L12: Invariants and Specification Inference

Software Analysis

IIT Guwahati · August 2026

Invariants

- A **program invariant** holds during the execution of a program, or during some portion of it

Invariants

- A **program invariant** holds during the execution of a program, or during some portion of it
- A **loop invariant** holds immediately before and after each iteration

Invariants

- A **program invariant** holds during the execution of a program, or during some portion of it
- A **loop invariant** holds immediately before and after each iteration
- A **class invariant** holds for all objects of a class: established on construction, maintained between calls to public methods

Invariants

- A **program invariant** holds during the execution of a program, or during some portion of it
- A **loop invariant** holds immediately before and after each iteration
- A **class invariant** holds for all objects of a class: established on construction, maintained between calls to public methods

Invariants are what make pre- and post-conditions of code containing loops and objects provable.

Naik, *Software Specifications*, CIS 5470, Penn

Black Board

Quiz: Loop Invariants

Which of the following are *valid* loop invariants, and which are strong enough to prove the post-condition?

```
i = 0; s = 0;
while (i != n) { s = s + a[i]; i = i + 1; }
// post:  s == a[0] + ...  + a[n-1]
```

1. `i == i`
2. `0 <= i && i <= n`
3. `s == a[0] + ...  + a[i-1]`
4. `s >= 0`

Solution: Loop Invariants

1. `i == i` — **valid**, but vacuous: it constrains nothing

Solution: Loop Invariants

1. `i == i` — **valid**, but vacuous: it constrains nothing
2. `0 <= i && i <= n` — **valid**, and needed for the array accesses, but says nothing about `s`

Solution: Loop Invariants

1. `i == i` — **valid**, but vacuous: it constrains nothing
2. `0 <= i && i <= n` — **valid**, and needed for the array accesses, but says nothing about `s`
3. `s == a[0] + ... + a[i-1]` — **valid** and **sufficient**: with $i = n$ at exit it gives the post-condition

Solution: Loop Invariants

1. `i == i` — **valid**, but vacuous: it constrains nothing
2. `0 <= i && i <= n` — **valid**, and needed for the array accesses, but says nothing about `s`
3. `s == a[0] + ... + a[i-1]` — **valid** and **sufficient**: with $i = n$ at exit it gives the post-condition
4. `s >= 0` — **not valid**: it fails if the array contains negative elements

Solution: Loop Invariants

1. `i == i` — **valid**, but vacuous: it constrains nothing
2. `0 <= i && i <= n` — **valid**, and needed for the array accesses, but says nothing about `s`
3. `s == a[0] + ... + a[i-1]` — **valid** and **sufficient**: with $i = n$ at exit it gives the post-condition
4. `s >= 0` — **not valid**: it fails if the array contains negative elements

Validity is not enough: a loop has infinitely many valid invariants, and the useful one is the one that implies the post-condition.

Example: Class Invariant

```
class Rational {
    //@invariant denom != 0;
    int num, denom;

    //@requires d != 0;
    Rational(int n, int d) { num = n; denom = d; }

    double getDouble() { return ((double) num) / denom; }

    public static void main(String[] a) {
        int n = readInt(), d = readInt();
        if (d == 0) return;
        Rational r = new Rational(n, d);
        print(r.getDouble());
    }
}
```

Example: Class Invariant

```
class Rational {
    //@invariant denom != 0;
    int num, denom;

    //@requires d != 0;
    Rational(int n, int d) { num = n; denom = d; }

    double getDouble() { return ((double) num) / denom; }

    public static void main(String[] a) {
        int n = readInt(), d = readInt();
        if (d == 0) return;
        Rational r = new Rational(n, d);
        print(r.getDouble());
    }
}
```

Example: Class Invariant

```
class Rational {
    //@invariant denom != 0;
    int num, denom;

    //@requires d != 0;
    Rational(int n, int d) { num = n; denom = d; }

    double getDouble() { return ((double) num) / denom; }

    public static void main(String[] a) {
        int n = readInt(), d = readInt();
        if (d == 0) return;
        Rational r = new Rational(n, d);
        print(r.getDouble());
    }
}
```

The class invariant makes the absence of a division by zero in `getDouble()` provable.

