Wegweiser zur Auswahl eines Werkzeuges zur statischen Codeanalyse

Werkzeuge zur statischen Codeanalyse ermöglichen das Auffinden von Softwarefehlern bereits früh im Entwicklungsprozess und tragen damit dazu bei Kosten und Risiken eines Projektes zu minimieren.
Die auf dem Markt erhältlichen Tools unterscheiden sich zum Teil erheblich hinsichtlich Preis und Leistung. Es ist daher nicht einfach das geeignete Werkzeug für einen definierten Einsatzbereich auszuwählen. Das vorliegende Dokument zeigt einige Beurteilungskriterien auf, die Sie dabei unterstützen können, sich für ein zu Ihrem Projekt passendes Werkzeug zu entscheiden.

Was ist eine „statische Codeanalyse“?

Eine statische Codeanalyse ist ein Quellcode Review, entweder manuell durchgeführt, oder automatisiert von einem intelligenten Programm. Vornehmlich dienen solche Analysetools der Aufdeckung von Programmierfehlern, können aber je nach Funktionsumfang auch die Einhaltung von Kodier-Regeln/Standards überprüfen oder z.B. Sicherheitslücken auffinden.

Was unterscheidet die statische von der dynamischen Codeanalyse?

Wird ein Programm ausgeführt um über sein Verhalten zur Laufzeit Informationen zu gewinnen, spricht man von dynamischer Analyse. Dynamische Analysen sind natürlich unumgänglich, setzen aber voraus, dass das Programm bereits weit genug entwickelt ist, um ablauffähig zu sein und dass darüber hinaus Testfälle erstellt wurden.
Diese Voraussetzungen entfallen bei der statischen Codeanalyse. Es wird hier nur der Quellcode eingelesen und untersucht. Als Vorbedingung ist es hinreichend, dass das Programm oder Programmteil fehlerfrei kompiliert werden kann. Analyseläufe können daher vom Entwickler bereits für Teilimplementierungen angestoßen werden. Da die statische Codeanalyse keine Testfälle benötigt, ist sie praktisch zu jeder Zeit unkompliziert durchführbar.

Welche Programmiersprachen werden vom Werkzeug unterstützt?

Natürlich sollte das gewählte Werkzeug die in Ihrem Projekt eingesetzte(n) Programmiersprache(n) auch unterstützen. Bitte achten Sie hier auch auf Programmiersprachenstandards. Falls erforderlich sollte z.B. für „C“ neben C11 auch der veraltete C99 Standard verstanden werden.

Auf welchen Betriebssystemen kann das Werkzeug installiert werden?

Neben der Frage ob das Werkzeug auch die in Ihrem Projekt eingesetzten Betriebssysteme unterstützt, gilt es darüber hinaus zu überprüfen, ob es für die von Ihnen verwendeten Betriebssystemversion auch ausgewiesen ist. Wird für die zu entwickelnde Software eine Zertifizierung angestrebt, kann dieser Punkt entscheidend sein.

Welches Lizenzmodell entspricht meinen Anforderungen?

Vertreiber von Anwendungssoftware sind kreativ in der Erfindung neuer Lizenzmodelle. Exemplarisch erwähnt werden können daher hier nur gängige „Basismodelle“. So gibt es maschinengebundene

