Integrierte Entwicklung zuverlässiger Software

Dipl.-Inf. Oliver Schneider, Dr.-Ing. Hubert B. Keller

May 12, 2016

Abstract

Modellierungstools und automatische Codegenerierung aus Modellen erlauben es Entwicklern Software top-down zu designen. Modellierung sollte jedoch nicht zur Programmierung jedes Details missbraucht werden. Der algorithmische Anteil eines Projekts kann mit händisch programmiertem Code oft verständlicher, kompakter und effizienter implementiert werden. Um dabei Fehler zu vermeiden, werden statische Analysen eingesetzt. Statische Analysetools werden jedoch vergleichsweise langsam weiterentwickelt, weshalb nur langfristig aus wiederkehrenden Fehlern gelernt wird. Moderne Compiler ermöglichen mit wenig Aufwand neue statische Analysen in den Compiliervorgang zu integrieren. Zudem entstehen Datenbanken von Analysen in der Community, die ohne Aufwand eingebunden werden können. Die existierenden Analysen aus der Community reichen von einfachen Analysen wie der Einhaltung von Namensgebungrichtlinien bis zu aufwändigen Analysen, die verlustbehaftete Gleitkommaberechnungen erkennen oder UML-Modelle bei jedem Compiliervorgang gegen den Code prüfen. In diesem Paper wird ein Softwareentwicklungszyklus vorgestellt, der diese transparente und flexible Verwaltung von statischen Analysen in vollem Umfang nutzt.

1 Modellierung vs. Programmierung

Model-Driven-Development wird erfolgreich in großen Softwareprojekten eingesetzt. Wegen der Komplexität von Modellierungssprachen wie UML, wird nur eine projektspezifische Untermenge eingesetzt [1] und dann über eine Modell-zu-Code Transformationsspezifikation in Programmcode übersetzt. Bei der Entwicklung der Prozesssteuerungssoftware INSPECT [2] wurden sowohl statische als auch dynamische Aspekte modelliert. Trotz des umfassenden Einsatzes von Modellierung war handgeschriebener Code nicht zu vermeiden. Dieser wurde zwar mit Ada-Restriktionen weiter eingeschränkt, jedoch konnten keine projekteigenen Restriktionen automatisiert geprüft werden.

2 Existierende Schnittstellen

2.1 Open Source Statische-Analyse-Tools

Statische-Analyse-Tools wurden mit `Lint` [3] populär. Wenn Compiler nicht strikt genug sind, werden, besonders bei Open-Source-Projekten, Tools zur statischen Analyse entworfen, bei der jeder Entwickler mit wenig Aufwand neue Analysen selbst entwickeln kann. Am meisten profitierten hiervon bisher dynamische Sprachen wie `Ruby` [4], `Javascript` [5] und `Haskell` [6], da deren Compiler meist viele Konstrukte erlauben, die entweder von der großen Mehrheit der Programmierer als "unschön" eingestuft werden oder nicht das bedeuten, was intuitiv als die Bedeutung angenommen wird.

2.2 Compilerunterstützung

Ein wichtiger Faktor bei der Nutzung von statischen Analysen ist der nötige Aufwand der betrieben werden muss, um die Analysen erstmalig durchzuführen. Damit die statischen Analysen auch regelmäßig durchgeführt werden, muss es möglich sein, diese automatisiert auszuführen. Pluginschnittstellen von Compilern (`Rust` [7], `C/C++` (gcc) [8], `C/C++` (clang) [9], `scala` [10], `haskell` [11]) bieten die Möglichkeit direkt die Fehlerberichterstattung des Compilers zu nutzen, sowie auf compilerinternes Wissen wie z.B. Typinformationen oder Kontrollflüsse zuzugreifen. Gegenüber klassischen externen Analysetools hat dies den Vorteil, dass keine Codeduplikation (Parser, Typsystem, etc.) stattfindet. Wenn der Compiler jedoch keine Stabilitätsgarantien für die Pluginschnittstelle gibt, kann es vorkommen, dass bei Updates des Compilers auch die Plugins angepasst werden müssen. Keiner der vorgestellten Compiler gibt hierzu Garantien, wobei der `Rust` Compiler zumindest einen Ausblick für Aufwärtskompatibilität bei Syntaxplugins hat.

