# Specifications and Implementation in Eiffel

Jonathan S. Ostroff [*]

22/06/2020

## Contents

[*]EECS, Lassonde School of Engineering, York University.

## 1 Significance of Specifications and Formal Methods

It is non-trivial to develop reliable software products for safety-critical systems such as medical devices or nuclear power generation, concurrent systems or cybersecurity. For example:

> 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 Example: Specifying and verifying Euclid's algorithm

To understand the use of specifications and formal methods, we examine a simple example. 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*).

In the sequel, we provide 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] The use of Hoare logic (via preconditions and postconditions) was invented at the dawn of software engineering. Design by Contract (DbC) as implemented in languages such as Eiffel and Ada have taken this conceptual machinery and made it applicable to large industrial strength software systems.

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

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

[1] *Report on the NSF Workshop on Formal Methods for Security*, 2016, `https://arxiv.org/pdf/1608.00678.pdf`. Italics not in the original.

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

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

```
// 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.

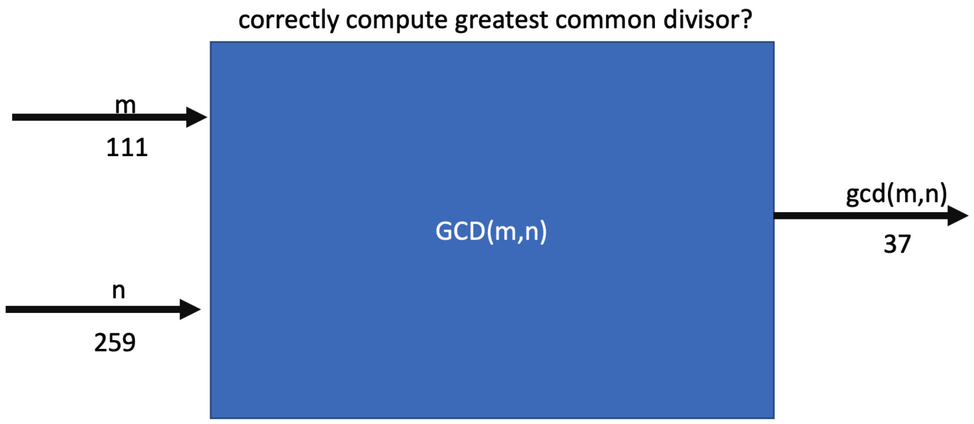
Figure 1: GCD implemented in Go

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]

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.

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

[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.

```
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

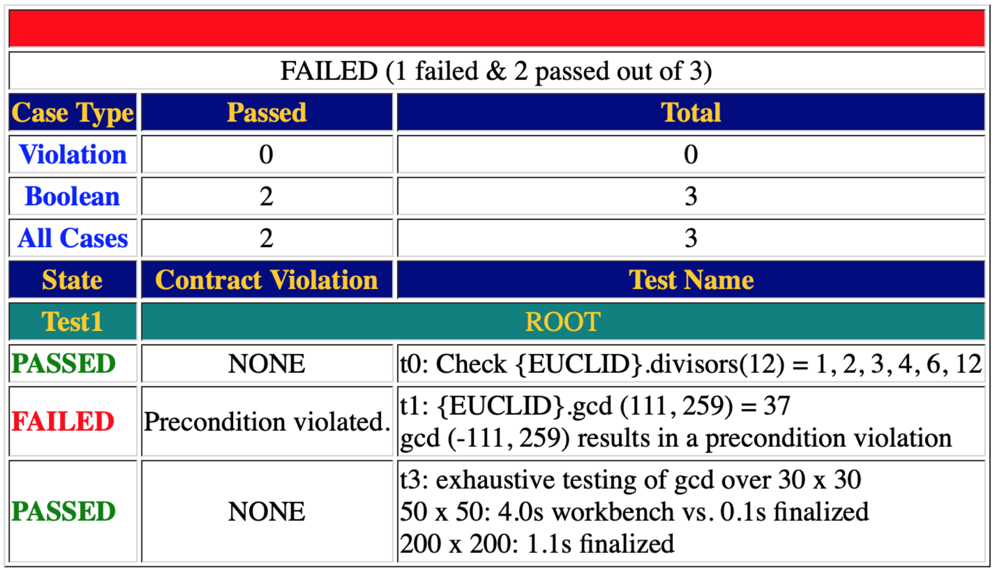
- $divisors(q) \ \widehat{=}\ \{d \in 1 \,..\, q \mid divides(d,q)\}$, where $divides(d,q)$ is true if $d$ divides $q$, i.e. $q \mod d = 0$.
- $gcd\_spec(m,n) \ \widehat{=}\ max(divisors(m) \cap divisors(n))$.

