

Embedded Testing 2016

Wegweiser zur Auswahl eines Werkzeuges zur Statischen Codeanalyse



Royd Lüdtkke
Verifysoft Technology GmbH
luedtke@verifysoft.com
+49 781 127 8118-8



Agenda

- Statische Codeanalyse vs. dynamische Analyse
- Kosten zur Fehlerbeseitigung
- Motivation zum Einsatz eines Werkzeuges zur statischen Codeanalyse
- Betriebssystem, Programmiersprachen, Compiler
- Metriken
- Programmierrichtlinien
- Beurteilung durch Testfälle
- Klassifizierung der Analyseergebnisse
- Statische Analyse und Approximation
- Einsatzszenarien
- Performance und Skalierung
- Updates und Upgrades
- Lizenzmodelle
- Eingliederung in vorhandene Toolchain
- Qualifizierung
- Checkliste



Statische vs. dynamische Analyse

Statische Analyse

```
void gyroDisp(const char *sdf){  
  
    char* sBuffer;  
  
    sBuffer = (char*) malloc(strlen(sdf) + 1);  
  
    if (sBuffer){  
        strcpy(sBuffer, sdf);  
    }  
  
    /* Format string for LCD */  
  
    free(sBuffer);  
}
```

Review

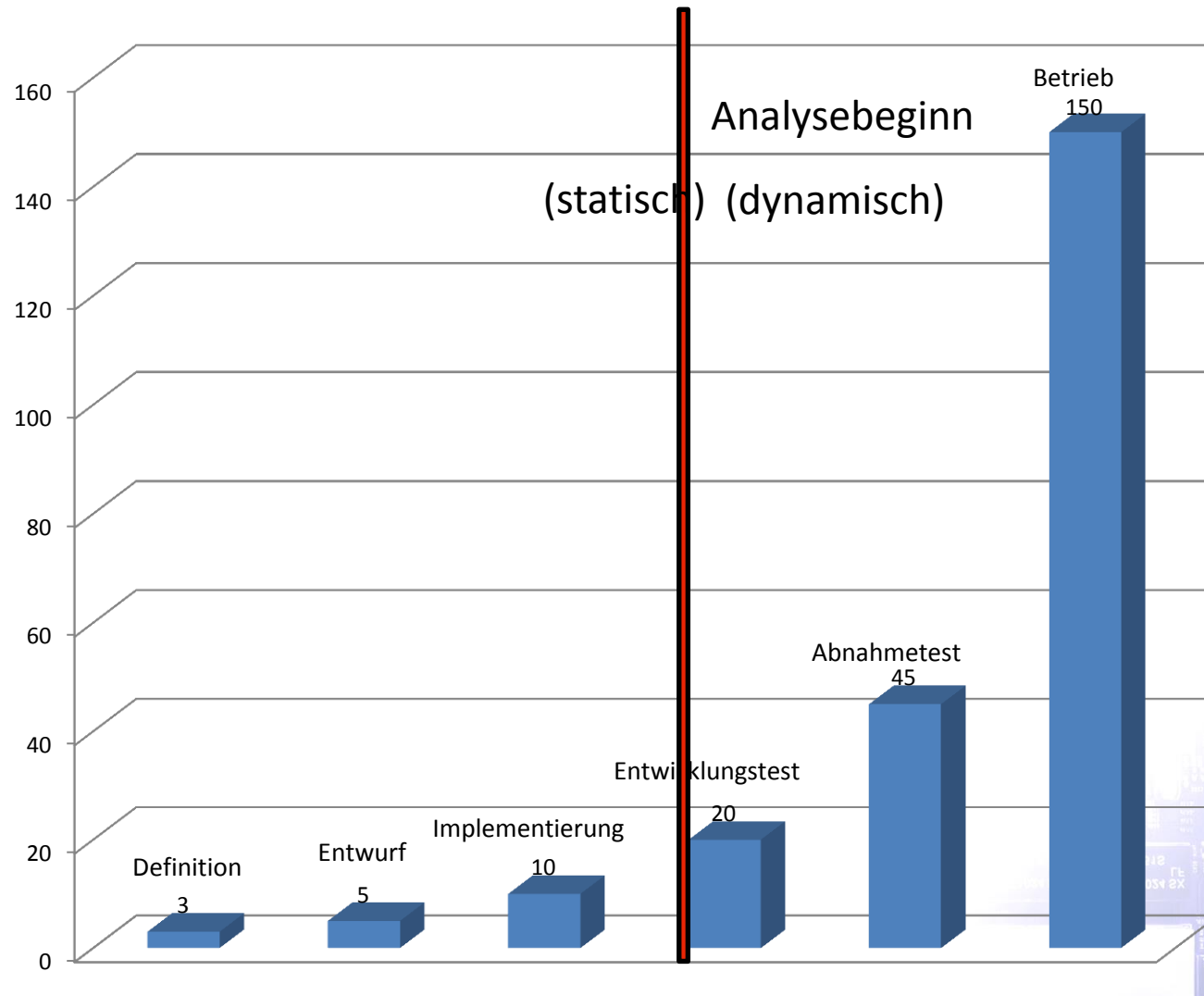


Dynamische Analyse

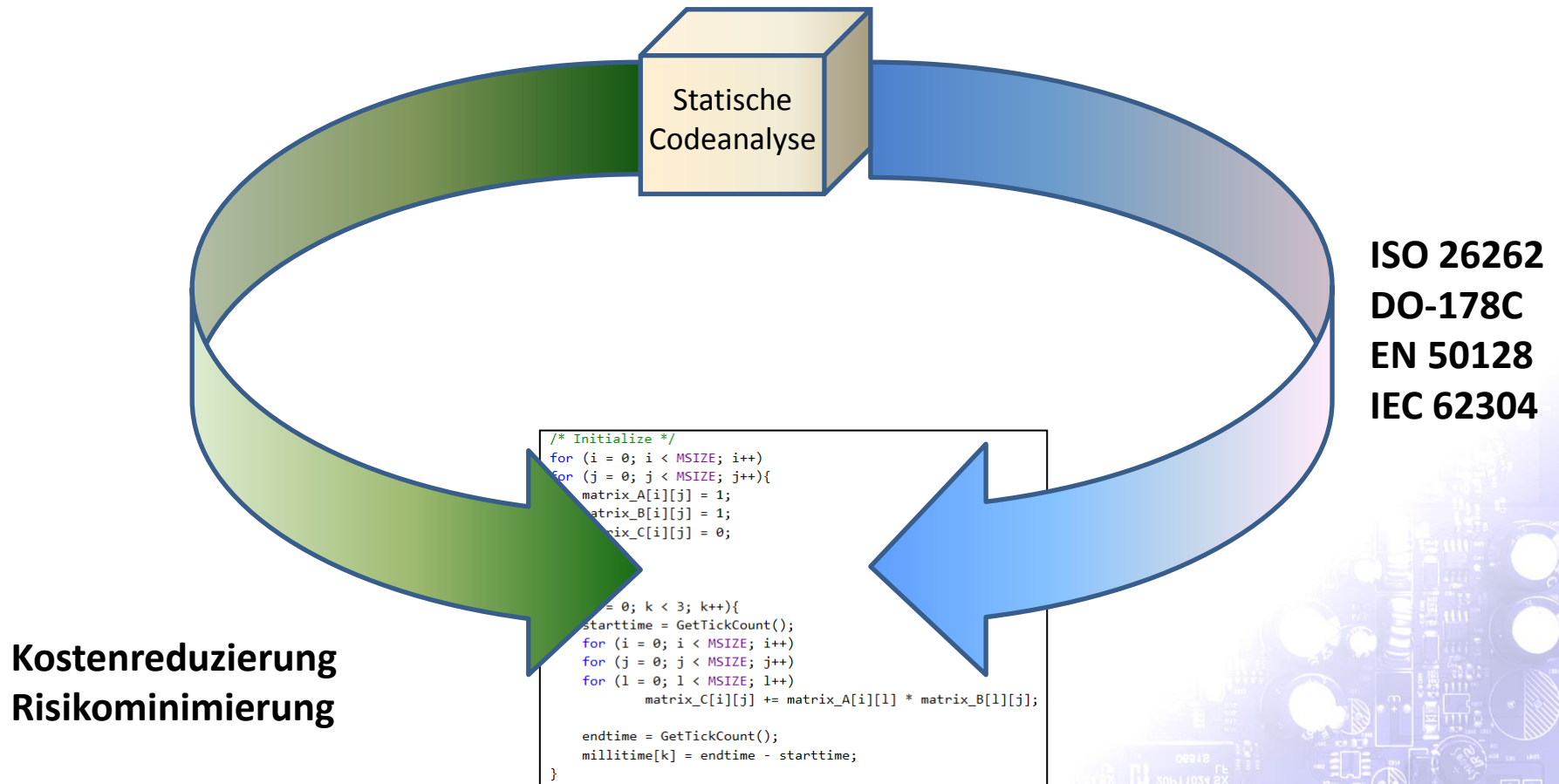


Laufzeitverhalten

Relative Kosten zur Fehlerbeseitigung (nach B. Boehm)



Motivation für den Einsatz eines Werkzeuges zur statischen Analyse



Verschiedene Aufgaben für Werkzeuge zur statischen Codeanalyse

Fehleraufdeckung (Syntax, Semantik)

Aufdecken von Datenfluss- und Kontrollflussanomalien

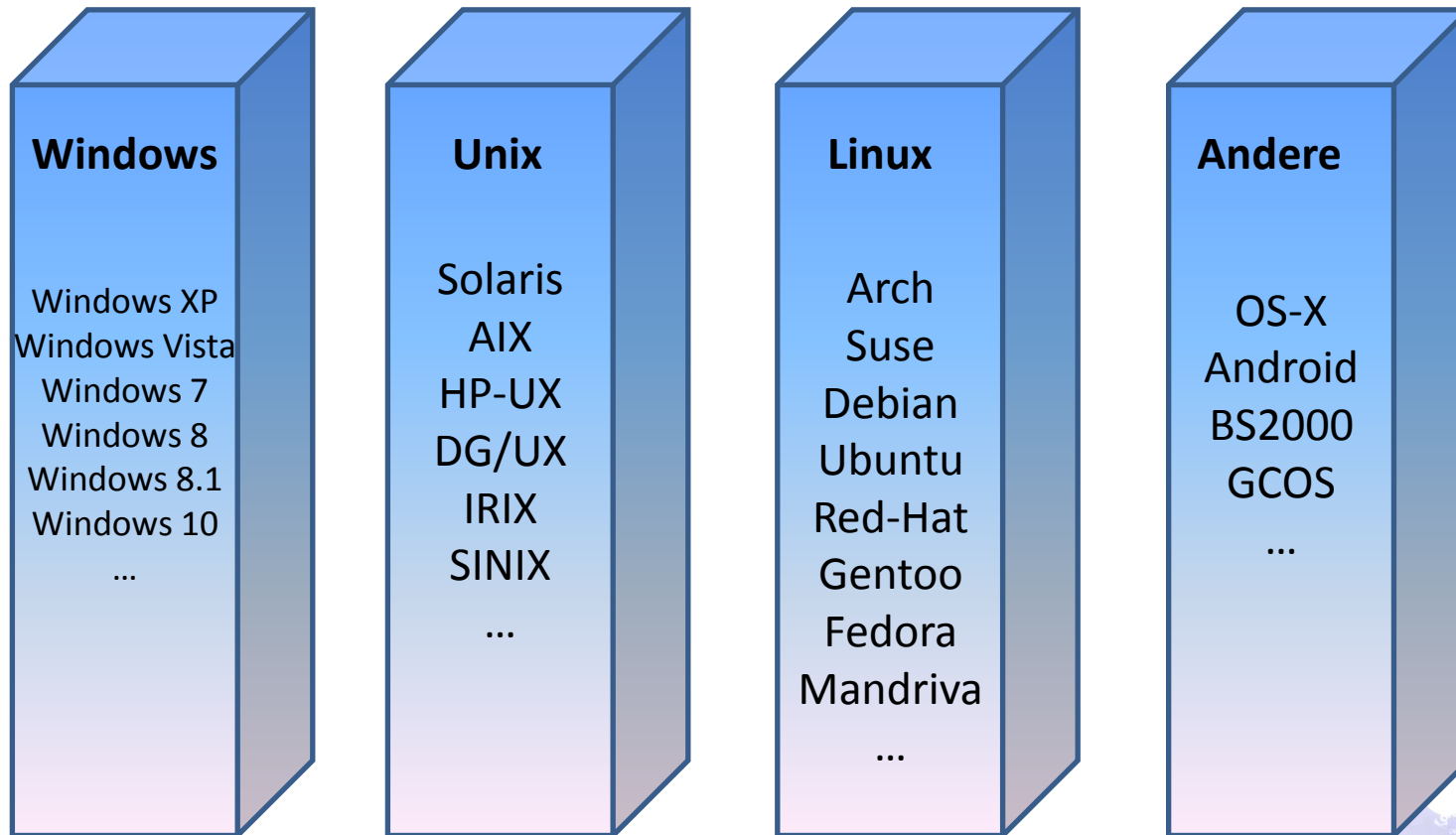
Erstellen von Metriken

Prüfung auf Einhaltung von Programmierrichtlinien/Standards

Aufdecken von Sicherheitsproblemen

Architekturanalyse

Welches Betriebssystem wird unterstützt?



Service Pack?

Patch Level?

