4.3 Analisi statica di programmi

Angelo Gargantini

Testing e Verifica del Software AA 2526

1 aprile 2026

Analisi statica

- Possiamo utilizzare molte tecniche per provare a migliorare la qualità, inclusi testing, inspezione del codice e specifiche formali.
- Gli strumenti di analisi statica valutano il software in astratto, senza eseguire il software o considerando un input specifico.
- Piuttosto che provare a dimostrare che il codice soddisfa le sue specifiche, tali strumenti cercano violazioni di pratiche di programmazione ragionevoli o consigliate.

Analisi statica

- Possiamo utilizzare molte tecniche per provare a migliorare la qualità, inclusi testing, inspezione del codice e specifiche formali.
- Gli strumenti di analisi statica valutano il software in astratto, senza eseguire il software o considerando un input specifico.
- Piuttosto che provare a dimostrare che il codice soddisfa le sue specifiche, tali strumenti cercano violazioni di pratiche di programmazione ragionevoli o consigliate.

Analisi statica

- Possiamo utilizzare molte tecniche per provare a migliorare la qualità, inclusi testing, inspezione del codice e specifiche formali.
- Gli strumenti di analisi statica valutano il software in astratto, senza eseguire il software o considerando un input specifico.
- Piuttosto che provare a dimostrare che il codice soddisfa le sue specifiche, tali strumenti cercano violazioni di pratiche di programmazione ragionevoli o consigliate.

Analisi statica

Cosa è l'analisi statica

Uno strumento di analisi statica S analizza il codice sorgente di un programma P per determinare se soddisfa una proprietà ϕ, come ad esempio "P non deferenzia un puntatore nullo" • "P chiude tutti i file che ha aperto" • "Nessun cast in P guiderà a un ClassCastException".

Scoprire errori di tipi o variabili mai usate o non inizalizzate non richiede l'esecuzione del codice, basta analizzarlo.

Limiti dell'analisi statica

L'analisi statica ha diversi limiti. Vediamo alcuni casi tipici

- Mancanza di conoscenza di quello che il programma dovrebbe fare

```
int calculateArea(int length, int width) {
        return (length + width);
}
```

Uno strumento di analisi statico può rilevare un possibile overflow in questo calcolo. Ma non può determinare fondamentalmente che la funzione non faccia ciò che ci si aspetta!

Limiti dell'analisi statica

- Regole che non sono statisticamente verificabili
  Alcune regole di codifica dipendono dalla documentazione esterna. Oppure sono aperti all'interpretazione soggettiva.
  Per esempio nel CERT-C MSC04 si prescrive: Utilizzare i commenti in modo coerente e leggibile.
  Come si possono giudicare i commenti leggibili?
-
- Possibile applicazione di chatGPT?

Limiti dell'analisi statica

- Regole che non sono statisticamente verificabili
  Alcune regole di codifica dipendono dalla documentazione esterna. Oppure sono aperti all'interpretazione soggettiva.
  Per esempio nel CERT-C MSC04 si prescrive: Utilizzare i commenti in modo coerente e leggibile.
  Come si possono giudicare i commenti leggibili?
-
- Possibile applicazione di chatGPT?

Limiti dell'analisi statica

- Possibili difetti portano a falsi positivi e falsi negativi
  In alcune situazioni, uno strumento può solo segnalare che esiste un possibile difetto.

```
int divide() {
        int x;
        if(foo()) { x = 0; } else { x = 5; }
        return (10/x);
}
```

  Se non sappiamo nulla di foo (), non sappiamo quale valore avrà x.
- In generale il risultato è indecidibile. Ciò significa che gli strumenti possono segnalare difetti che in realtà non esistono. Oppure potrebbero non riuscire a segnalare vizi reali.

L'analisi statica non è una ricerca di bug

- Per la ricerca di difetti, il testing è ancora la tecnica migliore perchè può **provare la presenza** di difetti
- Lo scopo dell'analisi statica non è trovare bugs.

Analisi statica è più che la ricerca di bugs

- Controllo dello stile di codifica nel senso più ampio di questa parola. Include sia un controllo della formattazione sia l'uso di parentesi, nomi, etc.
- Non solo il codice eseguibile può essere analizzato. Risorse come i file JSON, YAML, XML e .properties possono (e devono!) essere verificate automaticamente per la validità.
- La compilazione (o l'analisi per i linguaggi di programmazione dinamici) è anche una sorta di analisi statica
- Il controllo ortografico è anche una sorta di analisi statica.

