Computación: Dos visiones sobre el desarrollo de software

CAMILO CHACÓN SARTORI

La computación o informática tiene múltiples definiciones; algunas hacen referencia a una ciencia; otras a una ingeniería; algunos mencionan a que es simplemente un arte; y otros dicen que es una combinación de todas las anteriores. Independientes de estas definiciones, hay un concepto que es central en todas: *algoritmos*. Por lo tanto, podría decir que la informática es el estudio de los algoritmos computacionales los cuales se implementan en un artefacto (computador) con el fin de resolver problemas. Y, la actividad que implementa estos algoritmos se denomina: *programación*. Existe dos formas de comprobar si un algoritmo cumple con una especificación: (1) realizando *testing* y (2) a través de una verificación formal. En este breve artículo trataré sobre las dos visiones contrapuestas de ver la verificación formal en un software: (1) No podemos lograr la formalidad en la programación debido a los continuos cambios en el desarrollo y a la complejidad de realizarla; y (2) la verificación formal es la forma rigurosa de formalizar un algoritmo utilizando matemáticas lo cual reduce la cantidad de errores en un software.

Palabras claves: informática, computación, programación, algoritmos, corrección de software, verificación formal

1 INTRODUCCIÓN

La verificación formal nos permite comprobar si un algoritmo cumple con una especificación. Antes de eso, cabe preguntarse ¿qué es un algoritmo? Una definición muy simple sería: una secuencia ordenada de instrucciones computacionales que resuelven un problema. Donde la computación se trata del estudio de algoritmos. Entonces, ¿qué es el estudio de algoritmos? Es la forma de verificar la calidad y eficiencia de estos.

Si comenzamos a revisar las diversas áreas de la computación, tales como: sistemas operativos, base de datos, ingeniería de software, sistemas distribuidos, optimización, compiladores, visión por computadora, aprendizaje de máquina, etc. cada una de estas tiene algo en común: el uso de algoritmos para fundamentar su base de conocimiento. Hay una diversa cantidad de algoritmos. En algunos casos usan un mayor trasfondo estadístico (aprendizaje de máquina); más extenso en el uso de funciones geométricas (visión por computadora); basada en teoría de conjuntos (base de datos); donde la mantenibilidad de software y manejo de equipos de programadores es fundamental (ingeniería de software); etc.

La programación como tal, esta más relacionada a la ingeniería de software; la cual hace referencia al diseño, escalabilidad y calidad de un proyecto de software. Que abarca más conceptos que solo la escritura de código. Y, un software (programa computacional) es un conjunto de algoritmos que podrían interactuar entre sí para resolver uno o más problemas. (A su vez, la programación es realizada por programadores; los cuales resuelven problemas e implementan soluciones [software] a través de la escritura de código.)

Existen dos visiones de desarrollo de software que están contrapuesta. La primera, que no es necesario una formalidad en la especificación de los algoritmos dada su complejidad y, por tanto, se debe incentivar el *testing*. La segunda, es posible mejorar la correctitud de un algoritmo mediante el uso de sistemas formales que permiten reducir los errores.

Este artículo está organizado de la siguiente forma. Sección 2 trata sobre el tema en controversia: la verificación formal de algoritmos; Sección 3 y 4 expone cada una de las visiones contrapuestas; Finalmente, en la sección 5 realizo mis conclusiones.

Author's address: Camilo Chacón Sartori, camilochs@gmail.com.

2 VERIFICACIÓN FORMAL

La verificación de software es una disciplina de la ingeniería de software que esta dividida en dos categorías: estática y dinámica. La verificación formal es parte de la estática y, la dinámica hace referencia al conocido proceso de *testing* para encontrar errores (*bug* en inglés) en un software (aunque esta última se podría ver más como una *validación* que una verificación). La verificación formal es un área que trata sobre la *correctitud* de un algoritmo; es decir, hace referencia a si un algoritmo cumple o no con su especificación. Un algoritmo es correcto si cumple lo siguiente: (1) hace lo que dice la especificación; (2) los valores entradas generan los *correctos* valores de salida; y (3) se ejecuta en un tiempo finito (es decir, evita caer en el problema de la parada [se refiere a que no se puede determinar si un algoritmo entrara en un ciclo infinito el cual nunca terminaría de ejecutarse]).