und personengebundene Lizenzen. Ist geplant, die Analyse automatisiert, zentral auf einem Build-Server ablaufen zu lassen, so reicht es vielleicht aus, allein die Build-Maschine zu lizenzieren. Möchte dagegen jeder Entwickler selbst Analysen durchführen, bieten sich eher personengebundene Lizenzen an.
Vielfach gehören auch so genannte „Floating-Lizenzen“ zum Portfolio des Anbieters. Hier ist nicht die Anzahl der Installationen beschränkt, wohl aber die Anzahl der Anwender, die zeitgleich mit dem Werkzeug arbeiten. Ein solches Modell bietet größere Flexibilität bei einer sich dynamisch ändernden Anzahl von (nicht namentlich registrierten) Nutzern.
Ein wichtiges Kriterium ist auch die Laufzeit der Lizenzen. Einige sind unbegrenzt gültig (perpetual licenses), andere dagegen zeitlich befristet.
Wieder andere Lizenzmodelle binden die Lizenzen an die Anzahl der zu analysierenden Quellcodezeilen. Jeder Analyselauf erhöht einen Zähler um die Zahl der neu analysierten Codezeilen. Wird eine vorgegebene Zeilenzahl überschritten, müssen Kontingente nachgekauft, oder neue Lizenzen erworben werden.
Viele Lizenzen beinhalten bereits einen Wartungsvertrag oder der Toolhersteller bietet einen solchen gesondert an. Unter der Bedingung, dass Preis und Leistung dem vertraglich vereinbarten Umfang entsprechen, zahlt sich der Abschluss eines Supportvertrages meist aus. Codeanalysetools sind sehr komplexe Werkzeuge, die um ihre volle Leistungsfähigkeit entfalten zu können, im Hinblick auf ihre Einsatzumgebung optimal konfiguriert werden müssen. Die Unterstützung durch die Supportabteilung des Herstellers ist dabei auch durch eine gute Dokumentation nicht zu ersetzen. Zudem sind sowohl die Compiler als auch die Programmiersprachen einem Wandel unterworfen. Die im Supportvertrag enthaltenen Updates sind daher notwendig, um die Einsatzfähigkeit des Produktes zu erhalten.

Welche Compiler/Crosscompiler werden unterstützt?

Compiler? Zunächst stellt sich die Frage, warum für ein Werkzeug zur statischen Codeanalyse die Kenntnis über den Compiler überhaupt relevant ist? Das hat gleich mehrere Gründe.
Für eine tiefgreifende Analyse benötigt ein statisches Analysewerkzeug alle Pfadinformationen des zu prüfenden Programmes, um die Beziehungen einzelner Applikationsmodule untereinander auflösen zu können. Nur so ist eine interprozedurale Analyse überhaupt möglich. Einige Tools erhalten diese und weitere Informationen vom Compiler selbst durch eine entsprechende Abfrage beim Buildprozess. Solche Abfragen sind natürlich compilerspezifisch. Darüber hinaus weichen insbesondere im Embedded Bereich einige Compiler vom Standard ab indem z.B. proprietäre Spracherweiterungen eingeführt werden. Ein statisches Analysewerkzeug muss solche Eigenheiten kennen, um diese nicht als Fehler zu behandeln.
Es ist selbstverständlich, dass ein Analysewerkzeug nur die (nach Meinung des Toolherstellers) gebräuchlichsten Compiler unterstützt. Exoten werden meist nicht berücksichtigt. Gerade deshalb sollte es auch durch den Anwender weitgehend unkompliziert möglich sein, neue Compilermodelle zu erstellen und einzubinden.

Kann das Werkzeug für Projekte der gewünschten Größe eingesetzt werden?

Zur statischen Analyse von Quellcode wird dieser durch einen Parser strukturiert eingelesen und anschließend bewertet. Es ist einsichtig, dass die zu verarbeitende Datenmenge mit der Anzahl der Programmzeilen (loc = **l**ines **o**f **c**ode) überproportional ansteigt. Von der Effizienz, mit der das

Analyseprogramm diese Datenmenge bei einem vertretbaren Memoryfootprint und vertretbarer Verarbeitungszeit noch bewältigen kann, hängt letztendlich die maximal noch zu verarbeitende Quellcodegröße ab. Für qualitativ hochwertige Tools zur statischen Codeanalyse sind einige Millionen Codezeilen kein Problem.

Welche Zeit wird für einen Analyselauf benötigt?

