Greatest Common Divider (Eiffel)

JSO

22/06/2020

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

```
// 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

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]

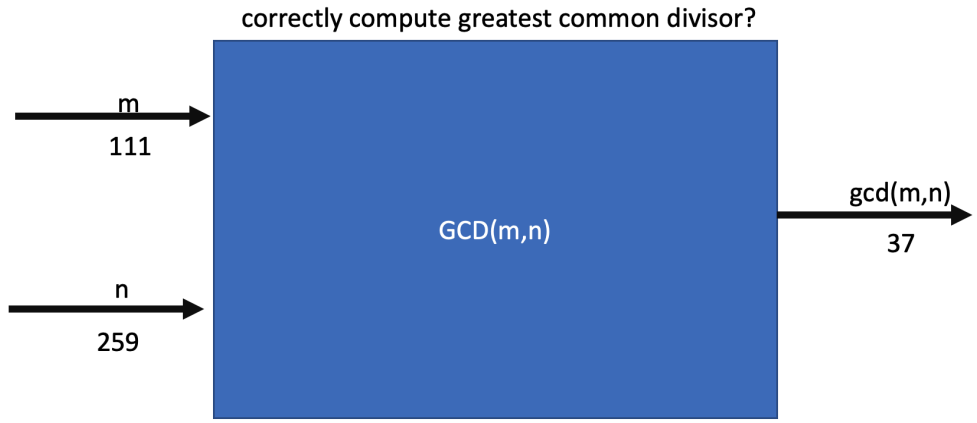
To understand the use of formal methods, we examine a simple example. In the sequel, we provide and explain the text of an Eiffel program that contains specification and implementation for Euclid's GCD algorithm. The implementation can be checked against the specification via runtime assertion checking.[2].

2 Using Formal Methods to Verify Euclid's algorithm

In Fig. 1, Euclid's algorithm is implemented in Golang.[3]

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.[4]

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

[2] See `https://github.com/yuselg/3311-W20-Public/tree/master/euclid/code/eiffel`.

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

[4] 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.

The earlier report (NSF Workshop on Formal Methods for Security) perhaps words it too strongly, but there is more than a grain of truth to it. Here is further quote from that report:

> Formal methods are the only reliable way to achieve security and privacy in computer systems. Formal methods, by modeling computer systems and adversaries, 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 is relative to 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.[5]

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

[5] "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

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:

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	
PASSED	NONE	t0: Check {EUCLID}.divisors(12) = 1, 2, 3, 4, 6, 12
FAILED	Precondition violated.	t1: {EUCLID}.gcd (111, 259) = 37 gcd (-111, 259) results in a precondition violation
PASSED	NONE	t3: exhaustive testing of gcd over 30 x 30 50 x 50: 4.0s workbench vs. 0.1s finalized 200 x 200: 1.1s finalized

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.

4 Hoare Logic

We use the notation of a Hoare triple $\{Q\}S\{R\}$ where Q is a precondition, S is a program statement (i.e. code) and R is a postcondition.

> [HT] Hoare Triple $\{Q\}S\{R\}$: Execution of the program statement S begun in a state satisfying predicate Q must (1) terminate, and (2) terminate in a state satisfying the predicate R.

For example, let S be the assignment statement "$x := x - y$". Then we might write $\{x > 2y\}\ x := x - y\ \{x > y\}$. This Hoare Triple (HT) is *valid*, i.e. execution of $x := x - y$ begun in a state satisfying $x > 2y$ is guaranteed to terminate in a state satisfying $x > y$.

The following HTA (Hoare Triple Assignment) Rule captures this type of logic:

$$\{R[x := exp] \wedge \mathrm{WD}(exp)\}\ x := exp\ \{R\} \qquad \text{(HTA)}$$

In the above, $R[x := exp]$ is a predicate similar to R, except that all free variables in R are replaced with expression exp.[6]

$\mathrm{WD}(exp)$ means that exp must be well defined. For example, the expression $1/x$ is not well-defined if $x = 0$; it is a partial function, not a total function. Thus $\mathrm{WD}(1/x) \equiv x \neq 0$.

[6] Provided there is no illegal capture, see [Tou08]

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
```