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<br>gcd (-111, 259) results in a precondition violation |
| **PASSED** | NONE | t3: exhaustive testing of gcd over 30 x 30<br>50 x 50: 4.0s workbench vs. 0.1s finalized<br>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:

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

In the above, $R[x := exp]$ is a predicate similar to $R$, except that all free occurrences of variables $x$ in $R$ are replaced with expression $exp$.[5]

[5]Provided there is no illegal capture, see [Tou08]. This can be extended to simultaneous replacement in the obvious manner: $R[x, y := exp1, exp2]$.

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
    $m \geq 1 \wedge n \geq 1$ -- 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 -- $\{y < x \text{ as } x \neq y\}$
        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
```

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 $\mathrm{WD}(1/x) \equiv x \neq 0$.

In many cases, $\mathrm{WD}(exp) \equiv true$; thus, $R[x := exp] \wedge \mathrm{WD}(exp) \equiv R[x := exp]$. We know from elsewhere that $R[x := exp] \wedge \mathrm{WD}(exp)$ 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

To understand a loop, and to prove that it terminates correctly, it is crucial to discover a loop invariant $inv$ (for partial correctness) and loop variant $t$ (an integer valued expression) for termination. We distinguish between partial correctness, which requires that if an answer is returned it will be correct, and total correctness, which additionally requires that the algorithm terminates.[6]

This *separates the concerns* of termination and correctness. There are five proof obligations; the first three prove partial correctness as shown in Fig.4.

Figure 4: Proof obligations for the termination and correctness of a loop

```
0  from
1     {$Q$}
2     init
3     {$inv$}
4  until
5     B -- exit condition
6  loop
7     {$inv \wedge \neg B$}
8     body
9     {$inv$}
10 variant $t$
11 end
12 {$inv \wedge B$}
13 {$R$}
```

1. $\{Q\}$ `init` $\{inv\}$: Given the precondition $Q$, show that the loop invariant is established initially (before execution of the loop begins).
2. $\{inv \wedge \neg B\}$ `body` $\{inv\}$: Show that each execution of the loop body preserves the invariant.
3. $inv \wedge B \Rightarrow R$: On termination (i.e. $B$ holds), the invariant and the exit condition entail the postcondition $R$.
4. $\{inv \wedge \neg B \wedge t = T_0\}$ `body` $\{t < T_0\}$ where $T_0$ is a constant: every execution of the loop results in a decrease of the variant $t$.
5. $inv \wedge \neg B \Rightarrow t \geq 0$: Variant $t$ is bound from below, i.e. every execution of the loop may never cause the variant to become negative.

In Fig. 5, we provide the fragment of the GCD code with the loop—annotated with Hoare assertion conditions (shown in red):

The last two proof obligations show that the loop terminates.

## 5.1 Prove that the invariant *inv* is established initially

We must prove (see lines 1–3):

```
{$m \geq 1 \wedge n \geq 1$}
x, y := m, n -- simultaneous assignment
{$inv$}
```

By HTA-PO, it is sufficient to prove $m \geq 1 \wedge n \geq 1 \Rightarrow inv[x, y := m, n]$. We start with the consequent:

$$\begin{array}{ll}
 & inv[x, y := m, n] \\
\equiv & \quad \{\text{definition of } inv : gcd\_spec(x, y) = gcd\_spec(m, n) \text{ and Leibniz}\} \\
 & (gcd\_spec(x, y) = gcd\_spec(m, n))[x, y := m, n] \\
\equiv & \quad \{\text{simultaneous assignment } gcd\_spec(x, y)[x, y := m, n] = gcd\_spec(m, n) \text{ and Leibniz}\} \\
 & gcd\_spec(m, n) = gcd\_spec(m, n) \\
\equiv & \quad \{\text{equality}\} \\
 & true \\
\Leftarrow & \quad \{\text{propositional logic, strengthening}\} \\
 & m \geq 1 \wedge n \geq 1 \quad \blacksquare
\end{array}$$

[6]Since there is no general solution to the halting problem, there is no guarantee that we can prove total correctness. However, loop variants and invariants have often been provided. See [GS93, Gri85] for deriving invariants for proof-by-construction.

Figure 5: Hoare annotations of the the GCD loop