Die Dauer eines Analyselaufes hängt von einer Vielzahl von Parametern ab. Zunächst einmal ist der Umfang des zu analysierenden Quellcodes relevant. Darüber hinaus spielt sicherlich auch der dem Werkzeug gestellte Aufgabenumfang eine Rolle. Werden zusätzliche Prüfungen erforderlich (z.B. MISRA-Rules-Check) wird damit auch die Laufzeit verlängert.
Ein besonders wichtiges Kriterium ist, inwieweit das Tool auf modernen Multi-Core-Maschinen skaliert. Ist das Produkt nebenläufig programmiert, kann sich die Analysedauer auf leistungsstarker Hardware deutlich reduzieren.
Einige Analysetools besitzen die Möglichkeit zur inkrementellen Analyse. Damit werden bei jedem neuen Analyselauf, unter Berücksichtigung bestehender Informationen, nur die neu hinzugekommenen Codezeilen überprüft, was die Verarbeitungsdauer erheblich herabsetzt.
Generell ist zu bemerken, dass der Analysedauer je nach Art der Eingliederung in den Entwicklungsprozess eine unterschiedliche Bedeutung zukommt. Läuft die Analyse auf einem Build-Server parallel zum „Nightly Build“, ist die Laufzeit in der Regel unkritisch, solange die Ergebnisse am nächsten Morgen vorliegen. Ist allerdings vorgesehen, dass jeder Entwickler das Werkzeug parallel zur Überprüfung seiner Implementierung nutzt, hängt die Akzeptanz des Tools sicherlich stark von dessen Arbeitsgeschwindigkeit ab.

Lässt sich das Werkzeug in meine Build/Entwicklungsumgebung integrieren?

Sind die wichtigsten Funktionen des statischen Analysetools auch von der Kommandozeile zu bedienen, ist die Wahrscheinlichkeit groß, dass eine Integration in Ihre Build- und Entwicklungsumgebung möglich ist. Die meisten Hersteller bieten eine Anbindung an diverse IDEs wie z.B. Eclipse sowie Integrationstools wie Hudson oder Jenkins an.

Ist das Analysetool zertifiziert?

Der Wert eines Zertifikates für ein Analysewerkzeug bietet Raum für Diskussionen. Ein Zertifikat wird stets für die spezielle Umgebung (Betriebssystem, Compiler, Hardware etc.) erteilt, unter der der Prüfer seine Tests vorgenommen hat. Ihre Umgebung weicht in der Regel davon ab. Sollten Sie für die von Ihnen entwickelte Software eine Zertifizierung anstreben, so müssen Ihre Test- und Analysewerkzeuge im Hinblick auf Ihre Umgebung qualifiziert werden.
Durch eine bereits bestehende Zertifizierung des Werkzeugs, wenn diese auch für eine abweichende Umgebung erfolgte, kann zumindest mit großer Wahrscheinlichkeit davon ausgegangen werden, dass auch für den Einsatz innerhalb von Ihrer Toolchain eine Qualifizierung möglich ist.

Überprüft das Tool die Einhaltung von Coding Standards (MISRA etc.)?

Viele statische Analysetools führen bei Bedarf auch so genannte „Coding Rules Checks“ durch. Hierbei wird der Quellcode auf die Einhaltung von Coding Standards hin überprüft. Coding Standards erhöhen die Softwarequalität durch Verbesserung der Les- und Wartbarkeit.

Einige Qualitätsnormen, wie ISO 26262, EN 50128 und EN 61508 empfehlen dringend deren Einhaltung oder setzen sie sogar voraus. Als eine Auswahl weithin gebräuchlicher Standards sind hier zu nennen:

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)

CERT Secure Coding Standard (von der CERT Coding Initiative sowie Mitgliedern der 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 bei anderen Unternehmen im Bereich Medizintechnik eingeführt)

Im Hinblick auf eine angestrebte Zertifizierung der von Ihnen entwickelten Software lohnt es sich beim Hersteller zu erfragen, inwieweit das statische Analysewerkzeug einen gewünschten Coding Standard unterstützt. Hierbei gilt es zu beachten, dass vielfach nicht alle Coding Rules eines Standards durch das Mittel der statischen Analyse überhaupt überprüfbar sind. Einige Hersteller treffen dann oft optimistische Annahmen anstatt klar zu kommunizieren, dass eine bestimmte Regel übergangen wurde.
Sollten Sie anstreben, eigene Codierregeln aufzustellen, bzw. Regeln eines bestehenden Standards abgeändert anzuwenden, sollte abgeklärt werden, ob eine leichtverständliche Schnittstelle dafür bereitsteht.