Il testing con analisi statica

Alcune volte il testing è usato in combinazione con analisi statica.

- I test di configurazione rappresentano in realtà una forma di analisi statica.
- Anche i test dell'architettura sono di analisi statica.

ArchUnit `https://www.archunit.org/`

ArchUnit is a free, simple and extensible library for checking the architecture of your Java code. That is, ArchUnit can check dependencies between packages and classes, layers and slices, check for cyclic dependencies and more. It does so by analyzing given Java bytecode, importing all classes into a Java code structure. ArchUnit's main focus is to automatically test architecture and coding rules, using any plain Java unit testing framework.

Analisi statica è più che la ricerca di bugs

Uno strumento di analisi statico S analizza il codice sorgente di un programma P per determinare se soddisfa una proprietà φ (P è negativo) o la viola (P risulta positivo). S accetta i programmi che soddisfano φ con qualche imprecisione. Potrebbe sbagliare in due modi:

- Se S è **sound**, (pessimistic o conservativo) se accetta un programma P, sicuramente soddisfa φ
  - Accetta solo programmi che soddisfano la proprietà.
    - Nessun **falso negativo**
  - Può rifiutare i programmi che soddisfano la proprietà.
    - Possibile falso positivo (sembra che ci sia un difetto ma non c'è).
- Se S è **completo** (ottimistico) se P soddisfa φ allora lo accetta
  - Accetta sempre programmi che soddisfano la proprietà
    - nessun falso positivo
  - Può accettare programmi che non soddisfano la proprietà.
    - Possibile falso negativo (sembra che non abbia il difetto ma ce l'ha - passono inosservati)

Analisi statica è più che la ricerca di bugs

Uno strumento di analisi statico S analizza il codice sorgente di un programma P per determinare se soddisfa una proprietà φ (P è negativo) o la viola (P risulta positivo). S accetta i programmi che soddisfano φ con qualche imprecisione. Potrebbe sbagliare in due modi:

- Se S è **sound**, (pessimistic o conservativo) se accetta un programma P, sicuramente soddisfa φ
  - Accetta solo programmi che soddisfano la proprietà.
    - Nessun **falso negativo**
  - Può rifiutare i programmi che soddisfano la proprietà.
    - Possibile falso positivo (sembra che ci sia un difetto ma non c'è).
- Se S è **completo** (ottimistico) se P soddisfa φ allora lo accetta
  - Accetta sempre programmi che soddisfano la proprietà
    - nessun falso positivo
  - Può accettare programmi che non soddisfano la proprietà.
    - Possibile falso negativo (sembra che non abbia il difetto ma ce l'ha - passono inosservati)

Analisi statica è più che la ricerca di bugs

Uno strumento di analisi statico S analizza il codice sorgente di un programma P per determinare se soddisfa una proprietà φ (P è negativo) o la viola (P risulta positivo). S accetta i programmi che soddisfano φ con qualche imprecisione. Potrebbe sbagliare in due modi:

- Se S è **sound**, (pessimistic o conservativo) se accetta un programma P, sicuramente soddisfa φ
  - Accetta solo programmi che soddisfano la proprietà.
    - Nessun **falso negativo**
  - Può rifiutare i programmi che soddisfano la proprietà.
    - Possibile falso positivo (sembra che ci sia un difetto ma non c'è).
- Se S è **completo** (ottimistico) se P soddisfa φ allora lo accetta
  - Accetta sempre programmi che soddisfano la proprietà
    - nessun falso positivo
  - Può accettare programmi che non soddisfano la proprietà.
    - Possibile falso negativo (sembra che non abbia il difetto ma ce l'ha - passono inosservati)

Analisi statica è più che la ricerca di bugs

Uno strumento di analisi statico S analizza il codice sorgente di un programma P per determinare se soddisfa una proprietà φ (P è negativo) o la viola (P risulta positivo). S accetta i programmi che soddisfano φ con qualche imprecisione. Potrebbe sbagliare in due modi:

- Se S è **sound**, (pessimistic o conservativo) se accetta un programma P, sicuramente soddisfa φ
  - Accetta solo programmi che soddisfano la proprietà.
    - Nessun **falso negativo**
  - Può rifiutare i programmi che soddisfano la proprietà.
    - Possibile falso positivo (sembra che ci sia un difetto ma non c'è).
- Se S è **completo** (ottimistico) se P soddisfa φ allora lo accetta
  - Accetta sempre programmi che soddisfano la proprietà
    - nessun falso positivo
  - Può accettare programmi che non soddisfano la proprietà.
    - Possibile falso negativo (sembra che non abbia il difetto ma ce l'ha - passono inosservati)

meglio falsi negativi o positivi?