Programmiersprachen und Standards

```
// C#
using System.Console;

private class Program {
    public static void Main() {
        WriteLine("Hello World!");
    }
}
```

```
/* C */
#include <stdio.h>

int main(void){
    printf("Hello World!");
    return 0;
}
```

```
! Fortran
program helloworld
write (*,*) 'Hello World!'
end program helloworld
```

```
// Java
public class HelloWorld
{
    public static void main (String[] args)
    {
        System.out.println("Hello World!");
    }
}
```

```
REM BASIC
10 PRINT "Hello World! "
20 GOTO 10
```

```
//C++
#include <iostream.h>
int main(void){
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

```
* COBOL
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO-WORLD.
PROCEDURE DIVISION.
    DISPLAY 'Hello World!'.
    STOP RUN.
```

Beispiel „C“:

ISO/IEC 9899:2011 (C11)

ISO/IEC 9899:1999 (C99)

Amendment 1990 (C94 / C95)

ISO/IEC 9899:1990 (C89 / C90)

```
--Ada
with Text_To; use Text_To
procedure hello is
begin
    put("Hello World!");
end hello
```

```
(* Pascal *)
program HelloWorld;
begin
    writeln('Hello World!');
end.
```


Beispiel – Anonyme Strukturen (ab C11)

```
struct aStruct
{
    int A;
    char B;

    struct
    {
        int C;
        char D;
    }; /* Kein Variablenname! */

    union
    {
        int E;
        char F;
    }; /* Kein Variablenname! */
};
```

Warum ist die Kenntnis des Compilers relevant?

Beispiel HI-TECH compiler 9.83:

```
typedef struct foo
{
    short long int bar;
} f;

int main(void)
{
    f baz;
    baz.bar = 4;
    return 0;
}
```

Beispiel MPLAB C30 compiler:

```
typedef int
__attribute__((__mode__(SI))) int32;

int32 add32(int32 a, int32 b)
{
    return(a + b);
}
```

Metriken zur Softwarebeurteilung

Auswahl aus über 100 bekannten Metriken:

- **LOC** (Lines Of Code): Anzahl der Codezeilen
- **Watson & McCabe**: Zyklomatische Komplexität
- **Halstead**: Implementierungsabschätzung durch abgeleitete Kenngrößen für
 - Lesbarkeit
 - Implementierungsaufwand
 - Implementierungszeit
- **HIS** (Herstellerinitiative Software): Von Audi, BMW, Daimler, Porsche und VW entwickelte Metriken zur Ermittlung von Softwarequalität und Bewertung von Entwicklungsprozessen

Programmierstandards (Auswahl)

- **MISRA C/C++** (entwickelt für Applikationen im Automotive Umfeld, kommt zunehmend aber auch in anderen Bereichen zur Anwendung)
- **JPL** (von der NASA in Anlehnung an MISRA C entwickelter Coding Standard für sicherheitskritische Anwendungen)
- **Power of Ten** (zehn von der NASA/JPL veröffentlichte Programmierrichtlinien für sicherheitskritische Anwendungen)
- **JSV AV** (ein ursprünglich von Lockheed Martin für das „Joint Strike Fighter“-Projekt entwickelter, auf MISRA basierender Standard)

Programmierstandards (Auswahl)

- **CERT Secure Coding Standard** (von der CERT Coding Initiative sowie Mitgliedern der Software Development- und Software Security Communities entwickelter Standard, um betriebssichere, verlässliche und sicherheitstechnisch abgesicherte Software zu erstellen)
- **Philips Healthcare – C++ Coding Standard** (ein von Philips Healthcare entwickelter und gepflegter Coding Standard. Er ist verpflichtend für alle medizinischen Philips C++-Implementierungen, wird aber zunehmend auch von anderen Unternehmen im Bereich Medizintechnik eingeführt)

Tests zur Beurteilung der Leistungsfähigkeit

Intraprozedurale und interprozedurale
Problemstellungen testen!

Kontrollflussanalyse (dead code, infinite loop)

Datenflussanalyse (uninitialized variable, division by zero)

Zeigerarithmetik

Numerische Problemstellungen

Nebenläufigkeit (data race, deadlock)

Statische Speicherallokation (buffer overrun, buffer underrun)

Dynamische Speicherallokation (memory leak, double free)

Interprozeduraler Fehler

Modul 1:

```
int main(void){  
    int* p = (int*)malloc(42 * sizeof(int));  
    if (p) {  
        eineFunktion(p);  
        free(p);  
    }  
    return 0;  
}
```

Modul 2:

```
void eineFunktion(int* z){  
    free(z);  
}
```


Kontrollflussanalyse

Beispiel: Dead Code

```
int main(void)
{
    int a = 25;
    int b = 50;

    if (a > 0)
    {
        if (!a && b)
        {
            b = a + b; /* nicht erreichbar */
            return b;
        }
    }
    return 0;
}
```

Datenflussanalyse

Beispiel: Nichtinitialisierte Variable

```
int somefunction(int x){  
    int y;  
    if (x >= 128){  
        y = 12;  
    }  
    return y;  
}
```

Speicherüberlauf

Statisch

```
void main(void)
{
    char array[50];
    char c = array[50];
}
```

Dynamisch + Dereferenzierung von Null-Pointer

```
void main(void)
{
    char* p;
    char* q;
    int i = 0;

    p = (char*)malloc(42 * sizeof(char));

    q = p;
    for (i = 0; i <= 42; i++)
        *q++ = 's';

    free(p);
}
```

Datentypüberlauf

```
#include <stdio.h>

struct {
    char name[20];
    int alter;
} person;

void main(void)
{
    scanf("%s", person.name);
}
```

Datenverlust durch implizite/explicite Typumwandlung

```
void main(void)
{
    int u;
    unsigned int v;
    u = -120;
    v = u + 42;
}
```

```
void main(void)
{
    double d;
    int i;
    d = 7.345712;
    i = (int)d;
}
```

Division durch Null

```
void main(void)
{
    int x = 0;
    int y = 27;
    int z = y / x;
}
```

```
void main(void)
{
    float x = 0.;
    float y = 27.;
    float z = y / x;
}
```

Kein Rückgabewert

```
int noretval(int j) {  
    if (j == 1) return;  
    else return 1;  
}
```


Memory Leak

```
int leak(void)
{
    char* p = (char*)malloc(42 * sizeof(int));
    if (!p)
        return 0;

    if (aFunction())
        return -1; /* Memory Leak */
    free(p);
    return 0;
}
```

Zugriff auf bereits freigegebenen Speicher

```
void Zugriff_nach_Freigabe(void)
{
    int* p1;
    int* p2;
    p1 = (int*)malloc(42 * sizeof(int));
    if (p1)
    {
        p2 = p1;
        free(p1);
        p2[25] = 16;
    }
}
```

Data Race