Werden Metriken generiert?

Um die Qualität von Sourcecode zu beurteilen werden häufig Metriken herangezogen. Es ist daher naheliegend, dass statische Analysewerkzeuge beim Analyselauf das Erheben solcher Metriken gleich mit erledigen.
Von Interesse können Metriken sein wie:

LOC (Lines Of Code): Anzahl der Codezeilen

Watson & McCabe: Zyklomatische Komplexität

Halstead: Implementierungsabschätzung durch abgeleitete Kenngrößen für Lesbarkeit,

Implementierungsaufwand und Implementierungszeit

Besteht die Notwendigkeit eigene Metriken zu definieren, ist auch hier abzuklären, inwieweit entsprechende Algorithmen eingebunden werden können.

Welche Fehler vermag das Werkzeug aufzudecken?

Wie viele verschiedene Fehler ein Werkzeug zur statischen Codeanalyse auffinden kann, ohne zu viele Falschmeldungen („False Positives“) zu generieren, ist natürlich der Qualitätsmaßstab schlechthin. Hier hilft es sich im Rahmen einer Evaluierung ein Bild durch Analyseläufe kleiner Testprogramme zu verschaffen.

Speicherüberläufe

Speicherüberläufe bei statisch allokiertem Speicher sollte ein statisches Analysewerkzeug souverän erkennen können. Hier ein kleines C-Programm um dies zu überprüfen:

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

Um die Hürde etwas höher zu legen, wird bei folgendem Test der Speicherüberlauf etwas verschleiert [1]:

```
void main(void)
{
	char buffer[10];
	char* pc;
	int j = 0;

	pc = buffer;
	for (int i = 0; i <= 10; i++)
		*pc++ = 'c';

	for (int i = 0; i < 6; i++)
		buffer[j++] = 'd';
}
```

Auch diese Fehler sollte ein gutes Werkzeug aufdecken können.

Datentypüberlauf

Zu den Speicherüberläufen gehören auch mögliche Datentypüberläufe. Hier ein einfaches Beispiel für die Gefahr eines Datentypüberlaufes, vor der allerdings nicht alle Analysetools warnen:

```
#include <stdio.h>


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

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

Shift über Typgrenzen hinaus

Operationen auf Bit-Ebene bergen die Gefahr, dass diese über die Typgrenzen hinaus wirken und damit zu Datenverlust führen.
Ein gutes Analysewerkzeug sollte vor solchen oder ähnlichen Fehlern warnen:

```
void main(void)
{
	int x = 42;
	int y = x << 167;
}
```

Datenverlust durch implizite Typumwandlung

Datenverluste durch Typumwandlung gehören zu den häufigsten Fehlerquellen in Applikationen. Die meisten statischen Analysetools warnen daher auch zuverlässig vor möglichen Datenverlusten bedingt durch Fehler dieser Art. Die beiden folgenden, einfachen Beispiele, sollten dazu geeignet sein, um durch eine statische Analyse erkannt und als kritisch eingestuft zu werden.

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

Datenverlust durch explizite Typumwandlung

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

Nichtinitialisierte Variable [1]

Nichtinitialisierte Variable führen oft zu Fehlern, die nur schwer auffindbar sind. Belegt mit einem zufälligen Wert, ist das Verhalten der Applikation zur Laufzeit unbestimmt. Zum Auffinden derartiger

Fehler ist das Verfahren der statischen Codeanalyse meist besser geeignet als dynamische Analyseverfahren.

```
int somefunction(int x){
	int y;
	if (x >= 42){
		y = 1;
	}
	return y;
}
```

Nicht erreichbare Codeabschnitte (“Dead Code”) [1]

Das Erkennen von nicht erreichbaren Codeabschnitten ist nicht unwichtig. Toter Code belegt unnötig Speicherplatz und erschwert die Wartbarkeit einer Applikation. Darüber hinaus werden Metriken verfälscht.