- Potremmo preferire un tool che ha pochi falsi negativi e tanti falsi positivi
- un tool pessimistico, da tanti falsi allarmi che poi il programmartore sistemerà
- Però:
  - Troppi falsi positivi fanno perdere tempo al programmatore che quindi smetterà di usare questa tecnica
  - oppure ignorerà i warning (tipo “al lupo al lupo”)

meglio falsi negativi o positivi?

- Potremmo preferire un tool che ha pochi falsi negativi e tanti falsi positivi
- un tool pessimistico, da tanti falsi allarmi che poi il programmartore sistemerà
- Però:
  - Troppi falsi positivi fanno perdere tempo al programmatore che quindi smetterà di usare questa tecnica
  - oppure ignorerà i warning (tipo “al lupo al lupo”)

Ci sono tantissimi tool anche commerciali per l'analisi statica. Eccone alcuni:

What is findBugs?

- Static analysis tool to find defects in Java code
  - not a style checker
- Can find hundreds of defects in each of large apps such as Bea WebLogic, IBM Websphere, Sun's JDK
- real defects, stuff that should be fixed
- Doesn't focus on security
- lower tolerance for false positives

Common Wisdom about Bugs

- Programmers are smart
- Smart people don't make dumb mistakes
- We have good techniques (e.g., unit testing, pair programming, code inspections) for finding bugs early
- So, bugs remaining in production code must be subtle, and require sophisticated techniques to find

Would You Write Code Like This?

- `if (in == null) try { in.close(); ...`
  - Oops
  - This code is from Eclipse (versions 3.0 - 3.2)
- You may be surprised what is lurking in your code

Why Do Bugs Occur?

- Nobody is perfect
- Common types of errors:
  - Misunderstood language features, API methods
  - Typos (using wrong boolean operator, forgetting parentheses or brackets, etc.)
  - Misunderstood class or method invariants
- Everyone makes syntax errors, but the compiler catches them
  - What about bugs one step removed from a syntax error?

Infinite recursive loop

```
/** Construct a WebSpider */
public WebSpider() { WebSpider w = new WebSpider(); }
```

- Double check against JDK: Found 4 infinite recursive loops
- Including one written by Joshua Bloch

```
public String foundType() { return this.foundType(); }
```

- Smart people make dumb mistakes
- Embrace and fix your dumb mistakes

Hashcode/Equals

- Equal objects must have equal hash codes
- Programmers sometimes override equals() but not hashCode()
- Or, override hashCode() but not equals()
- Objects violating the contract won't work in hash tables, maps, sets
  - Examples (53 bugs in 1.6.0-b29)
    - javax.management.Attribute
    - java.awt.geom.Area

Fixing hashCode

- What if you want to define equals, but don't think your objects will ever get put into a HashTable?
- Suggestion:

```
public int hashCode() {
        assert false : "hashCode␣method␣not␣designed";
        return 0;
}
```

- Or throw a RuntimeException

Null Pointer Dereference

- Dereferencing a null value results in NullPointerException
- Warn if there is a statement or branch that if executed, guarantees a NPE
- Example:

```
// Eclipse 3.0.0M8
Control c = getControl();
if (c == null && c.isDisposed()) return;

// Eclipse 3.0.0M8
String sig = type.getSignature();
if (sig != null || sig.length() == 1) {
        return sig;
}
// JDK 1.5 build 42
if (name != null || name.length > 0) { ..
```

More Null Pointer Dereferences

```
javax.security.auth.kerberos.KerberosTicket
// flags is a parameter
// this.flags is a field
if (flags != null) {
        if (flags.length >= NUM_FLAGS)
                this.flags = ...
        else
                this.flags = ...
} else
        this.flags = ...
        if (flags[RENEWABLE_TICKET_FLAG]) {
```

Redundant Null Comparison

- Comparing a reference to null when it is definitely null or definitely non-null
- Not harmful per se, but often indicates an inconsistency that might be a bug
- Example (JBoss 4.0.0DR3):

```
protected Node findNode(Fqn fqn, ...) {
        int treeNodeSize = fqn.size(); ...
        if (fqn == null) return null;
```

How should we fix this bug?

```
if (name != null || name.length > 0)
```

Should we just change it to

```
if (name != null && name.length > 0)
```

- Will that fix it?
- We have no idea.
- Obviously, we've never tested the situation when name is null.
- Try to write a test case first, then apply the obvious fix

Bad Binary operations

- Binary operators can be used as bitwise operators

```
if ((f.getStyle () & Font.BOLD) == 1) {
        sbuf.append ("<b>"); isBold = true;
}
if ((f.getStyle () & Font.ITALIC) == 1) {
        sbuf.append ("<i>"); isItalic = true;
}
```

What is the meaning? True? Or really 1?