In many cases, WD(exp) $\equiv$ $true$; thus, $R[x := exp] \wedge \text{WD}(exp) \equiv R[x := exp]$. We know from elsewhere that $R[x := exp] \wedge \text{WD}(exp)$ is the weakest precondition such that executing $x := exp$ terminates with R true. For simplicity, we will use $R[x := exp]$ as the weakest precondition—and, we only mention WD(exp) in a context in which exp is a partial function. The proof obligation HPA-PO to show that an assignment satisfies its specification is as follows:

> [HTA-PO] To prove that the assignment $x := exp$ satisfies the Hoare specification $\{Q\}\ x := exp\ \{R\}$, it suffices to show that $Q \Rightarrow R[x := exp]$.

5 Proof of Termination and Correctness

In Fig. 4, we provide the fragment of the GCD code with the loop—annotated with Hoare assertion conditions (shown in red):

There are five proof obligations that must be discharged to prove that the loop terminates, and terminates correctly. The first three proof obligations have to do with partial correctness, i.e. if we can assume termination, then:

- the invariant `inv` is established initially;

Figure 4: The GCD loop

```
1  from
2    x, y := m, n -- simultaneous assignment
3    {inv}
4  invariant
5    inv: gcd_spec(x,y) = gcd_spec(m,n)
6  until
7    x = y -- exit codition
8  loop
9    if x < y then
10     {x < y ∧ inv} y := y - x {inv}
11   else -- y < x
12     {y < x ∧ inv} x := x - y {inv}
13   end
14 variant x + y
15 end
16 {x = y ∧ inv}
17 {x = gcd_spec(m, n)}
```

- each execution of the loop preserves the invariant;
- thus, on termination the invariant and the exit condition entail line 17: $x = gcd\_spec(m, n)$.

The last two proof obligations show that the loop terminates.

5.1 Prove that the invariant inv is established initially

By HTA-PO, given that we must prove (see lines 1–5)

```
{true}
x, y := m, n -- simultaneous assignment
{inv}
```

it is sufficient to prove $true \Rightarrow inv[x, y := m, n]$:

$true \Rightarrow inv[x, y := m, n]$

$\equiv$ {propositional logic}

$inv[x, y := m, n]$

$\equiv$ {definition of inv : $gcd\_spec(x, y) = gcd\_spec(m, n)$ and Leibniz}

$(gcd\_spec(x, y) = gcd\_spec(m, n))[x, y := m, n]$

$\equiv$ {simultaneous assignment $gcd\_spec(x, y)[x, y := m, n] = gcd\_spec(m, n)$ and Leibniz}

$gcd\_spec(m, n) = gcd\_spec(m, n)$

$\equiv$ {equality}

$true$ ■

5.2 Prove that each iteration of the loop preserves *inv*

So long as we are in the loop, the negation of the exit condition holds, i.e. $x \neq y$. There are two branches to the conditional:

1. $\{x < y \wedge inv\}$ y := y - x $\{inv\}$
2. $\{y < x \wedge inv\}$ x := x - y $\{inv\}$

We prove each branch separately.

For the first branch, by HTA-PO, it is sufficient to prove $x < y \wedge inv \;\Rightarrow\; inv[y := y - x]$. We start with the consequent:

$$
\begin{array}{ll}
 & inv[y := y - x] \\
\equiv & \quad \{\text{definition of } inv \text{ and Leibniz}\} \\
 & (gcd\_spec(x, y) \;=\; gcd\_spec(m, n))[y := y - x] \\
\equiv & \quad \{\text{assignment of free variables and Leibniz}\} \\
 & gcd\_spec(x, y - x) \;=\; gcd\_spec(m, n) \\
\equiv & \quad \{\text{GCD theorem: } y > x \;\Rightarrow\; gcd\_spec(x, y) = gcd\_spec(x, y - x)\} \\
 & y > x \Rightarrow (gcd\_spec(x, y) \;=\; gcd\_spec(m, n)) \\
\equiv & \quad \{\text{definition of } inv \text{ and Leibniz}\} \\
 & y > x \;\Rightarrow\; inv \qquad \blacksquare
\end{array}
$$

The GCD theorem $y > x \;\Rightarrow\; gcd\_spec(x, y) = gcd\_spec(x, y-x)$ holds because any divisor of x and y is also a divisor of x and $y - x$. We need $y > x$ to ensure that WD($gcd(x, y - x)$), so that $y - x \geq 1$.