Quiz: Class Invariants

Is `count >= 0` a class invariant of this class? If not, what is the smallest change that makes it one?

```
class Counter {
    //@invariant count >= 0;
    int count;

    Counter()       { count = 0; }
    void inc()      { count = count + 1; }
    void dec()      { count = count - 1; }
    int  get()      { return count; }
}
```

Solution: Class Invariants

- `Counter()` **establishes** it: $0 \geq 0$

Solution: Class Invariants

- `Counter()` **establishes** it: $0 \geq 0$
- `inc()` and `get()` **preserve** it

Solution: Class Invariants

- `Counter()` **establishes** it: $0 \geq 0$
- `inc()` and `get()` **preserve** it
- `dec()` **breaks** it: calling it on a counter with `count == 0` yields -1

Solution: Class Invariants

- `Counter()` **establishes** it: $0 \geq 0$
- `inc()` and `get()` **preserve** it
- `dec()` **breaks** it: calling it on a counter with `count == 0` yields -1

Two ways to repair the class:

```
//@requires count > 0;            // oblige the caller
void dec() { count = count - 1; }

void dec() { if (count > 0) count = count - 1; }    // guard in the method
```

Inferring Specifications

Motivation and Approach

- Programmers are reluctant to write and maintain specifications

Motivation and Approach

- Programmers are reluctant to write and maintain specifications
- An idea: infer them automatically

Motivation and Approach

- Programmers are reluctant to write and maintain specifications
- An idea: infer them automatically
- Many approaches to **specification mining**, with varying trade-offs

Motivation and Approach

- Programmers are reluctant to write and maintain specifications
- An idea: infer them automatically
- Many approaches to **specification mining**, with varying trade-offs
- A popular guess-and-check approach: the **Houdini algorithm**

  requires an automated theorem prover

Houdini Algorithm

- An annotation assistant for the static modular verifier **ESC/Java**

Flanagan & Leino, *Houdini, an Annotation Assistant for ESC/Java*, FME 2001 · Naik, *Software Specifications*, CIS 5470, Penn

Houdini Algorithm

- An annotation assistant for the static modular verifier **ESC/Java**
- Generate many **candidate invariants**; use ESC/Java to verify or refute each

Flanagan & Leino, *Houdini, an Annotation Assistant for ESC/Java*, FME 2001 · Naik, *Software Specifications*, CIS 5470, Penn

Houdini Algorithm

- An annotation assistant for the static modular verifier **ESC/Java**
- Generate many **candidate invariants**; use ESC/Java to verify or refute each
- Candidates may be guessed by

  static analysis — mined from the source code using heuristics

  dynamic analysis — facts observed while running the program, as in **Daikon**

Flanagan & Leino, *Houdini, an Annotation Assistant for ESC/Java*, FME 2001 · Naik, *Software Specifications*, CIS 5470, Penn

Houdini: Example Candidate Invariants

For `int i, j`, with `cmp` $\in \{<, <=, ==, !=, >, >=\}$:

```
//@ invariant i cmp j;
//@ invariant i cmp 0;
```

Houdini: Example Candidate Invariants

For `int i, j`, with `cmp` $\in \{<, <=, ==, !=, >, >=\}$:

```
//@ invariant i cmp j;
//@ invariant i cmp 0;
```

For `Object[] a`:

```
//@ invariant a != null;
//@ invariant a.length cmp i;
//@ invariant (forall int k; 0 <= k && k < a.length ==> a[k] != null);
```

Houdini: Example Candidate Invariants

For `int i, j`, with `cmp` $\in \{<, <=, ==, !=, >, >=\}$:

```
//@ invariant i cmp j;
//@ invariant i cmp 0;
```

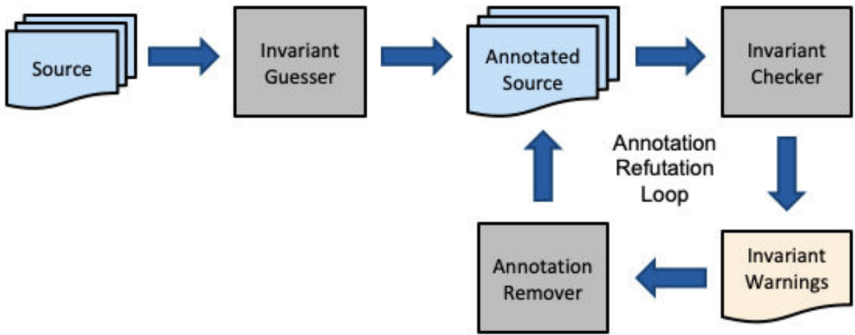
For `Object[] a`:

```
//@ invariant a != null;
//@ invariant a.length cmp i;
//@ invariant (forall int k; 0 <= k && k < a.length ==> a[k] != null);
```