```
from
   {$precondition : m \geq 1 \wedge n \geq 1$}
   x, y := m, n -- simultaneous assignment
   {$inv$}
invariant
   inv: gcd_spec(x,y) = gcd_spec(m,n)
until
   x = y -- exit condition B
loop
   if x < y then
      {$x < y \wedge inv$} y := y - x {$inv$}
   else -- {$y < x$ as $x \neq y$}
      {$y < x \wedge inv$} x := x - y {$inv$}
   end
variant t: x + y
end
{$x = y \wedge inv$}
{$x = gcd\_spec(m, n)$}
```

## 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:

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

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 $\mathrm{WD}(gcd(x, y - x))$, 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 $x = y \wedge inv \;\Rightarrow\; x = gcd\_spec(m, n)$.

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

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 $gcd(m, n)$.

We have used various theorems about GCD in our proofs. We can prove these theorems using our GCD specification $gcd\_spec(x, y)$. To take a simple example,

$$
\begin{array}{ll}
 & gcd(x, x) \\
= & \qquad \{\text{definition } gcd\_spec(x, y)\} \\
 & max(divisors(x) \cap divisors(x)) \\
= & \qquad \{\text{set theory } A \cap A = A\} \\
 & max(divisors(x)) \\
= & \qquad \{\text{definition of divisors}\} \\
 & max(\{d \in 1\,..\,x \mid divides(d, x)\}) \\
= & \qquad \{\text{definition of } max\} \\
 & x
\end{array}
$$

Thus, $(\forall x \geq 1 : gcd(x,x) = x)$. ■

## 5.4 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.

## 5.5 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.

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

# 6 Eiffel: Exhaustive Runtime Assertion Checking

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

The Eiffel runtime will automatically check all the assertions of Section 5. In addition, any time we execute the query $gcd(m,n)$, all the contracts will be automatically checked, and violations will be reported.[7] This will provide us with confidence that the loop variant and invariant are well-posed and that the formal verification is feasible. Even without further formal verification, this may provide sufficient confidence in the software product beyond testing.

## 6.1 Specification Tests

The specification $gcd\_spec(x,y)$ using Mathmodels is an executable specification (albeit inefficient) of the GCD. This specification can thus act as an Oracle in exhaustive testing as shown in Fig. 6 lines 12–15.

Thus, writing a specification provides a method for writing automated tests beyond that of unit-testing. These concepts can be transferred over to testing written in other languages

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

Figure 6: Exhaustive Specification Test with Oracle gcd_spec (line 15)