Esta área fue introducida por Robert W. Floyd en su artículo «Assigning Meanings to Programs» (W. Floyd, 1967), por medio de aserciones lógicas se prueba la validez de un algoritmo usando pruebas de correctitud, equivalencia y terminación. El mismo autor menciona en dicho artículo: «This paper attempts to provide an adequate basis for formal definitions of the meanings of programs in appropriately defined programming languages» [Este documento intenta proporcionar una base adecuada para las definiciones formales de los significados de los programas en los lenguajes de programación adecuadamente definidos].

Cada instrucción de un algoritmo es comprobada a través de expresiones lógicas (aserciones). Es decir, podemos comprobar la validez de un algoritmo a nivel de la especificación sin necesidad de ejecutarlo para saber si retornara el valor correcto. Es por esto, que se trata de una verificación estática.

2.1 Lógica de Hoare

Tony Hoare en su artículo «An Axiomatic basis for computer programming» (Hoare, 1969), introdujo un sistema formal con un conjunto de reglas lógicas aplicadas sobre declaraciones en un lenguaje determinado para verificar la correctitud de un algoritmo (se le conoce como lógica de Hoare o lógica de Floyd-Hoare). Básicamente una forma de axiomatizar un algoritmo. Se le dice «Hoare Triple» a la formula: $\{\phi_1\}A\{\phi_2\}$, donde A es un algoritmo definido en algún lenguaje formal imperativo, ϕ_1 es una precondición y ϕ_2 una postcondición. (En la literatura se usa la palabra «program» [que vendría siendo un software] en vez de «algoritmo» dado que se enfoca más en un conjunto de estos últimos, para simplificar utilizaré solamente «algoritmo».)

Por ejemplo, podemos crear un simple lenguaje imperativo utilizando la notación BNF:

⟨*variable*⟩ ::= [a-z]+

⟨*number*⟩ ::= [0-9]+

⟨*Expr*⟩ ::= ⟨*variable*⟩
| ⟨*number*⟩
| ⟨*Expr*⟩ + ⟨*Expr*⟩
| ⟨*Expr*⟩ - ⟨*Expr*⟩
| ⟨*Expr*⟩ * ⟨*Expr*⟩
| ...

⟨*Condition*⟩ ::= ⟨*Expr*⟩ '==' ⟨*Expr*⟩
| '!' ⟨*Condition*⟩
| ⟨*Condition*⟩ 'and' ⟨*Condition*⟩

| ⟨*Condition*⟩ ‘`or`’ ⟨*Condition*⟩

| ...

⟨A⟩ ::= ⟨*variable*⟩ ‘`:=`’ ⟨*number*⟩ | ⟨A⟩; ⟨A⟩

| ‘`if`’ ⟨*Condition*⟩ ‘`begin`’ ⟨A⟩ ‘`else`’ ⟨A⟩ ‘`end`’

| ‘`while`’ ⟨*Condition*⟩ ‘`begin`’ ⟨A⟩ ‘`end`’

| ...

Esta gramática puede soportar las siguientes expresiones:

(a)

```
v := 10;
if v == 10 begin
    v := 0;
else
    v := v + 1;
end
```

(b)

```
y := 5;
x := y * x + 1;
```

A continuación, veamos algunos ejemplos —bastante triviales— pero instructivos, donde podemos verificar un posible defecto en un algoritmo usando la lógica de Hoare:

(1) $\{x > 0\} \quad x := x * 2; \quad \{x \geq 2\}$

(2) $\{x == y - 1\} \quad x := x + 1; \quad y := y + 2; \quad \{x == y\}$

(3) $\{x == y == z\} \quad x := y + 1; \quad z := x + 1; \quad \{(x > y) < z\}$

(4) $\{(x > y) \wedge (x \geq 0) \wedge (y > 0)\} \quad if\ x > 0\ begin \quad x := y - 1;\ else \quad x := y * -2;\ end \quad \{x < y\}$

Cada uno de los ejemplos anteriores son validos. Siempre que la precondición sea verdadera la postcondición debe cumplirse, en caso contrario, tenemos un error en nuestro algoritmo. Por ejemplo en (1), si reemplazamos x por cualquier número mayor a 0, el valor de x después de la ejecución del algoritmo debe ser mayor o igual a 2. Y, dado que el algoritmo incrementa x por su doble, esto siempre sera verdadero. Lo mismo ocurre para los ejemplos siguientes. En caso contrario, si la precondición o la postcondición no se cumple, entonces el algoritmo no cumple con la verificación y por tanto no es valido.