3 Beheben von Fehlerklassen statt von Fehlern

Oft gibt es fehlerhafte Codekonstrukte, die regelmäßig auftreten. Es ist daher wünschenswert, diese Codekonstrukte generell zu verbieten, statt im Einzelfall zu beheben. Wenn in dem Projekt bereits ein Open-Source Tool für statische Analysen verwendet wird, kann dieses Tool um neue Analysealgorithmen erweitert werden. Dabei gibt es jedoch mehrere Problematiken. Die Entwickler des Tools haben nicht unbedingt die Ansicht, dass das Codekonstrukt generell als fehlerbehaftet zu bezeichnen ist. Außerdem ist es viel Aufwand, sich in eine neue Codebasis einzuarbeiten, vor allem wenn das Tool in einer anderen Sprache oder

nach völlig anderen Codingstandards geschrieben ist.

3.1 Statische Analyse als Compilerplugin

Es bietet sich an, die Pluginschnittstelle des Compilers zu nutzen, um mit minimalem Aufwand den Code als Datenstruktur analysieren zu können. Ein weiterer Vorteil dieses Ansatzes ist, dass die Fehlermeldungen über den Compiler ausgegeben werden, die dann durch Entwicklungsumgebungen und Continuous Integration bereits verarbeitet werden können und in einem dem Entwickler bereits bekannten Format sind.

3.2 Beispiel

Ein Projekt kann z.B. Variablen- und Typnamen mit mehr als 20 Zeichen verbieten. Am Beispiel der Programmiersprache `Rust` wird der komplette Entwicklungszyklus einer statischen Analyse gezeigt. Compilerplugins in `Rust` werden wie eine Bibliothek entworfen. Die Projektdatei würde hierbei folgendermaßen aussehen:

```
[package]
name = "lange_namen"
version = "1.0.0"
authors = [
    "Hubert Keller <hubert.keller@kit.edu>",
   "Oliver Schneider <oliver.schneider@kit.edu>"
]

[lib]
plugin = true
```

Der einzige Unterschied zu einer normalen Bibliothek ist das "`plugin = true`", das den Paketmanager informiert, dass diese Bibliothek als Plugin genutzt wird und daher nicht statisch gelinkt werden darf.

Der Programmcode der statischen Analyse muss nun nur nach Namen mit mehr als 20 Zeichen suchen. Zuerst muss der `rust` Compiler als Bibliothek geladen werden, diverse Teile des Compilers importiert werden und dem Compiler gemeldet werden, dass ein neuer Lint verfügbar ist.

```
#![feature(plugin_registrar, rustc_private)]
#[macro_use] extern crate rustc;
extern crate rustc_plugin;
extern crate syntax;

use rustc::lint::*;
use rustc_plugin::Registry;
use syntax::ast;
use syntax::codemap::Span;

#[plugin_registrar]
pub fn plugin_registrar(reg: &mut Registry) {
    reg.register_early_lint_pass(Box::new(Pass));
}
```

Nun wird der Lint mit Namen, Kritikalität und einer kurzen Beschreibung eingeführt. Mögliche Kritikalitäten sind `Deny` (ein Fehler), `Warn` (eine Warnung) und `Allow` (entscheidung über Kritikalität trifft der zu prüfende Code).

```
declare_lint!(
    LANGE_NAMEN, // Name des lints
    Deny, // Fehlermeldung statt Warnung
    "Verbietet Namen mit mehr als 20 Zeichen",
);
```