```
long x;

void main(void){
    DWORD   threadId_1, threadId_2;
    HANDLE   threadHandle_1, threadHandle_2;

    threadHandle_1 = CreateThread(NULL, 0, &increase, NULL, 0, &threadId_1);
    threadHandle_2 = CreateThread(NULL, 0, &increase, NULL, 0, &threadId_2);

    CloseHandle(threadHandle_1);
    CloseHandle(threadHandle_2);
}
```

```
DWORD WINAPI increase(LPVOID arguments){

    x = x + 1;

    return 0;
}
```

Auflösung von Funktionszeigerzielen

```
typedef void(*funcCtrl)(void);

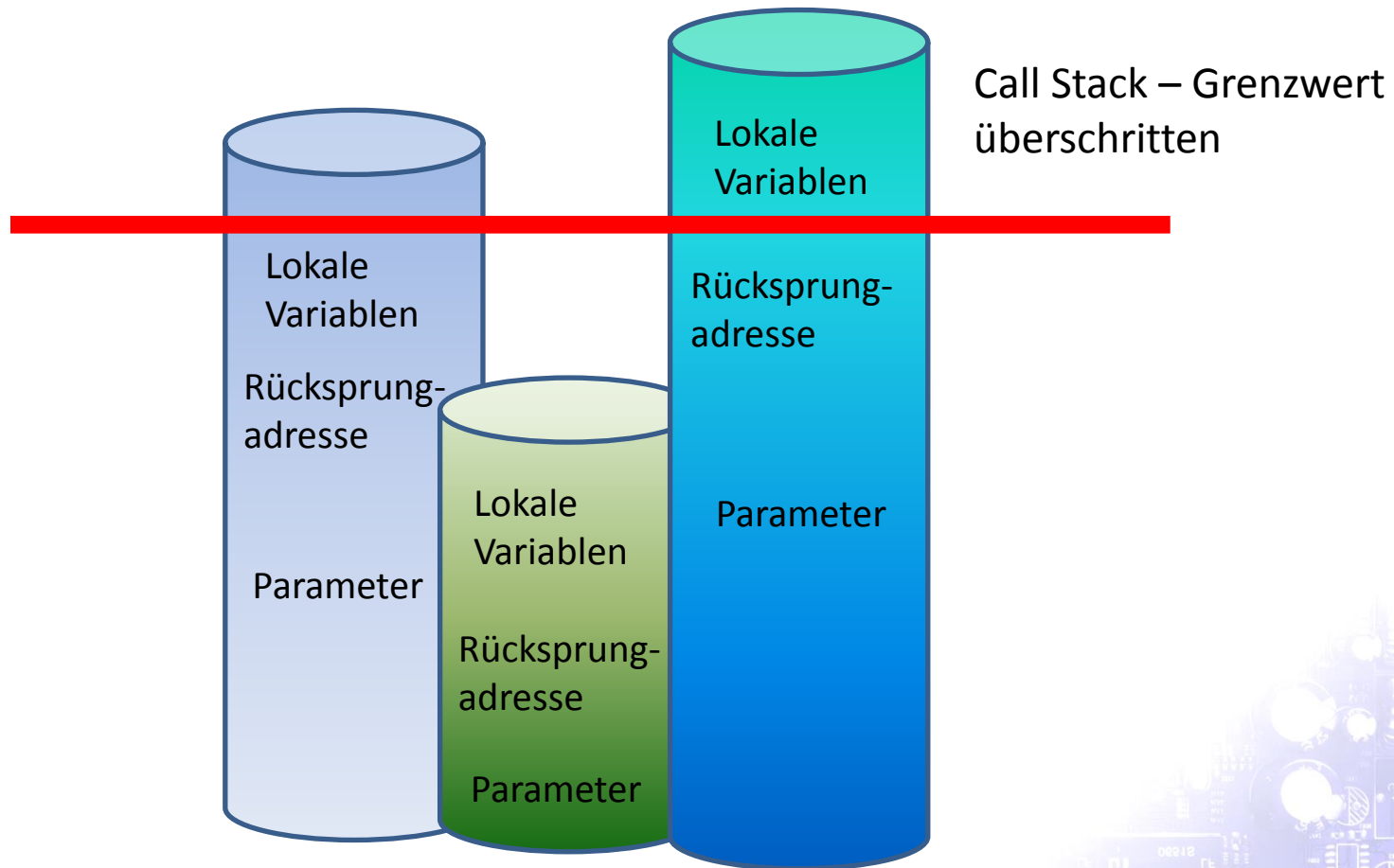
void setCtrl(funcCtrl* x, int op){

    switch (op){
        case 0: *x = ctrlIdle;
        break;
        case 1: *x = ctrlStandard;
        break;
        case 2: *x = ctrlHigh_Performance;
        break;
        case 3: *x = ctrlTest;
        break;
        default: *x = ctrlTest;
    }
}
```

Erkennen von Inline-Assembler

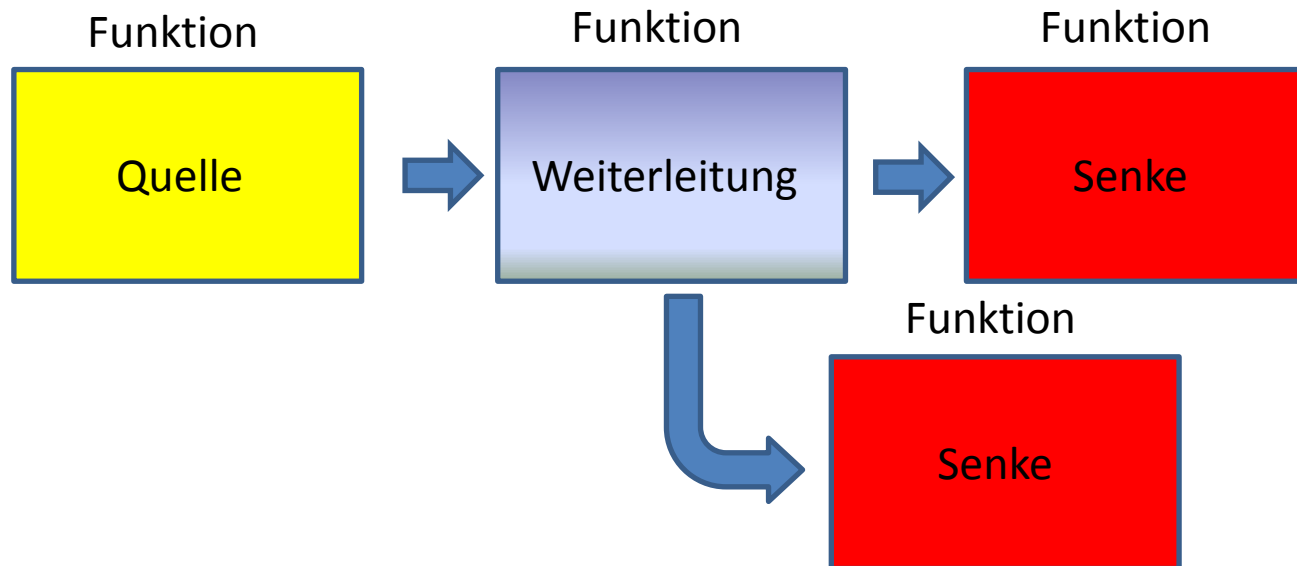
```
void aFunction(void){  
    __asm__("1:"           "\n\t"  
            "Sbiw %0,1"    "\n\t"  
            "brcc 1b"  
            : "+w" (count));  
}
```

Berechnung der Call Stack-Tiefe



Aufdecken von Sicherheitsproblemen

1. SQL Injection, Buffer Overflow

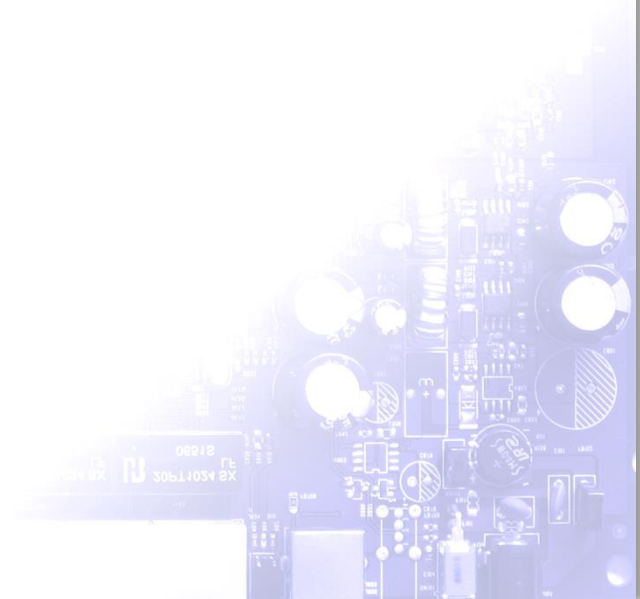


2. Unverschlüsselt abgespeicherte Passwörter

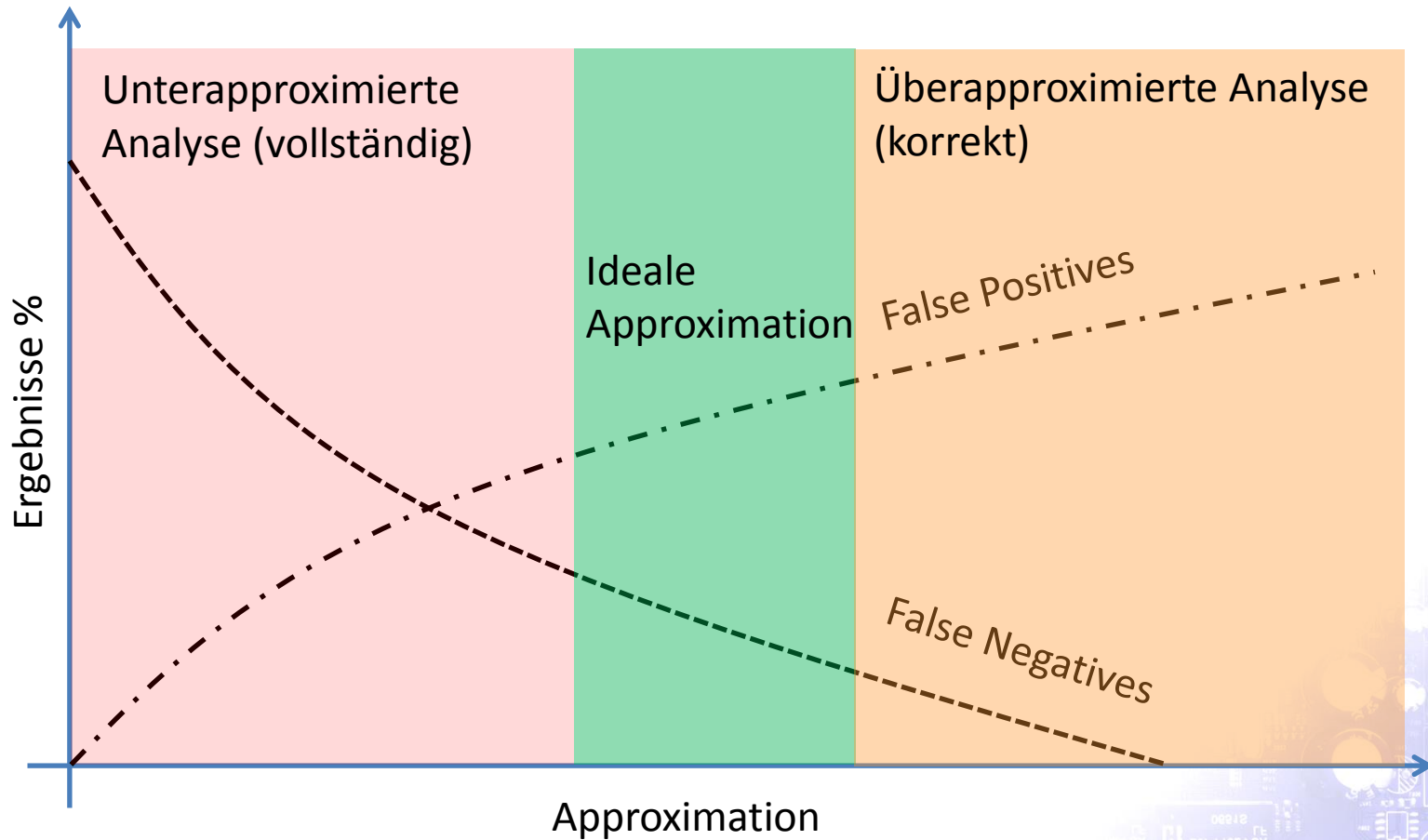