For `boolean f`:

```
//@ invariant f == false;
//@ invariant f == true;
```

Workflow of Houdini



Houdini Algorithm

Input: An unannotated program P

Output: An annotated version of P

Algorithm:

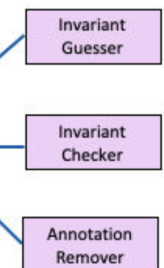
Generate set of candidate invariants and put into P;

repeat

invoke the invariant checker to check P;

remove any refuted candidate invariants from P;

until no warnings remain;



Example: Houdini on a Buggy Program

```
class Rational {
    //@invariant num != 0;
    //@invariant denom != 0;
    int num, denom;

    //@requires n != 0;
    //@requires d != 0;
    Rational(int n, int d) { num = n; denom = d; }

    double getDouble() { return ((double) num) / denom; }

    public static void main(String[] a) {
        int x = readInt(), y = readInt();
        if (x == 0) return;
        Rational r = new Rational(x, y);
        print(r.getDouble());
    }
}
```

Example: Houdini on a Buggy Program

```
class Rational {
    //@invariant num != 0;
    //@invariant denom != 0;
    int num, denom;

    //@requires n != 0;
    //@requires d != 0;
    Rational(int n, int d) { num = n; denom = d; }

    double getDouble() { return ((double) num) / denom; }

    public static void main(String[] a) {
        int x = readInt(), y = readInt();
        if (x == 0) return;
        Rational r = new Rational(x, y);
        print(r.getDouble());
    }
}
```

The four annotations in blue are **guesses**; the guard checks the wrong variable.

Refutation on the Buggy Program

1. Check the constructor's pre-conditions at its only call site, in `main`: `x` is guarded but `y` is not, so `requires d != 0` is removed

Refutation on the Buggy Program

1. Check the constructor's pre-conditions at its only call site, in `main`: `x` is guarded but `y` is not, so `requires d != 0` is removed
2. Check the class invariants as implicit post-conditions of the constructor: without `d != 0`, `denom` may be zero, so `invariant denom != 0` is removed

Refutation on the Buggy Program

1. Check the constructor's pre-conditions at its only call site, in `main`: `x` is guarded but `y` is not, so `requires d != 0` is removed
2. Check the class invariants as implicit post-conditions of the constructor: without `d != 0`, `denom` may be zero, so `invariant denom != 0` is removed
3. Without that class invariant, ESC/Java reports a possible **division by zero** in `getDouble()` and rejects the program

Refutation on the Buggy Program

1. Check the constructor's pre-conditions at its only call site, in `main`: `x` is guarded but `y` is not, so `requires d != 0` is removed
2. Check the class invariants as implicit post-conditions of the constructor: without `d != 0`, `denom` may be zero, so `invariant denom != 0` is removed
3. Without that class invariant, ESC/Java reports a possible **division by zero** in `getDouble()` and rejects the program

Example: Houdini on the Corrected Program

```
class Rational {
    //@invariant num != 0; // removed
    //@invariant denom != 0; // kept
    int num, denom;

    //@requires n != 0; // removed
    //@requires d != 0; // kept
    Rational(int n, int d) { num = n; denom = d; }

    double getDouble() { return ((double) num) / denom; }

    public static void main(String[] a) {
        int x = readInt(), y = readInt();
        if (y == 0) return;
        Rational r = new Rational(x, y);
        print(r.getDouble());
    }
}
```

Example: Houdini on the Corrected Program

```
class Rational {
    //@invariant num != 0; // removed
    //@invariant denom != 0; // kept
    int num, denom;

    //@requires n != 0; // removed
    //@requires d != 0; // kept
    Rational(int n, int d) { num = n; denom = d; }

    double getDouble() { return ((double) num) / denom; }

    public static void main(String[] a) {
        int x = readInt(), y = readInt();
        if (y == 0) return;
        Rational r = new Rational(x, y);
        print(r.getDouble());
    }
}
```

Example: Houdini on the Corrected Program

```
class Rational {
    //@invariant num != 0; // removed
    //@invariant denom != 0; // kept
    int num, denom;

    //@requires n != 0; // removed
    //@requires d != 0; // kept
    Rational(int n, int d) { num = n; denom = d; }

    double getDouble() { return ((double) num) / denom; }

    public static void main(String[] a) {
        int x = readInt(), y = readInt();
        if (y == 0) return;
        Rational r = new Rational(x, y);
        print(r.getDouble());
    }
}
```

No warnings remain, and the surviving class invariant proves the division safe.