Da unser Lint weder Zustandsbehaftet ist, noch Konfigurationen oder Modelle verwendet, enthält die Lintdatenstruktur `Pass` auch keine Felder.

```
struct Pass;
```

Ein Lint kann möglicherweise unterschliedliche Fehler melden, in unserem Fall wird jedoch nur eine Art von Fehler gemeldet. Die Funktion `get_lints`, welche eine Liste der Lints zurückgibt, die von `Pass` gemeldet werden dürfen, gibt daher nur den `LANGE_NAMEN` lint zurück.

```
impl LintPass for Pass {
    fn get_lints(&self) -> LintArray {
        lint_array!(LANGE_NAMEN)
    }
}
```

Um nun tatsächlich auch eine Überprüfung durchzuführen wird der Trait `EarlyLintPass` implementiert. Traits sind allgemein auch als Interface-Typ oder abstrakte Basisklasse bekannt. `EarlyLintPass` ist nach dem Designpattern Visitor aufgebaut, weshalb nur die Funktion `check_ident` überladen werden muss, die auf Namen zugreift. Das geerbte Verhalten von `EarlyLintPass` sorgt dafür, dass alle Sprachkonstrukte die Namen haben, auch die funktion `check_ident` aufrufen.

```
impl EarlyLintPass for Pass {
    fn check_ident(&mut self, cx: &EarlyContext,
                   span: Span, ident: ast::Ident) {
        if ident.name.as_str().len() > 20 {
            cx.span_lint(LANGE_NAMEN, span,
                         "Name mit mehr als 20 Zeichen");
        }
    }
}
```

Um das Plugin auch zu nutzen benötigt es nur zwei Zeilen:

```
#![feature(plugin)]
#![plugin(lange_namen)]
```

Falls nun im Projekt ein zu langer Name vorkommt, z.B.

```
fn dies_ist_ein_ziemlich_langer_funktionsname() { }
```

gibt der `rust` Compiler automatisch eine Fehlermeldung im normalen Fehlerformat des Compilers aus. Der einzige visuelle Unterschied zu den Fehlern, die der Compiler normalerweise ausgibt, ist eine Nennung des Lintnamens `lange_namen`.

3.3 Statische Analysen aus der Community

Wenn die selbstentwickelten statischen Analysen der Community zur Verfügung gestellt werden, entwickelt

sich mit der Zeit eine Datenbank von statischen Analysen. Diese kann direkt im eigenen Code eingesetzt werden, um auch Codekonstrukte zu erkennen, die im eigenen Projekt noch nicht als fehlerbehaftet eingestuft wurden. Die lint-Sammlung `clippy` [12] für den `rust` Compiler ist so entstanden und erhält regelmäßig neue statische Analysen, die von Syntaxanalysen über Datenflussanalysen bis zu Kontrollflussanalysen (z.B. zyklomatische Komplexität) reichen. Facebook hat für `clang` ein Open Source Projekt [13] gestartet mit diversen Syntaxanalysen.

4 Praxistauglicher Einsatz statischer Analysen

4.1 Einsatzgebiete

Style

Projektspezifische Stylevorschriften werden eingesetzt, damit jedes Teammitglied den Code anderer Entwickler schneller verstehen kann, da die syntaktischen Strukturen denen ähneln, die der Entwickler selbst auch einsetzt. Das Einhalten der Vorschriften kann jedoch für neue Teammitglieder mit viel Aufwand verbunden sein und Kleinigkeiten werden auch von den Erfahrenen oft vergessen. Automatische Codeformatierungstools helfen zwar bei der Automatisierung der Formatierung, können aber in Anwesenheit von mehreren Lösungen nicht immer die richtige Entscheidung treffen. Ein Syntaxanalyzer hingegen meldet dem Entwickler den Verstoß und kann mehrere Möglichkeiten zur Behebung anbieten

Da für Syntaxanalysen nur die aktuelle Datei zu betrachten ist, können diese dem Entwickler während der Texteingabe verzögerungsfrei Verbesserungsvorschläge anbieten.