```
int main(void)
{
	int x = 1;
	int y = 12;

	if (x)
	{
		if (y > 10 && !x)
		{
			x++; /* dead code */
			return 1;
		}
	}
	return 0;
}
```

Divison durch null

Divisionen durch null können zu Programmabstürzen führen. Ein Analyseprogramm sollte daher zuverlässig warnen, wenn die Möglichkeit, dass der Divisor den Wert null annehmen kann nicht unterbunden wird.

```
void main(void)

{

	int x = 0;

	int y = 27;

	int z = y / x;

}
```

Division durch null unter Verwendung von Fließkommavariablen

```
void main(void)

{

	float x = 0.;

	float y = 27.;

	float z = y / x;

}
```

Dieser Fall ist eine Besonderheit, die von verschiedenen Tools unterschiedlich bewertet wird. Grundsätzlich besitzt eine Fließkommazahl (float, double) eine Genauigkeit, die durch ihre Repräsentation als Maschinenzahl begrenzt ist. Es bleibt für ein Analysewerkzeug also spekulativ, ob Divisionen wie im obigen Beispiel dargestellt als Fehler zu werten sind oder als legale Divisionen durch einen infinitesimal kleinen Wert.
Es werden daher von Werkzeug zu Werkzeug entweder eher optimistische oder eher pessimistische Annahmen getroffen. So kann es sein, dass bedingt durch einen optimistischen Ansatz, an dieser Stelle keine Warnung erfolgt (False Negative). Vielfach werden im Rahmen der Analyse „float“ und „double“ Datentypen intern durch „int“-Typen ersetzt. Das führt sicher zu Warnungen vor Fällen gleich dem vorliegenden, erhöht damit einhergehend aber auch die Zahl von Fehlwarnungen (False Positives).
Einige Analysetools ermöglichen es ihr Verhalten hinsichtlich von Fließkommavariablen durch Konfiguration zu verändern. Handbuch oder der Hersteller können darüber Auskunft geben.

Kein Rückgabewert

Ein Fehler, der zu unerwartetem Laufzeitverhalten führen kann, weist die unten stehende Funktion auf. Da Fehler dieser Art häufig anzutreffen sind, wurde dieses Beispiel mit in die Reihe der Testfälle aufgenommen.

```
int noretval(int j) {

	if (j == 1) return;

	else return 1;

}
```

Memory Leak [1]

Die Verwendung von dynamischem Speicher ermöglicht einen besonders effizienten Umgang mit den Ressourcen. Für die Verwaltung von Speicher auf dem Heap hat der Programmierer allerdings selbst zu sorgen. Dangling Pointers, Memory Leaks und Zugriffe auf bereits freigegebene Speicherblöcke

sind die damit verbundenen typischen Fehler.
Speicherlecks werden oft erst nach längerer Programmlaufzeit auffällig und daher oft zu spät erkannt. Ein statisches Analysewerkzeug kann hier seine Stärken voll ausspielen.

```
int simple_leak(void)
{
	char* p = (char*)malloc(12);
	if (!p)
		return 0;
	if (!some_function())
	{
		free(p);
		return-1;
	}
	if (!some_other_function())
		return-2; /* Memory Leak */
	free(p);
	return1;
}
```

Zugriff auf bereits freigegebenen Speicher [1]

Zum Bereich der dynamischen Speicherverwaltung zählen auch Programmfehler, die durch Zugriffe auf bereits zuvor freigegebenen Speicher bedingt sind. Solche Fehler führen oft zu Programmabstürzen, deren Ursache dann meist aufwendig mit dynamischen Analyseverfahren ermittelt werden muss (Debugger). Statische Analyseverfahren können dazu beitragen, derartige Aufwände deutlich zu reduzieren.