Quiz: Inferring Contracts Using Houdini

Which pre- and post-conditions does Houdini infer for `foo` and `bar`?

```
main() {
    foo(5, 0);
}
```

```
foo(x, y) {
    if (x <= 0) z := y;
    else z := bar(x, y);
    return z;
}
```

```
bar(x, y) {
    x := x - 1;
    y := y + 1;
    return foo(x, y);
}
```

	candidate pre-conditions				candidate post-conditions	
	`x >= 0`	`y >= 0`	`x == y`	`x > 0`	`ret >= 0`	`ret == 0`
`foo`						
`bar`						

Solution: Inferring Contracts Using Houdini

	candidate pre-conditions				candidate post-conditions	
	`x >= 0`	`y >= 0`	`x == y`	`x > 0`	`ret >= 0`	`ret == 0`
`foo`						
`bar`						

Solution: Inferring Contracts Using Houdini

	candidate pre-conditions				candidate post-conditions	
	`x >= 0`	`y >= 0`	`x == y`	`x > 0`	`ret >= 0`	`ret == 0`
`foo`	yes	yes	–	–		
`bar`						

- `foo`: `x == y` fails at `foo(5, 0)`; `x > 0` fails at the call in `bar`, after `x := x - 1`

Solution: Inferring Contracts Using Houdini

	candidate pre-conditions				candidate post-conditions	
	`x >= 0`	`y >= 0`	`x == y`	`x > 0`	`ret >= 0`	`ret == 0`
`foo`	yes	yes	–	–	yes	–
`bar`						

- `foo`: `x == y` fails at `foo(5, 0)`; `x > 0` fails at the call in `bar`, after `x := x - 1`
- `foo`: `ret >= 0` holds on both branches; `ret == 0` fails

Solution: Inferring Contracts Using Houdini

	candidate pre-conditions				candidate post-conditions	
	`x >= 0`	`y >= 0`	`x == y`	`x > 0`	`ret >= 0`	`ret == 0`
`foo`	yes	yes	–	–	yes	–
`bar`	–	yes	–	yes	yes	–

- `foo`: `x == y` fails at `foo(5, 0)`; `x > 0` fails at the call in `bar`, after `x := x - 1`
- `foo`: `ret >= 0` holds on both branches; `ret == 0` fails
- `bar` is called only where `x > 0`, and returns `foo(x, y)`

Houdini: Pros and Cons

Pros

- Infers both loop invariants and method contracts
- Infers the strongest invariant within the candidate set
- Easy to implement

Houdini: Pros and Cons

Pros

- Infers both loop invariants and method contracts
- Infers the strongest invariant within the candidate set
- Easy to implement

Cons

- Infers only **conjunctions**: from candidates a, b, c it can infer $a \wedge b \wedge c$, but not $a \vee \neg b$

Houdini: Pros and Cons

Pros

- Infers both loop invariants and method contracts
- Infers the strongest invariant within the candidate set
- Easy to implement

Cons

- Infers only **conjunctions**: from candidates a, b, c it can infer $a \wedge b \wedge c$, but not $a \vee \neg b$
- High-quality invariants require many candidates, which makes refutation expensive

Houdini: Pros and Cons

Pros

- Infers both loop invariants and method contracts
- Infers the strongest invariant within the candidate set
- Easy to implement

Cons

- Infers only **conjunctions**: from candidates a, b, c it can infer $a \wedge b \wedge c$, but not $a \vee \neg b$
- High-quality invariants require many candidates, which makes refutation expensive
- No guarantee that the inferred invariants verify the property of interest

Flanagan & Leino, *Houdini, an Annotation Assistant for ESC/Java*, FME 2001 · Naik, *Software Specifications*, CIS 5470, Penn

Finding Inductive Loop Invariants using Large Language Models

ADHARSH KAMATH, Microsoft Research, India
ADITYA SENTHILNATHAN*, Cornell, USA
SAIKAT CHAKRABORTY, Microsoft Research, USA
PANTAZIS DELIGIANNIS, Microsoft Research, USA
SHUVENDU K. LAHIRI, Microsoft Research, USA
AKASH LAL, Microsoft Research, India
ASEEM RASTOGI, Microsoft Research, India
SUBHAJIT ROY, IIT Kanpur, India
RAHUL SHARMA, Microsoft Research, India