Typische Logik-Fehler

Boolesche Ausdrücke können nach Refactorings redundante $(A \wedge (A \vee B))$ oder widersprüchliche $(A \wedge \neg A)$ Ausdrücke enthalten. Eine statische Analyse nach Quine-McCluskey [14] kann derartige Redundanzen melden.

Sicherheitskritische Fehler

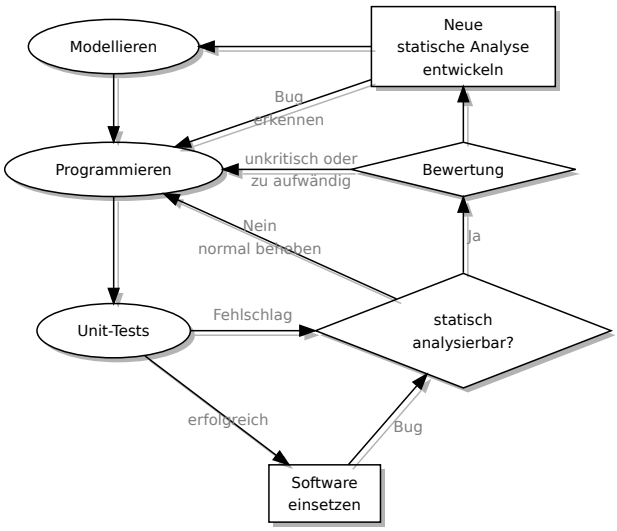
Die Pluginschnittstelle des `clang` Compilers wurde von Ruef [15] eingesetzt um den Heartbleed Bug [16] automatisiert zu erkennen. Dabei wurde eine Datenflussanalyse durchgeführt, die erkennt, wenn Nutzereingaben ohne weitere Prüfungen zur Indizierung von Arrays genutzt werden.

Modellverifikation

Pluginbasierte statische Analysen können den modellgetriebenen Entwicklungsprozess unterstützen, ohne dass Codegeneratoren eingesetzt werden müssen oder der Code manuell auf Modellkompatibilität geprüft werden muss. Dabei lädt das Plugin das Modell und eine oder mehrere statische Analysen prüfen modellgetrieben den Code. Durch die zusätzlichen Informationen des Modells, lassen sich komplexere Analysen durchführen als bei einer Codeanalyse, die den Code ohne Kontext betrachten muss.

4.2 Ablauf

Am Anfang jedes Projektes steht weiterhin ein abstraktes Modell. Aufgrund der fehlenden automatischen Codegenerierung ist es in dem vorgestellten Ablauf notwendig, dass das Modell nicht so detailliert wird, dass dem Programmierer die Freiheitsgrade genommen werden, die die Vorteile der manuellen Programmierung ausmachen. Derartige Übermodellierung fällt bei der Programmierung jedoch sofort auf, da der Modellchecker direkt den Konflikt melden kann, statt erst in einer späteren Prüfungsphase.



Wenn im Betrieb Fehler gemeldet werden, müssen diese in einem Zwischenschritt auf ihre statische Analysierbarkeit geprüft werden. Dabei gibt es mehrere Kriterien nach denen diese Bewertung durchgeführt wird:

1. Rate, mit der ähnliche Fehler auftreten
2. Kritikalität des Fehlers
   - Anzeigefehler
   - Logikfehler/algorithmische Fehler
   - Kontrollierter Programmabsturz
   - Memory Safety Verletzung
3. Aufwand

Fehlerrate

Wenn regelmäßig ähnliche Fehler im Code auftauchen, kann eine statische Analyse schnell weniger Aufwand bedeuten als weiterhin die Fehlerart manuell zu erkennen und zu beheben. Dies ist der klassische Ansatz, bei dem zuerst Erfahrung mit Fehlern gesammelt wird, und diese Fehler in zukünftigen Projekten ausgeschlossen werden. Der Vorteil hierbei ist eine kürzere

Zykluszeit zwischen dem Erkennen des Fehlermusters und der automatisierten Erkennung der einzelnen Fehler.