Klassifizierung der Analyseergebnisse

- Technische Interpretation
 - **True Positive**
Ein wirklicher Fehler wurde detektiert
 - **False Positive**
Es wurde fälschlicherweise ein Fehler gemeldet
 - **False Negative**
Ein Fehler wurde bei der Analyse übersehen
 - **True Negative**
Das Werkzeug löst keinen Fehlalarm aus

- Einfachere Interpretation
 - **True Positive**
Ein Fehler, der behoben werden sollte
 - **False Positive**
Kein Handlungsbedarf

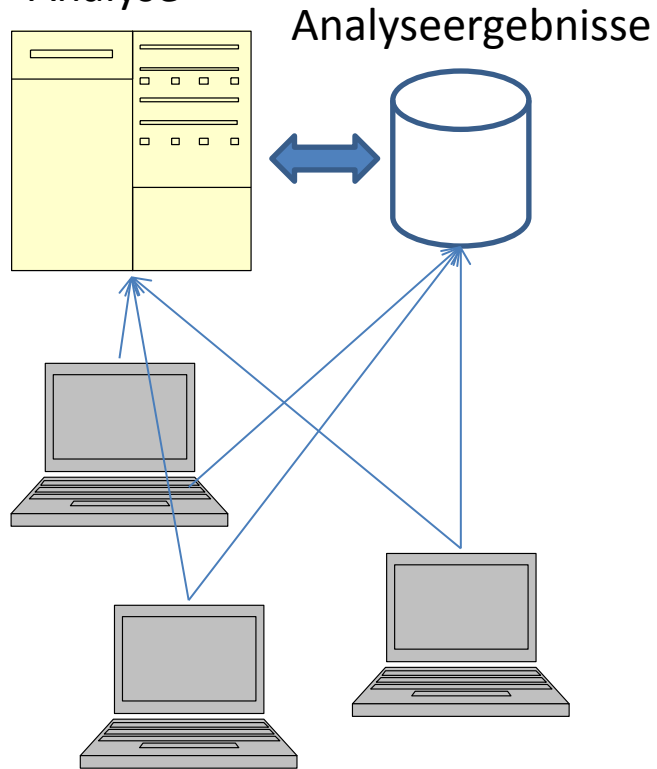


Statische Analyse und Approximation



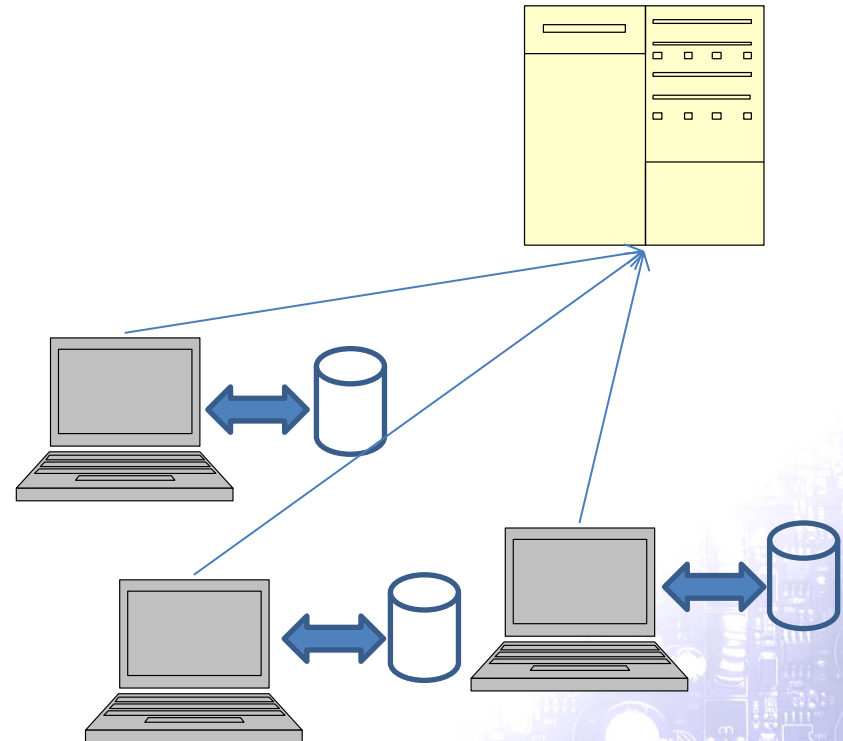
Einsatzszenarien

Build-Server +
Analyse



Implementierung

Build-Server

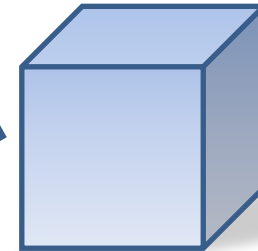
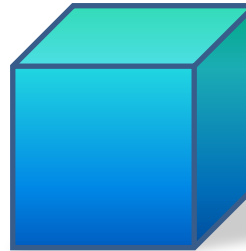
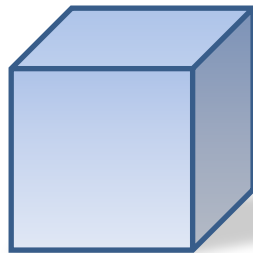


Implementierung + Analyse

Anbindung an externe Werkzeuge

Version Control System
(Subversion, git)

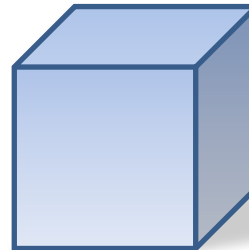
Statische Analyse



Continuous Integration
(Jenkins, Hudson)



Bug Tracking System
(Bugzilla)



Performance und Skalierung

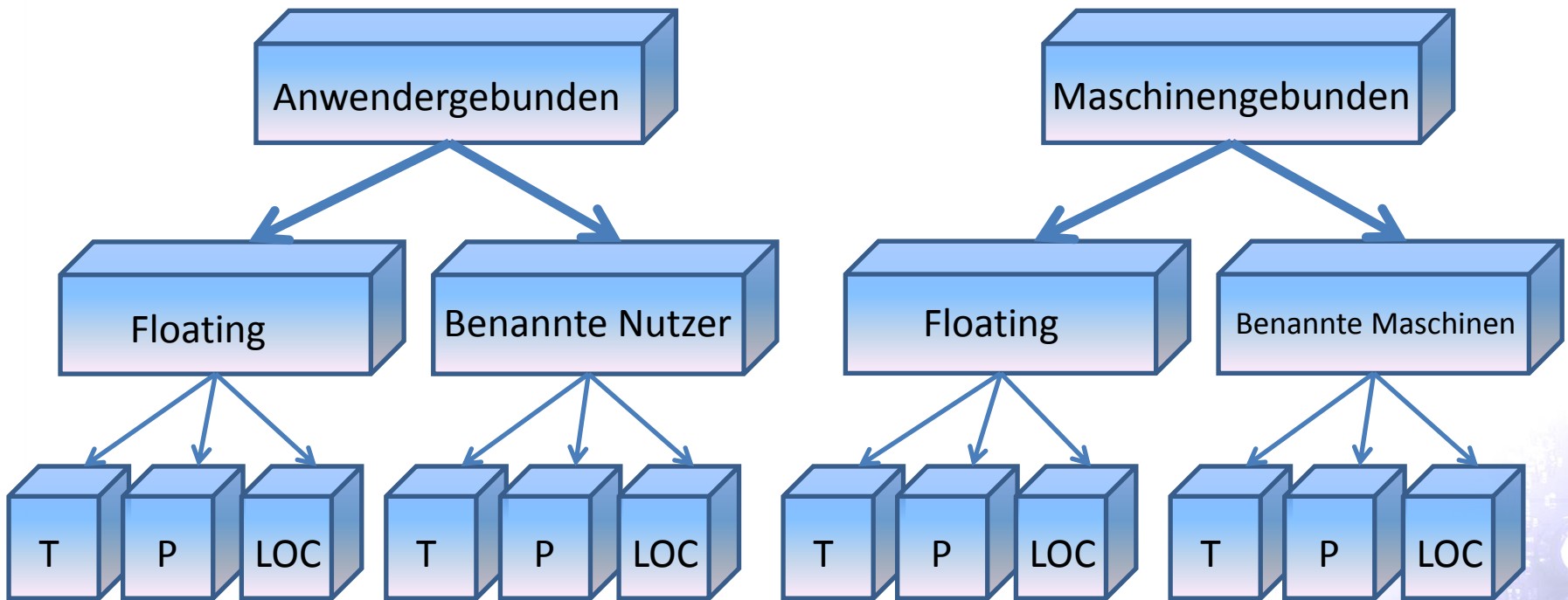
- Welchen Umfang an Codezeilen kann das Werkzeug in einem vertretbaren Zeitrahmen noch analysieren?
- Wie groß ist der Memory-Footprint?
- Skaliert das Werkzeug auf multicore Hardware?