```
void simple_use_after_free(void)
{
	char* pc2;
	char* pc3;
	pc2 = (char*)malloc(10 * sizeof(char));
	if (pc2)
	{
		pc3 = pc2;
		free(pc2);
		pc3[0] = ´b´;
	}
}
```

Nebenläufigkeitsfehler, hier Data Race

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>


pthread_mutex_t mutex0 = PTHREAD_MUTEX_INITIALIZER;

double sum = 0.;



void *calculate(void *params){
        double d, w;
        unsigned long l;

        unsigned long* part = (unsigned long*) params;
        d = 1.0 / part[2];
        for (l = part[0]; l < part[1]; ++l){
                w = d * (l + 0.5);
/*              pthread_mutex_lock(&mutex0); */
                        sum += 4.0 / (1.0 + w * w);
/*              pthread_mutex_unlock(&mutex0); */
        }
}



void main(void){

        pthread_t pth1, pth2;
        unsigned long segment_info0[3] = {0, 5000000, 10000000};
        unsigned long segment_info1[3] = {5000001, 10000000, 10000000};

        if (pthread_create(&pth1, NULL, calculate, (void*) &segment_info0) != 0){
                printf("Thread creation failed!\n");
                exit (EXIT_FAILURE);
        }

        if (pthread_create(&pth2, NULL, calculate, (void*) &segment_info1) != 0){
                printf("Thread creation failed!\n");
                exit (EXIT_FAILURE);
        }

        pthread_join(pth1, NULL);
        pthread_join(pth2, NULL);

        printf("PI = %25.20lf\n", sum / segment_info0[2]);
}
```

Von sehr großer Bedeutung ist die Möglichkeit zur Aufdeckung von Nebenläufigkeitsfehlern. Das obige Beispiel führt zu einer „Race Condition“, hier Data Race. Race Conditions sind oft nicht so einfach zu erkennen wie in diesem Beispiel. Sie können zu sporadisch auftretendem, fehlerhaftem Laufzeitverhalten führen, was das Triggern eines solchen Ereignisses erschwert.
Ein Werkzeug zur statischen Codeanalyse, das derartige Fehler zu detektieren vermag, kann das

Kostenrisiko eines Projektes, das Nebenläufigkeit implementiert, erheblich reduzieren.

Eine Voraussetzung für das Auffinden von Nebenläufigkeitsproblemen ist das Ermitteln der „Threadeinstiegspunkte“. Diese können in Objekten verborgen sein. Oftmals werden auch eigene Threading Libraries verwendet. Daher besteht die Notwendigkeit, neue Threadeinstiegspunkte zu spezifizieren und diese in das Tool einpflegen zu können.

Bitte beachten Sie, dass viele Hersteller die Funktionalität zur Analyse von Nebenläufigkeitsfehlern in der Grundeinstellung abgeschaltet haben. Im Hinblick auf Untersuchungen von Single-threaded Applikationen kosten die entsprechenden Checks nur unnötig Zeit.
Einige Tools setzen eine Instrumentierung des Quellcodes voraus.
Das Handbuch oder im Zweifel der Hersteller geben Auskunft, ob und wie die Prüfungen auf Nebenläufigkeitsfehler aktiviert werden können.

Nebenläufigkeitsfehler, hier Dead Lock

```
void *calculate1(void *params){
	double d, w;
	unsigned long l;

	unsigned long* part = (unsigned long*)params;
	d = 1.0 / part[2];
	for (l = part[0]; l < part[1]; ++l){
		w = d * (l + 0.5);
		pthread_mutex_lock(&mutex0);
		pthread_mutex_lock(&mutex1);
		sum += 4.0 / (1.0 + w * w);
		pthread_mutex_unlock(&mutex1);
		pthread_mutex_unlock(&mutex0);
	}
}


void *calculate2(void *params){
	double d, w;
	unsigned long l;

	unsigned long* part = (unsigned long*)params;
	d = 1.0 / part[2];
	for (l = part[0]; l < part[1]; ++l){
		w = d * (l + 0.5);
		pthread_mutex_lock(&mutex1);
		pthread_mutex_lock(&mutex0);
		sum += 4.0 / (1.0 + w * w);
		pthread_mutex_unlock(&mutex0);
		pthread_mutex_unlock(&mutex1);
	}
}
```