Kritikalität

Die Einstufung von Fehlern in eine Kritikalitätsstufe ist projektabhängig. Ein Anzeigefehler in einer Desktopanwendung kann vernachlässigt werden, während die korrekte Batteriestandsanzeige eines Quadrokopters Abstürze vermeiden kann.

Anzeigefehler: Schreibfehler in angezeigtem Text oder Grafikfehler, die die Benutzbarkeit nicht einschränken sind meist sehr spezifisch und nicht durch kontextfreie statische Analysen erkennbar. Der Einsatz von Textdatenbanken für die Internationalisierung trennt hier bereits Text von Code.

Logikfehler und algorithmische Fehler: Logikfehler wie z.B. unerreichbarer Code oder unnötige Anweisungen können oft bereits durch Syntaxanalysen erkannt werden. Eine kontextfreie Erkennung fehlerhaft implementierter Algorithmen ist jedoch nicht möglich. Hierbei muss entweder eine Spezifikation hinterlegt werden, gegen die der Code geprüft wird, oder der Algorithmus muss in einer formal verifizierbaren Sprache entwickelt werden und automatisiert in die Zielsprache übersetzt werden.

Kontrollierte Programmabstürze (z.B. das Auslösen von Assertions) treten auf, wenn die Schnittstellen von Bibliotheken falsch genutzt werden, und die Bibliotheken Laufzeitprüfungen zum Erkennen von Missbrauch einsetzen. `Ada` Compiler, gcc (ab 4.6), clang (ab 3.0) und der `rust` Compiler warnen oder verbieten Situationen in denen nicht alle Varianten eines Fehlercodes abgehandelt werden. Clang und der `rust` Compiler geben eine Warnung aus, wenn ein Funktionsaufruf einen Fehlercode zurückgibt, dieser jedoch nicht im Code abgehandelt wird. Eine statische Analyse von `clippy` generalisiert diese Warnung auf alle Funktionsaufrufe mit einem Rückgabewert.

Diese allgemein gehaltenen Konformitätsprüfungen der Schnittstellennutzung fangen nicht alle bibliotheksspezifischen Fehler ab. Bibliotheken können jedoch selbst statische Analysen mitliefern, die die korrekte Nutzung zur Compilezeit garantieren oder zumindest erleichtern.

Memory Safety Verletzungen wie z.B. die Dereferenzierung ungültiger Zeiger sind die Hauptursache von Sicherheitslücken in Software. Diese Art von Fehlern ist leicht auszunutzen und schwer zu verhindern in Sprachen die Zugriff auf die Hardware geben (`Ada`, `C`/`C++`). Neuanfänge wie die Programmiersprache `Rust` enthalten Analysen zur Garantie der Memory Safety. Abwärtskompatible Ansätze wie Stroustrup [17] bieten statische Analysen zur nachträglichen Verifikation von `C++` Code. Sowohl Neuanfänge als auch abwärtskompatible Ansätze haben jedoch gemein, dass es weiterhin Funktionen und Typen gibt, deren Implementierung nicht durch den Compiler als "Memory Safe" verifiziert wird, da eine vollständige Prüfung eine Lösung des Halteproblems vorraussetzt oder da der Code hardwareabhängige Instruktionen enhält.

Der Umkehrschluss ist nicht zu verifizieren, dass der Code korrekt ist, sondern zu prüfen, dass der Code keine als inkorrekt bekannte Strukturen oder Abläufe enthält. Ein Beispiel ist die ungeprüfte Umwandlung eines Zeigers auf ein Objekt des Types `T` in einen Zeiger auf ein Objekt des Types `U`, wenn das Alignment des Types `T` kleiner ist als das des Types `U`. Eine statische Analyse kann hierbei garantieren, dass eine derartige Konvertierung nur stattfinden kann, wenn zusätzlich eine Laufzeitprüfung des Alignments stattfindet.