The symmetric GCD theorem is $x > y \;\;\Rightarrow\;\; gcd\_spec(x, y) = gcd\_spec(x - y, y)$. Thus the second branch of the conditional can be proved symmetrically with the first.

5.3 Prove that exit condition and invariant entails postcondition

If the loop terminates (line 16), it must terminate with the exit condition true and the invariant must hold (as it has been shown to be preserved by every execution of the loop), i.e. we know: $x = y \;\wedge\; inv$.

We must now show that line 17 (i.e. $x = gcd\_spec(m, n)$) holds. We must thus show that $x = y \wedge inv \;\;\Rightarrow\;\; x = gcd\_spec(m, n)$.

$$
\begin{array}{ll}
 & x = y \;\wedge\; inv \\
\equiv & \quad \{\text{definition of } inv \text{ and Leibniz}\} \\
 & x = y \;\wedge\; gcd\_spec(x, y) = gcd\_spec(m, n) \\
\Rightarrow & \quad \{x = y \text{ and Leibniz}\} \\
 & gcd\_spec(x, x) = gcd\_spec(m, n) \\
\equiv & \quad \{\text{GCD Theorem: } gcd(x, x) = x, \text{ obvious}\} \\
 & x = gcd\_spec(m, n) \qquad \blacksquare
\end{array}
$$

Finally, by HTA-PO, the following trivially holds:

$\{x = gcd\_spec(m, n)\}$
`Result := x`
$\{Result = gcd\_spec(m, n)\}$

We have thus established the postcondition of $gcd(m, n)$.

5.4 Show that the variant t is bounded from below

The variant t is the integer expression $x + y$. So long as we are in the loop, $\neg B$ holds where B is the exit condition of the loop. We must thus show that $inv \wedge \neg B \Rightarrow t \geq 0$. The truth is that we do not need the antecedent. We know that the precondition is $x \geq 1 \wedge y \geq 1$, so we can also trivially prove that $inv2 : x \geq 1 \wedge y \geq 1$ is also a loop invariant.

$$
\begin{array}{ll}
 & inv2 : x \geq 1 \wedge y \geq 1 \\
\Rightarrow & \quad \{\text{arithmetic}\} \\
 & x + y \geq 0 \\
\equiv & \quad \{\text{definition of } t\} \\
 & t \geq 0 \qquad \blacksquare
\end{array}
$$

5.5 Show that the variant decreases in each iteration

Must show that: $\{inv \wedge \neg B \wedge t = T_0\}$ *loop* $\{t < T_0\}$, where B is the exit condition of the loop. In the above, T_0 is an undetermined constant that represents the value of the variant at the beginning of each iteration.

Left as an exercise.

6 Eiffel: Runtime assertion checking

There is a "lightweight" method of checking that the proofs hold for a bounded set of inputs, say $m \in 1 \,..\, 100 \ \wedge \ n \in 1 \,..\, 100$.

The Eiffel runtime will automatically check all the assertions of Section 5. This will provide us with confidence that the loop variant and invariant are well-posed and that the formal verification is feasible.

In addition, any time we execute the query $gcd(m, n)$, all the contracts will be automatically checked, and violations will be reported.

The Eiffel program text with the specification and implementation is provided at `https://github.com/yuselg/3311-W20-Public/tree/master/euclid/code/eiffel`.

References

[Tou08] George Tourlakis. *Mathematical Logic*. Wiley, 2008.