Greatest Common Divider (Eiffel)

JSO

19 June 2020

Contents

1 Formal Methods

In mathematics, the Euclidean algorithm is an efficient method for computing the greatest common divisor (GCD) of two integers, the largest number that divides them both without a remainder. It is named after the ancient Greek mathematician Euclid, who first described it in his Elements (c. 300 BC). It is used in cryptographic computations to ensure the security of a variety of systems (*Wikipedia*).

> Cybersecurity is everyone's problem. The target may be the electric grid, government systems storing sensitive personnel data, intellectual property in the defense industrial base, or banks and the financial system. Adversaries range from small-time criminals to nation states and other determined opponents who will explore an ingenious range of attack strategies. And the damage may be tallied in dollars, in strategic advantage, or in human lives. Systematic, secure system design

> is urgently needed, and we believe that rigorous formal methods are essential for substantial improvements.
>
> Formal methods enable reasoning from logical or mathematical specifications of the behaviors of computing devices or processes; they offer rigorous proofs that all system behaviors meet some desirable property. They are crucial for security goals, because they can show that no attack strategy in a class of strategies will cause a system to misbehave. Without requiring piecemeal enumeration, they rule out a range of attacks. They offer other benefits too: Formal specifications tell an implementer unambiguously what to produce, and they tell the subsequent user or integrator of a component what to rely on it to do. Since many vulnerabilities arise from misunderstandings and mismatches as components are integrated, the payoff from rigorous interface specifications is large.[1]

2 Using Formal Methods to Verify Euclid's algorithm

To understand the use of formal methods, let's look at a simple example. In Fig. 1, Euclid's algorithm is implemented in Golang.[2]

2.1 Testing can show the presence of bugs, but not their absence

There is a while loop in the code of Fig. 1. How sure are we that this implementation always terminates? If it does terminate, how do we know it always terminates with the correct result?

We can test the $gcd(m, n)$ subroutine. For example, we can provide inputs $m := 111$ and $n := 259$, and then we can check if the output is $gcd(111, 259) = 37$.

If we discover an an error in the code, then we can fix it. But here is the concern with testing alone: *Dijkstra*: "testing can show the presence of bugs, but not their absence".

- It does not matter how many tests we run, we can never exhaustively check the correctness of the algorithm. There are just too many combinations of inputs!
- To write a test, we also need to (manually?) compute the answer, a time consuming process. For example, to test gcd(111,259), we had to first manually compute the GCD by hand.[3]

The earlier NSF report perhaps words it too strongly, but there is more than a grain of truth to it.

> Formal methods are the only reliable way to achieve security and privacy in computer systems. Formal methods, by modeling computer systems and adversaries,

[1] *Report on the NSF Workshop on Formal Methods for Security*, 2016, `https://arxiv.org/pdf/1608.00678.pdf`.

[2] Try it at `https://play.golang.org`.

[3] GCD is built-in in most programming languages. But we are assuming that, for the sake of illustration, that we are computing a new function, one for which there is no oracle.

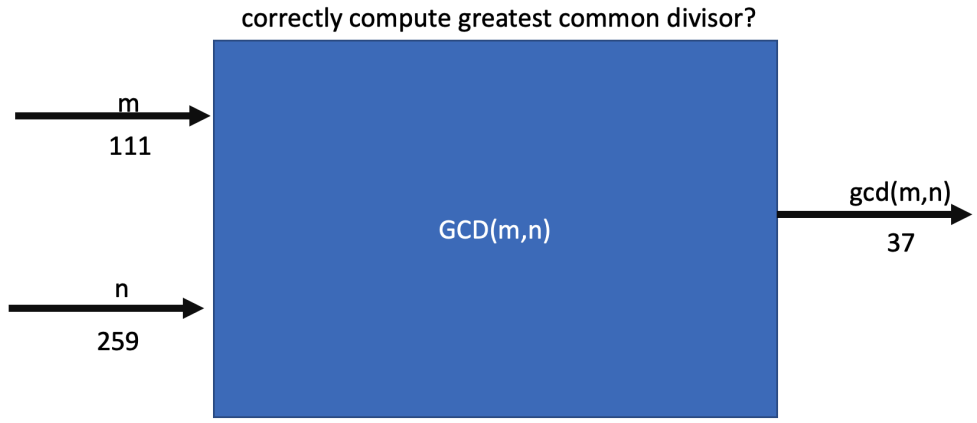
```
// golang
package main

import (
        "fmt"
)

// greatest common divisor (GCD) via Euclidean algorithm
// use only addition and subtraction
func gcd(m, n int) int {
        x := m
        y := n
        for x != y {                    // while loop
                if x < y {
                        y = y - x
                } else {
                        x = x - y
                }
        }
        return x
}

func main() {
        fmt.Println(gcd( 111, 259))
        fmt.Println(gcd(-111, 259))
}
```

The statement `fmt.Println(gcd(-111, 259))` is non-terminating. One might add an assert or defensive programming, neither of which is ideal.