```
 1  t3: BOOLEAN
 2    local
 3      euclid: EUCLID
 4      gcd, gcd_spec: INTEGER
 5      k: INTEGER -- iterations
 6    do
 7      comment (t3: exhaustive testing of gcd over 30 x 30)
 8      k := 30 -- use finalized for larger sets
 9      create euclid
10      across 1 |..| k is m loop
11      across 1 |..| k is n loop
12        gcd := euclid.gcd (m, n)
13        gcd_spec := euclid.gcd_spec (m, n)
14        -- specification tests
15        Result := gcd = gcd_spec
16        check Result end
17      end
18      end
19    end
```

as well (depending on their "expressivity"). When the software is deployed contract checking can be turn off for efficiency.

## 7 Why Design by Contract?

- In the program text itself, write *specifications* (not only implementations).
- Documents the contract between client and supplier. See the contract view of the program text in Fig. 7. There is a contract between the client and supplier, each having obligations and benefits. Loosely coupled objects are guaranteed to interact correctly with each other as specified in the contract.
- Verify that the implementation satisfies the *specification*.
- Exhaustively test software products using Specification Tests.
- Executions are raised only when there are contract violations. Avoids code bloat by eliminating the need for constant defensive programming.
- Subcontracting: ensures the Liskov substitution principle so that inheritance is used correctly.

As stated by Bertrand Meyer, Design by Contract can be used throughout the design process.[8]

[8]See [Mey97] for his description of Dbc.

The Eiffel method treats the whole process of software development as a continuum; unifying the concepts behind activities such as requirements, specification, design, implementation, verification, maintenance and evolution; and working to resolve the remaining differences, rather than magnifying them.

Formal specification languages look remarkably like programming languages; to be usable for significant applications they must meet the same challenges: defining a coherent type system, supporting abstraction, providing good syntax (clear to human readers and parsable by tools), specifying the semantics, offering modular structures, allowing evolution while ensuring compatibility.

The same kinds of ideas, such as an object- oriented structure, help on both sides. Eiffel as a language is the notation that attempts to support this seamless, continuous process, providing tools to express both abstract specifications and detailed implementations.[9]

## 8 The Academy and Industry

Security researchers recently disclosed 19 vulnerabilities in a small library designed in the 90s that has been widely used and integrated into countless of enterprise and consumer-grade products over the last 20 years.

The number of impacted products is estimated at “hundreds of millions” and includes products such as smart home devices, power grid equipment, healthcare systems, industrial gear, transportation systems, printers, routers, mobile/satellite communications equipment, data center devices, commercial aircraft devices, various enterprise solutions, and many others.[10]

From the description in the article, the three most serious vulnerabilities seem to be buffer overflows. C and C++ can be dangerous languages. For these devices, perhaps we should be using languages with strong static typing guarantees (e.g. Rust, Ada etc.). Even so, there is a way to use C safely if we have mastered disciplined programming methods discussed in this article. Such errors are easily avoidable but new vulnerabilities will continue to be built into products until programmers change the way they write and verify software.

As pointed out by safety experts, “thousands of development teams have incorporated these library routines in their products and, unsurprisingly, failed to find the vulnerabilities in their testing. Yet today, thousands of development teams will continue to resist using better methods, tools and languages”.

Lesley Lamport writes that engineers draw detailed plans before a brick is laid or a nail is hammered. Programmers don’t. Can this be why houses seldom collapse and programs often crash?

Blueprints help engineers and architects to ensure that what they are planning to build will work. “Working” means more than not collapsing; it means serving the required purpose. Engineers and their clients use blueprints to understand what they are going to build before

[9] `https://bertrandmeyer.com/2014/12/07/lampsort/`.

[10] `https://www.zdnet.com/article/ripple20-vulnerabilities-will-haunt-the-iot-landscape-for-years-to-come/`. Accessed 22 June 2020.

they start building it. But few programmers write even a rough sketch of what their programs will do before they start coding.[11]

### 8.1 Foundational Principles in CS/software engineering education

> 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.[12]

## References

[Gri85] David Gries. *The Science of Programming.* Springer-Verlag, 1985.

[GS93] David Gries and Fred B. Schneider. *A Logical Approach to Discrete Math.* Springer Verlag, 1993.

[Mey97] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1997.

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

## A Contract View of class EUCLID for GCD

[11] `https://channel9.msdn.com/Events/Build/2014/3-642`. Lesley Lamport is the author of TLA+ and winner of the Turing Award in 2013: `https://amturing.acm.org/award_winners/lamport_1205376.cfm`.

[12] "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.

Figure 7: Contract View of class EUCLID

```
class EUCLID feature

  gcd (m, n: INTEGER): INTEGER
      -- return the gcd of `m` and `n`
    require
      non_zero: m >= 1 and n >= 1
    ensure -- $Result = max(divisors(m) \cap divisors(n))$
      gcd_spec: Result = max(divisors(m) |/\| divisors(n))

feature -- gcd-spec

  divisors (n: INTEGER): SET [INTEGER]
      -- return set of divisors of `q`
    require
        n >= 1
    ensure -- $Result = \{d \in 1 \mathinner{..} q \mid is\_divisible(q, d)\}$
      divisors_set: Result ~ (range (1, n) | agent is_divisible (n, ?))

  gcd_spec (m, n: INTEGER): INTEGER
      -- specification definition of gcd
    ensure
        Result = max (divisors (m) |/\| divisors (n))

  max (s: SET [INTEGER]): INTEGER
    require
        not s.is_empty
    ensure -- $(Result \in s) \wedge (\forall i \in s \mid Result \geq i)$
        contains: s.has (Result)
        is_max: across s is i all Result >= i end
end
```