&self, branch: &mut Block) {
let stmts : &mut Vec<Stmt> = &mut branch.stmts;
let id : i32 = self.branch_id;
let stmt: Stmt = parse_quote! {
Monitor::trace_branch(#id);
};
stmts.insert(index: 0, element: stmt);
}
Liste von „Targets“
Alle Funktionen im SUT,
die getestet werden können
Liste von „Targets“
Initiale zufällige
Testsuite
Instrumentiertes SUT mit
generierten Tests kompilieren
und Tests ausführen
Generierte Testsuite
#[test]
fn some_test() {
let x : i32 = 2;
let y : i32 = 1;
let mut point : Point = Point::new(x, y);
do_something(&mut point);
}
Trace.log
Berechnungen durchführen
(Branch Distsances,
nicht abgedeckte Branches, usw.)
Population evolvieren lassen

Population evolvieren lassen

Population evolvieren lassen

Bloat Control

Maximale Länge für Tests

Maximale Verzweigungstiefe

In der Selektionsphase Tests nach Länge ranken