Doomed Equals

- Equals that will never be true

```
public static final ASDDVersion getASDDVersion(BigDecimal version){
        if(SUN_APPSERVER_7_0.toString().equals(version))
                return SUN_APPSERVER_7_0;
...
```

Unintended regular expression

- The use of regular expression?

```
String[] valueSegments = value.split("."); // NOI18N
```

field Self Assignment

```
public TagHelpItem(String name, String file, String startText,
        int startOffset, String endText, int endOffset,
        String textBefore, String textAfter){

        this.name = name;
        this.file = file;
        this.startText = startText;
        this.startTextOffset = startTextOffset;
        this.endText = endText;
        this.endTextOffset = endTextOffset;
        this.textBefore = textBefore;
        this.textAfter = textAfter;
        this.identical = null;
}
```

Bad Naming

- Wrong capitalization

```
package org.eclipse.jface.dialogs;
public abstract
class Dialog extends Window {
        protected Button getOKButton() {
                return getButton(IDialogConstants.OK_ID);
        };
}
public class InputDialog extends Dialog {
        protected Button getOkButton() {
                return okButton;
        };
}
```

Confusing/bad naming

- Methods with identical names and signatures -- but different capitalization of names -- could mean you don't override method in superclass -- confusing in general
- Method name same as class name -- gets confused with constructor

Ignored return values

- Lots of methods for which return value always should be checked -- E.g., operations on immutable objects

```
// Eclipse 3.0.0M8
String name= workingCopy.getName();
name.replace('/', '.');
```

Ignored Exception Creation

```
/**
* javax.management.ObjectInstance
* reference impl., version 1.2.1 **/
public ObjectInstance(ObjectName objectName, String className) {
        if (objectName.isPattern()) {
                new RuntimeOperationsException(
                new IllegalArgumentException( "Invalid name->"+ objec
        }
        this.name = objectName;
        this.className = className;
}
```

- Inconsistent Synchronization
- Common idiom for thread safe classes is to synchronize on the receiver object (“this”)
- We look for field accesses -- find classes where lock on “this” is sometimes, but not always, held -- Unsynchronized accesses, if reachable from multiple threads, constitute a race condition

Inconsistent Synchronization Example

- GNU Classpath 0.08,

```
java.util.Vector
public int lastIndexOf(Object elem) {
        return lastIndexOf(elem, elementCount -- 1);
}
public synchronized int lastIndexOf( Object e, int index) { ... }
```

Unconditional Wait

- Before waiting on a monitor, the condition should be almost always be checked
  - Waiting unconditionally almost always a bug
  - If condition checked without lock held, could miss the notification
  - Example (JBoss 4.0.0DR3):

```
if (!enabled) {
        try { log.debug(...);
              synchronized (lock) { lock.wait(); }
```

  - condition can become true after it is checked but before the wait occurs

Bug Categories

- Correctness
- Bad Practice
  - equals without hashCode
  - comparing Strings with ==,
  - equals should handle null argument
- Performance
- Multithreaded correctness
- Malicious code vulnerability

PMD - Java Rules

- Best Practices
- Code Style
- Design
- Documentation
- Error Prone
- Multithreading
- Performance
- Security

Other tools

- **UCDetector:** Unnecessary (dead) code Code where the visibility could be changed to protected, default or private; Methods or fields, which can be final
- **SonarLint**: thousands of rules ; which detect common mistakes, tricky bugs and known vulnerabilities. An IDE extension that helps you detect and fix quality issues as you write code.
- **Mypy** is an optional static type checker for Python that aims to combine the benefits of dynamic (or "duck") typing and static typing.
- **PyLint**: Coding Standard, Error detection, Refactoring help ...
- **stan4j**: Dependency Analysis, Quality Metrics, ...
- **Structure101 Studio:** Una serie di strumenti per comprendere, analizzare, eseguire il refactoring e controllare basi di codice