 - Kann die Analyse evtl. auf mehrere Maschinen verteilt werden?
- Kann der Konsum von Hardware-Ressourcen (Speicher, Threads, Prozesse) durch Konfiguration limitiert werden?
- Ist inkrementelle Analyse möglich?

Sichere Update-Prozedur?

- Wie können bestehende Analyseergebnisse gesichert / archiviert werden?
- Besteht bei einem Versionsupdate oder -upgrade Gefahr durch Datenverlust?

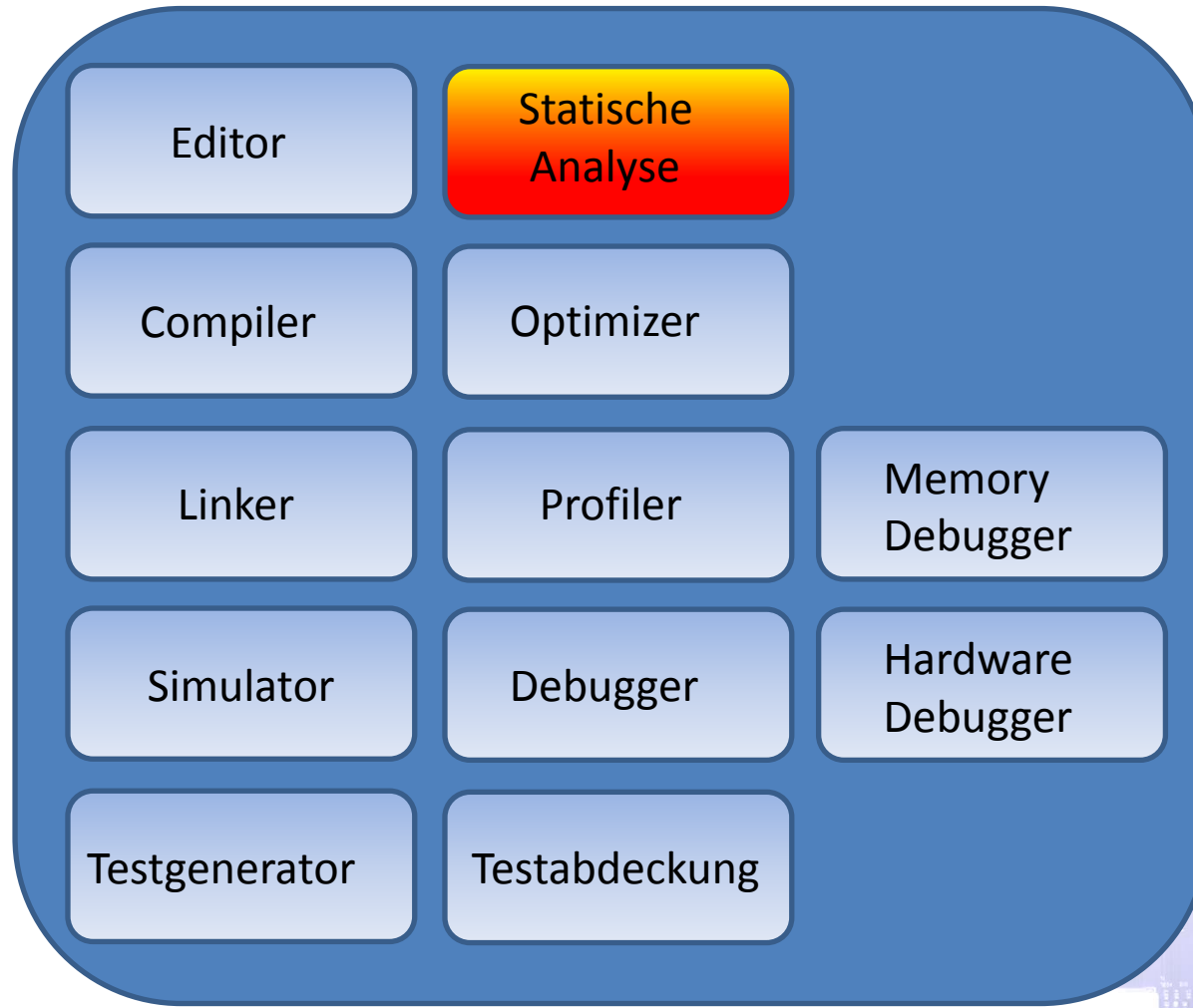


Lizenzmodelle (Auswahl)

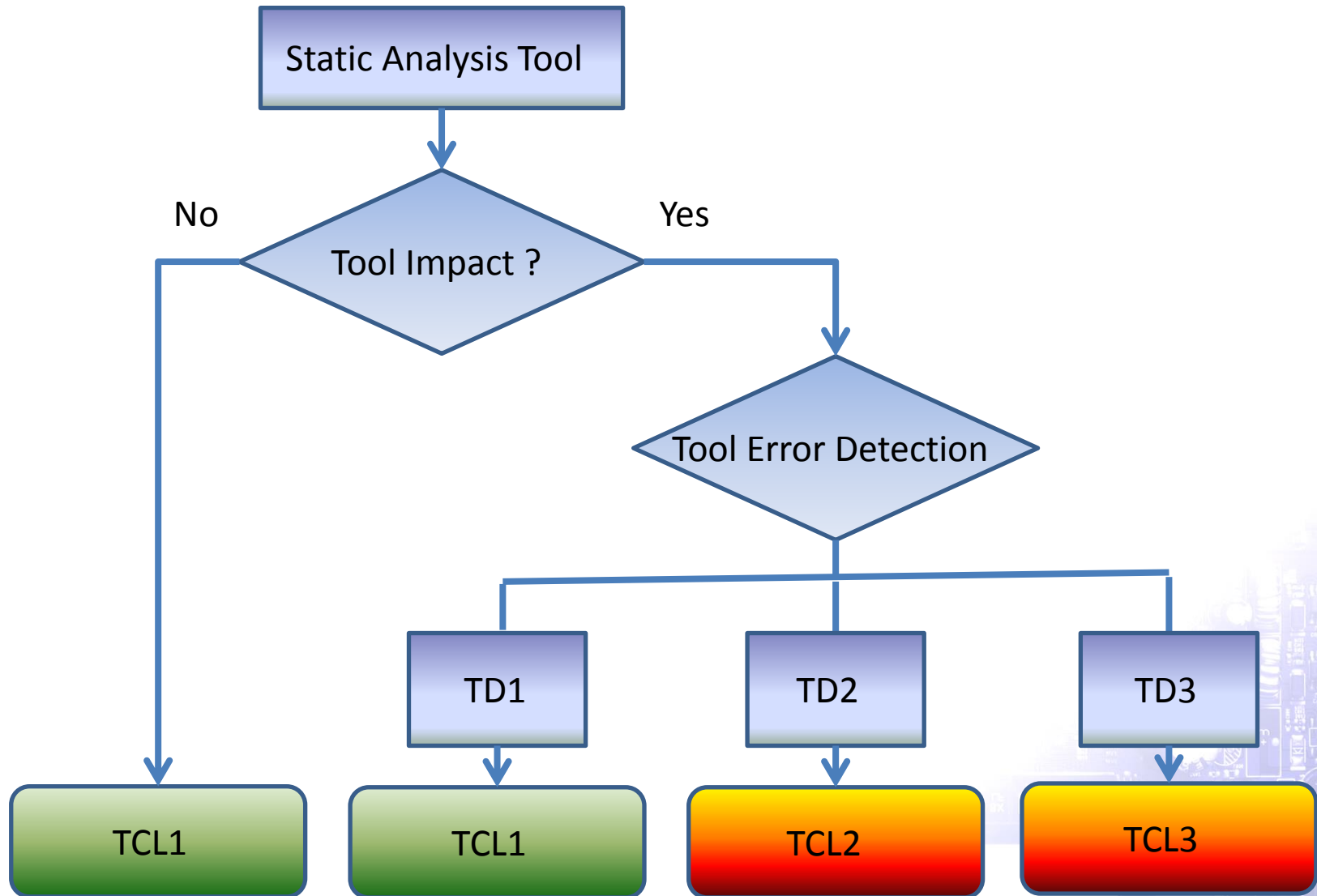


T = Temporär P = Permanent LOC = Lines Of Code

Eingliederung in vorhandene Toolchain



Tool Confidence Level (TCL)



Bedeutung von Zertifizierungen



CERTIFICATE NO FS/71/220/14/0044

ZERTIFIKAT NR.

PAGE 1/1

SEITE(N)

LICENCE HOLDER
GENEHMIGUNGSEHABER

MANUFACTURING PLANT
FERTIGUNGSGEPLATZ

PROJECT NO/ID
PROJEKT NR./ID

LICENSED TEST MARK
GENEHMIGTES PRÜFZEICHEN

CERT. REPORT NO.
ZERTIFIKATSBERICHT NR.

G1TM-AU03



G1TM0005

Tested according to
Geprüft nach

IEC 61508:2010
ISO 26262:2011
EN 50128:2011

Certified product(s)
Zertifizierte Produkte

Model(s)
Modell(e)

Release 4.0, Patchlevel 1

Technical Data and
Parameter
Technische Daten und Parameter

Usable in development of safety related software acc. to:

- ISO 26262 up to ASIL D, TCL1 can be reached
- IEC 61508 up to SIL 4, class T2 tool