Figure 1: GCD implemented in Go

> can prove that a system is immune to entire classes of attacks (provided the assumptions of the models are satisfied). By ruling out entire classes of potential attacks, formal methods offer an alternative to the "cat and mouse" game between adversaries and defenders of computer systems.
>
> Formal methods can have this effect because they apply a scientific method. They provide scientific foundations in the form of precise adversary and system models, and derive cogent conclusions about the possible behaviors of the system as the adversary interacts with it. This is a central aspect of providing a science of security.

For a formal proof Euclid's algorithm in TLA+, see `https://lamport.azurewebsites.net/pubs/euclid.pdf`. Tools such as TLA+ has been used at Amazon, Microsoft and elsewhere.



3 Correctness needs a Specification

To judge whether code is correct, we need a **specification**—this is something different from the Go **implementation** in Fig. 1. A specification is the software engineering equivalent of blue-prints in other engineering disciplines.

> Our recommendations are threefold, ... First, computer science majors, many of whom will be the designers and implementers of next-generation systems, should get a grounding in logic, ... "To designers of complex systems, the need for formal specs should be as obvious as the need for blueprints of a skyscraper. (Lesley Lamport)
>
> The methods, tools, and materials for educating students about "formal specs" are ready for prime time. Mechanisms such as "design by contract," now available in mainstream programming languages, should be taught as part of introductory programming, as is done in the introductory programming language sequence at Carnegie Mellon University. ⋯ We are failing our computer science majors if we do not teach them about the value of formal specifications.[4]

In Eiffel, we can specify the GCD algorithm using Design by Contract (DbC), as shown in Fig. 2.

If the GCD query is invoked by a client in a manner that violates the precondition (e.g. `gcd(-111,259)`) then this illegal call will automatically terminate with a precondition violation:

[4] "Teach Foundational Language Principles", Thomas Ball and Benjamin Zorn , *Communications of the ACM*, May 2015, Vol. 58 No. 5, Pages 30-31. `https://cacm.acm.org/magazines/2015/5/186023-teach-foundational-language-principles/fulltext`.

Thomas Ball (tball@microsoft.com) is a principal researcher and co-manager of the Research in Software Engineering (RiSE) group at Microsoft Research, Redmond, WA. Benjamin Zorn (zorn@microsoft.com) is a principal researcher and co-manager of the Research in Software Engineering (RiSE) group at Microsoft Research, Redmond, WA.

```
gcd(m, n: INTEGER): INTEGER
     -- return the greatest common divider of m and n
  require
    m ≥ 1 ∧ n ≥ 1 -- at least should not be zero
  ensure
       gcd_spec: Result = max(divisors(m) ∩ divisors(n))
  end
```

The **require** clause is a precondition: gcd is a partial function that is not well-defined for all possible inputs. For example, what is $gcd(0, 0)$? So the precondition documents the fact that a client using this function must check that the precondition is true before calling it. Without loss of generality, our precondition is $m \geq 1 \wedge n \geq 1$.

The **ensure** clause is a postcondition. It asserts that the function must terminate with this condition true. But how shall we write this postcondition? In general, for that we need to define what a GCD is using predicate logic and set theory (with the help of the Mathmodels library).

- $divisors(n)$ is the set of all divisors of the number n.
- $max(S)$ is the maximum of the set of numbers S.
- $gcd\_spec(m, n)$ is the GCD of the numbers m and n.

Formally, using set theory and predicate logic, we write

- $divisors(q) \mathrel{\widehat{=}} \{d \in 1 \mathinner{..} q \mid divides(d, q)\}$, where $divides(d, q)$ is true if d divides q, i.e. $p \mod d = 0$.
- $gcd\_spec(m, n) \mathrel{\widehat{=}} max(divisors(m) \cap divisors(n))$.

These specifications can themselves be written in an Eiffel-like form, e.g. $max(S)$ is as follows:

```
max(s: SET[INTEGER]): INTEGER
  require s ≠ ∅
  ensure (Result ∈ s) ∧ (∀i ∈ s | Result ≥ i)
```

Figure 2: Specification of the GCD query in Eiffel

FAILED (1 failed & 2 passed out of 3)		
Case Type	**Passed**	**Total**
Violation	0	0
Boolean	2	3
All Cases	2	3
State	**Contract Violation**	**Test Name**
Test1	ROOT	

3.1 Termination and Correctness

But we still need to prove that in the case the client makes a legal call, the GCD query terminates, and terminates with the correct result (i.e. satisfies the specification).