4.2.1 Aufwand

Unter dem Aufwand ist sowohl der Entwicklungsaufwand der Analyse zu verstehen, als auch der Rechenaufwand den die Analyse benötigt. Syntaxanalysen und Strukturanalysen sind meist leicht zu entwerfen und benötigen nur wenige Millisekunden Ausführungzeit. Kontroll- und Datenflussanalysen hingegen können für den Entwickler merkliche Verlängerungen des Kompiliervorgangs auslösen.

Der Aufwand muss gegen die Kritikalität und die Fehlerrate abgewogen werden, damit die Entwicklung der statischen Analyse nicht mehr Kosten verursacht als die auftretenden Fehler.

Zusammenfassung

Statische Analysen erleichtern die fehlerarme Softwarentwicklung enorm. Tools zur Durchführung der Analysen sind jedoch sehr träge und allgemein gehalten. Es wurde ein flexibler Entwicklungszyklus zur Einbindung und Entwicklung allgemeiner, modellbasierter und projektspezifischer statischer Analysen durch Nutzung von Pluginschnittstellen der eingesetzten Compiler vorgestellt. Des Weiteren wurde ein Überblick des Standes der Compilerunterstützung und der existierenden Nutzung gegeben.

Literatur

1. Kersten M., Matthes J., Manga C.F., Zipser S., Keller H. (1999) Customizing UML for the development of distributed reactive systems and code generation to Ada 95. Ada User Journal 23:

2. Matthes J., Keller H.B., Heker W.-D., Kersten M., Fouda C. (2003) Zuverlässige Software durch den Einsatz von UML, MDA und der Programmiersprache Ada. In: GI Jahrestagung (Schwerpunkt "Sicherheit-Schutz und Zuverlässigkeit"). pp. 167–178

3. Johnson S.C. (1977) Lint, a C program checker.

Citeseer

4. Batsov B. (2016) A Ruby static code analyzer, based on the community Ruby style guide. https://github.com/bbatsov/rubocop.

5. Waldron R., Potter C., Sherov M., Pennisi M., Page L. (2016) A static code analysis tool for JavaScript. http://jshint.com/docs/.

6. Mitchell N. (2016) Haskell source code suggestions. https://github.com/ndmitchell/hlint.

7. Rust 'plugin' feature. https://github.com/rust-lang/rust/issues/29597.

8. GCC 4.5.0 plugin support. https://gcc.gnu.org/wiki/plugins.

9. Clang plugins. http://clang.llvm.org/docs/ClangPlugins.html.

10. Writing scala compiler plugins. http://www.scala-lang.org/old/node/140.

11. Extending and using GHC as a library - compiler plugins. https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/compiler-plugins.html.

12. Goregaokar M., Bogus A., Brandl G., Carton M. (2016) A bunch of lints to catch common mistakes and improve your Rust code. https://github.com/Manishearth/rust-clippy.

13. Facebook Plugins to clang-analyzer and clang-frontend. https://github.com/facebook/facebook-clang-plugins.

14. McCluskey E.J. (1956) Minimization of boolean functions*. Bell system technical Journal 35:1417–1444.

15. Ruef A. (2014) Using static analysis and clang to find heartbleed. http://blog.trailofbits.com/2014/04/27/using-static-analysis-and-clang-to-find-heartbleed/.

16. Durumeric Z., Kasten J., Adrian D., Halderman J.A., Bailey M., Li F., Weaver N., Amann J., Beekman J., Payer M., others (2014) The matter of heartbleed. In: Proceedings of the 2014 conference on internet measurement conference. ACM, pp. 475–488

17. Stroustrup B., Sutter H., Reis G.D. (2015) A brief introduction to c++'s model for type- and resource- safety.