- EN 50128 up to SW-SIL 4, class T2 tool

Specific Requirements
Spezifische Anforderungen

The certificate is based on voluntarily tests. Any changes to the design, components or processing may require repetition of some parts of the qualification in order to retain the certification. The certificate report is an integral part of this certificate. The safety related application guidelines (refer to G1TM0005) shall be maintained.

Eine Zertifizierung bescheinigt die Einsatzfähigkeit eines Werkzeuges für einen definierten Einsatzbereich.

Die Notwendigkeit einer Qualifizierung des Werkzeuges als Teil der eingesetzten Toolchain bleibt in den meisten Fällen erforderlich.

Qualification Kit



Ein gutes Qualification Kit erleichtert den Qualifizierungsprozess erheblich

Checkliste

- ✓ Die Analysesoftware unterstützt die eingesetzten Betriebssysteme
- ✓ Alle in den Projekten verwendeten Programmiersprachen werden unterstützt
- ✓ Alle verwendeten Compiler werden unterstützt
- ✓ Compiler-Modelle lassen sich konfigurieren bzw. können selbst erstellt werden
- ✓ Das Werkzeug zeigt eine gute Leistung bei der Fehleraufdeckung und das Verhältnis True Positives / False Positives ist zufriedenstellend
- ✓ Nebenläufigkeitsfehler können in der projektspezifischen Implementierung erkannt werden