To prove termination and correctness, we must provide the implementation with a loop **variant** (i.e. $x + y$) and **loop invariant** (i.e. $gcd\_spec(x, y) = gcd\_spec(m, n)$) as shown in Fig. 3.

Figure 3: Specification and implementation of the GCD query in Eiffel

```
gcd(m, n: INTEGER): INTEGER
     -- return the greatest common divider of m and n
  require
    m ≥ 1 ∧ n ≥ 1 -- at least should not be zero
  local
    x, y: INTEGER
  do
    from
      x := m; y := n
    invariant
      inv: gcd_spec(x,y) = gcd_spec(m,n)
    until
      x = y
    loop
      if x < y then
        y := y - x
      else -- x < y
        x := x - y
      end
    variant x + y
    end
    check x = y and x = gcd_spec(m,n) end
    Result := x
  ensure
      Result = gcd_spec(m,n)
  end
```

4 Proof of Termination and Correctness

There are five proof obligations that must be discharged to prove that the loop terminates, and terminates correctly.

4.1 Prove that P is true initially

$$
\begin{aligned}
& wp(a := 0, P) \\
= & \quad \{\text{defn. of weakest precondition for assignment}\} \\
& P[a := 0] \\
= & \quad \{\text{defn. of } P\} \\
& (0 \leq a^2 \leq n)[a := 0] \\
= & \quad \{\text{substituting all free occurrences of } a \text{ in } P \text{ by zero}\} \\
& 0 \leq 0^2 \leq n \\
= & \quad \{0 \leq 0^2 \leq n \ \equiv \ 0 \leq n \text{ and Leibniz}\} \\
& 0 \leq n
\end{aligned}
$$

5 Prove that each iteration of the loop preserves P

Must show that: $\{P \wedge B\}\ a := a + 1\ \{P\}$, where B is the guard of the loop.

$$
\begin{aligned}
& P \wedge B \\
= & \quad \{\text{definition pf } P \text{ and } B\} \\
& (0 \leq a^2 \leq n) \wedge ((a+1)^2 \leq n) \\
\Rightarrow & \quad \{\text{weakening}\} \\
& 0 \leq (a+1)^2 \leq n \\
= & \quad \{\text{by substitution}\} \\
& (0 \leq a^2 \leq n)[a := a + 1] \\
= & \quad \{\text{defn of } P\} \\
& P[a := a + 1] \\
= & \quad \{\text{defn. of wp}\} \\
& wp(a := a + 1, P)
\end{aligned}
$$

6 Prove that $P \wedge \neg B \Rightarrow R$, i.e. the postcondition

$$
\begin{array}{ll}
 & P \wedge \neg B \\
= & \quad \{\text{defn. of } P \text{ and } B\} \\
 & (0 \leq a^2 \leq n) \wedge \neg((a+1)^2 \leq n) \\
= & \quad \{\text{arithmetic and negation}\} \\
 & 0 \leq a^2 \leq n) \wedge ((a+1)^2 > n) \\
= & \quad \{\text{rearranging}\} \\
 & 0 \leq a^2 \leq n < (a+1)^2
\end{array}
$$

7 Show that the variant t is bounded from below

$$
\begin{array}{ll}
 & P \wedge B \\
= & \quad \{\text{definition pf } P \text{ and } B\} \\
 & (0 \leq a^2 \leq n) \wedge ((a+1)^2 \leq n) \\
\Rightarrow & \quad \{\text{weakening}\} \\
 & a^2 \leq n \\
= & \quad \{\text{sqrt of both sides and } n \geq 0\} \\
 & a \leq \sqrt{n} \\
= & \quad \{\text{arithmetic}\} \\
 & \sqrt{n} - a \ \geq \ 0 \\
= & \quad \{\text{defn. of } t\} \\
 & t \geq 0
\end{array}
$$

8 Show that the variant decreases in each iteration

Must show that: $\{P \wedge B \wedge t = T_0\}\ a := a + 1\ \{t < T_0\}$, where B is the guard of the loop.

$$
\begin{array}{ll}
 & wp(a := a + 1, t < T_0) \\
= & \quad \{\text{defn. of weakest precondition}\} \\
 & (t < T_0)\ [a := a + 1] \\
= & \quad \{\text{defn. of } t\} \\
 & (\sqrt{n} - a < T_0)\ [a := a + 1] \\
= & \quad \{\text{substitution}\} \\
 & \sqrt{n} - (a + 1) < T_0 \\
= & \quad \{\text{In the precondition, } t = T_0 = \sqrt{n} - a\} \\
 & \sqrt{n} - (a + 1) < \sqrt{n} - a \\
= & \quad \{\text{Arithmetic and Leibniz}\} \\
 & (\sqrt{n} - a) - 1 < \sqrt{n} - a \\
= & \quad \{\text{Arithmetic } x - 1 < x \text{ for any } x\} \\
 & true
\end{array}
$$