Wird das „Data Race“-Beispiel in oben dargestellter Weise abgeändert, kann dies zu einem möglichen Deadlock führen. Jeder Thread erhält nun seine eigene Funktion. Für den Produktivbetrieb wenig sinnvoll, ist dieses Beispielprogramm dennoch nützlich um die Fähigkeit zum Erkennen von Deadlocks verschiedener statischer Analysetools zu testen.
Auch hier gilt die Forderung, weitere Locks spezifizieren und unkompliziert dem Werkzeug mitteilen zu können.

Ist das Werkzeug in der Lage auch interprozedurale Analysen durchzuführen?

Viele der aufgezeigten Fehler können sowohl intraprozedural als auch interprozedural auftreten. Einige Werkzeuge zeigen erhebliche Schwächen, sobald sich ein Fehler über Modul- oder Funktionsgrenzen hinaus erstreckt. Im Folgenden sind zwei Beispielprogramme wiedergegeben, anhand derer die Fähigkeit zur interprozeduralen Analyse überprüft werden kann.

Doppelte Speicherfreigabe [1]

Modul 1:

```
void test_driver(void)
{
	int* pi1 = (int*)malloc(sizeof(int));

	if (pi1)
		some_function(pi1);

	if (pi1)
		free(pi1);
}
```

Modul 2:

```
void some_function(int* p)
{
	if (p)
		free(p);
}
```

Versteht ein Werkzeug zur statischen Codeanalyse Abhängigkeiten auch interprozedural aufzulösen, sollte es die doppelte Speicherfreigabe erkennen können.

Hier ein weiteres Beispiel:

Memory Leak [1]

Modul 1:

```
void afunction(void)
{
	int* pi1 = NULL;
	pi1 = (int*)malloc(sizeof(int));
	test_free(pi1, 20);
}
```

Modul 2:

```
void test_free(int* p, int x)
{
	if (p && x < 10)
		free(p);
}
```

Das kleine Beispiel für ein Speicherleck ist ebenso dafür geeignet, die Fähigkeit zur interprozeduralen Analyse zu überprüfen.

Diese Sammlung von Testbeispielen erhebt sicher keinen Anspruch auf Vollständigkeit. Sie soll vielmehr dazu anhalten weitere, eigene Testfälle zu entwickeln.

Wer soll Zugriff auf die Analyseergebnisse erhalten?

Die Beantwortung dieser Frage ist ein weiteres, wichtiges Kriterium für die Auswahl des Analysewerkzeuges. Sollen die Analyseergebnisse nicht allein von der Person, die die Analyse durchführt eingesehen werden können, sollten diese an zentraler Stelle hinterlegt und allgemein zugänglich sein. Besteht allerdings die Forderung, die Zugriffe auf autorisierte Benutzer einzuschränken, wird eine Zugriffskontrolle nötig. Um mehreren Benutzern zugleich einen geregelten Zugriff auf die Ergebnisse zu ermöglichen, nutzen einige Tools Datenbanken. Dadurch können einerseits die Ergebnisse persistiert und andererseits die Zugriffsmechanismen der Datenbank (Session Handling) genutzt werden. Unbedingt abzuklären ist, wie eine Datensicherung durchgeführt werden kann.

Ist die Dokumentation vollständig und leicht verständlich geschrieben?

Tools zur statischen Codeanalyse sind naturgemäß hochkomplexe, erklärungsbedürftige Applikationen. Ihr effizienter Einsatz ist nur mit Hilfe einer stets aktuellen, verständlich geschriebenen, gut strukturierten und umfassenden Produktdokumentation möglich. Bitte planen Sie im Rahmen Ihrer Evaluation Zeit ein, um sich einen Eindruck von der Qualität des mitgelieferten Benutzerhandbuches zu verschaffen.