- ✓ Programmierschnittstellen erlauben das Erstellen eigener Prüfungen
- ✓ Das Werkzeug kann die geforderten Metriken erheben
- ✓ Erweiterung zur Erhebung eigener Metriken ist möglich
- ✓ Das Analysetool prüft auf Einhaltung von Coding Rules /Standards
- ✓ Coding Rules lassen sich anpassen, bzw. können selbst erstellt werden
- ✓ Die Analyseergebnisse sind leicht verständlich und übersichtlich präsentiert
- ✓ Das Tool ist performant und skaliert auf Multicore-Maschinen
- ✓ Das Werkzeug ist zertifiziert
- ✓ Der Hersteller bietet ein Qualification Kit an
- ✓ Gewünschte Zugriffsregelung / Autorisierung hinsichtlich der Analyseergebnisse lässt sich umsetzen
- ✓ Das Werkzeug eignet sich für die Projektgröße
- ✓ Die Dokumentation ist vollständig und leicht verständlich
- ✓ Das Lizenzmodell wird den Anforderungen gerecht und der Lizenzpreis ist angemessen
- ✓ Eine Anbindung an externe Werkzeuge ist möglich (Continuous Integration, Bug Tracking, Versioning ...)
- ✓ Es wird nur eine kurze Einarbeitungszeit benötigt
- ✓ Der Hersteller liefert guten und schnellen Support
- ✓ Eine Datensicherung ist problemlos möglich
- ✓ Durch Updates und Upgrades besteht keine Gefahr für bestehende Daten

VIELEN DANK !

Royd Lüdtke
Verifysoft Technology GmbH
luedtke@verifysoft.com
+49 781 127 8118-8