Ist das Werkzeug bedienerfreundlich?

Abschließend soll noch auf die Handhabung eingegangen werden. Dazu zählt die unkomplizierte Installation genauso wie eine schnelle und weitgehend intuitive Bedienbarkeit. Letztere spart Kosten einer zeitintensiven Einarbeitung.
Im Hinblick auf die Installation ist insbesondere darauf zu achten, wie der Prozess des Einspielens von Updates geregelt ist. Es sollte gewährleistet sein, dass bestehende Analyseergebnisse dabei nicht zerstört werden können.
Von Wichtigkeit ist auch, wie die Analyseergebnisse präsentiert werden. Einige Tools weisen, eher spartanisch, lediglich die Bezeichnung der Warnung/des Fehlers zusammen mit der Zeilennummer auf der Konsole aus. Andere Werkzeuge dagegen markieren die problematischen Bereiche im Quellcode und fügen aussagekräftige Erläuterungen hinzu.
In der Praxis hat es sich als vorteilhaft erwiesen, Fehler und Warnungen kategorisiert auflisten, sowie Prioritäten zuweisen zu können. Beim Einsatz in großen Projekten sind frei konfigurierbare Ergebnisfilter, die selektiv Warnungen und Fehler ein- bzw. auszublenden vermögen, eine große Hilfe.
Zur Ergebnispräsentation gehört auch die Generierung von Reports. Diese sind für das Management notwendig, um über den Projektstatus umfassend informiert zu sein. Form und Inhalt der Reports sollte möglichst an die jeweiligen Erfordernisse anpassbar sein.

Schlussbemerkung

Die vorliegende Abhandlung erhebt nicht den Anspruch Qualitätsrichtlinien für Werkzeuge zur statischen Codeanalyse aufzustellen. Die auf dem Markt erhältlichen Tools unterscheiden sich stark in Bezug auf Preis, Leistung und Funktionalität. Die Auswahlkriterien, die hier angesprochen wurden, sind lediglich als Hilfestellung zu verstehen, um den Bedarf zu ermitteln und diesen mit Funktions- und Leistungsumfang verschiedener Produkte im Rahmen einer Evaluation abgleichen zu können.

Checkliste

- ☑ **Im Projekt verwendete Programmiersprache(n) wird / werden unterstützt**
- ☑ **Alle notwendigen Betriebssysteme werden unterstützt**
- ☑ **Alle im Projekt verwendeten Compiler werden unterstützt**
- ☑ **Das Werkzeug zeigt eine gute Leistung bei der Fehleraufdeckung und das Verhältnis True Positives / False Positives ist gut**
- ☑ **Das Tool ist performant und skaliert auf Multicore-Maschinen**
- ☑ **Das Werkzeug kann die geforderten Metriken generieren**
- ☑ **Das Analysetool prüft auch auf Einhaltung von Coding Rules**
- ☑ **Integration in die vorhandene Entwicklungsumgebung ist möglich**
- ☑ **Gewünschte Zugriffsregelung / Autorisierung lässt sich umsetzen**
- ☑ **Lizenzmodell und Lizenzpreis passen**
- ☑ **Der Hersteller liefert schnellen und qualitativ guten Support**
- ☑ **Das Werkzeug eignet sich für die Projektgröße**
- ☑ **Die Dokumentation ist vollständig und leicht verständlich**
- ☑ **Die Handhabung ist weitgehend intuitiv**
- ☑ **Es wird nur eine kurze Einarbeitungszeit benötigt**
- ☑ **Das Werkzeug ist zertifiziert, eine Qualifizierung für meine Toolchain ist möglich**

Über den Autor:
Dipl.-Ing. (FH) Royd Lüdtke ist als Director Static Analysis Tools für die Verifysoft Technology GmbH tätig.

Quellenverzeichnis

[1] Codebeispiel nach „Bug Hunting mit statischer Codeanalyse“, Embedded Software Engineering Kongress 2013, Prof. Dr. Daniel Fischer, Andreas Behr, Roland Bär