La lógica de Hoare nos permite pensar en las condiciones lógicas de los valores de entrada y salida. Es decir, podemos definir correctamente el estado previo y posterior a la ejecución del algoritmo.

Regla de Composición. Por otro lado, también podemos hacer uso de reglas de inferencias. Una de ellas es la regla de composición, la cual tiene la siguiente estructura:

$$\frac{\{\phi_1\} \quad A_1 \quad \{\phi_2\} \quad , \quad \{\phi_2\} \quad A_2 \quad \{\phi_3\}}{\{\phi_1\} \quad A_1; \quad A_2; \quad \{\phi_3\}}$$

Si tenemos dos axiomas en la parte superior: $\{\phi_1\} \quad A_1 \quad \{\phi_2\}$ y $\{\phi_2\} \quad A_2 \quad \{\phi_3\}$ podemos inferir la siguiente regla: $\{\phi_1\} A_1; \quad A_2; \{\phi_3\}$. Esto quiere decir que, si las instrucciones superiores son verdaderas por inferencia las inferiores también lo serán.

Las siguientes instrucciones cumplen con esta regla:

$$\frac{\{x > 0\} \quad x := x + 5; \quad \{x \geq 5\} \quad , \quad \{x \geq 5\} \quad x := x + 1; \quad \{x \leq 10\}}{\{x > 0\} \quad x := x + 5; \quad x := x + 1; \quad \{x \leq 10\}}$$

Regla de Iteraciones. Está la podemos usar cuando necesitamos comprobar la correctitud de un algoritmo que usa ciclos (*while*). Nos permite comprobar si una variable se mantiene invariante[1], es decir, si su estado se mantiene *igual* antes y después del ciclo.

$$\frac{\{\phi_1 \wedge \phi_2\} \quad A_1 \quad \{\phi_1\}}{\{\phi_1\} \quad while \quad \phi_2 \quad begin \quad A_1 \quad end \quad \{\neg\phi_2 \wedge \phi_1\}}$$

Esta regla quiere decir que ϕ_1 es la expresión lógica que se mantiene invariante antes y después de A_1, por otro lado, ϕ_2 es la expresión que permite terminar el ciclo (evitando así un ciclo infinito). Un ejemplo de esta regla es la siguiente:

$$\frac{\{y < 10 \wedge (x > 0 \wedge x < 10)\} \quad y := y + 1; x := x * 2; \quad \{y < 10\}}{\{y < 10\} \quad while \quad x > 0 \wedge x < 10 \quad begin \quad y := y + 1; x := x * 2; \quad end \quad \{\neg(x > 0 \wedge x < 10) \wedge y < 10\}}$$

Podemos ver en el ejemplo superior que el axioma ($x > 0 \wedge x < 10$) es ϕ_2, se evaluara en el *while* hasta que x sea igual o superior a 10. Por tanto, la variable y ira incrementando pero seguirá siendo invariante a la expresión: $y < 10$.

2.2 Herramientas para Verificación Formal

Las siguientes herramientas permiten realizar verificación formal en distintos lenguajes:

- Boogie[2]: Es un lenguaje de verificación formal intermedio. Permite utilizar otros lenguajes diseñados para la verificación (por ejemplo, Havoc para C y Spec# para C#).
- Why3[3]: Es una plataforma para verificación de software. La cual provee un lenguaje llamado Why3ML que también puede servir como intermediario a otros lenguajes de verificación.
- TLA^+[4]: Es un lenguaje de verificación formal similar a las matemáticas. Especialmente utilizado para algoritmos distribuidos y concurrente. Según el mismo sitio web «It's the software equivalent of a blueprint.»[Esta es equivalente a un plano en el software].

3 LOS PROBLEMAS DE LA VERIFICACIÓN FORMAL

En el artículo: «Social Processes and Proofs of Theorems and Programs» (De Millo et al., 1979) los autores mencionan que la verificación formal no debería jugar un rol fundamental en la ingeniería de software, por las siguientes razones:

- Dado que la verificación formal es una forma de acercar la matemática a la programación, los teoremas matemáticos no son infalibles y se ha demostrado (según ellos) que pueden presentar contradicciones. Es decir, con el solo hecho de usar matemática no asegura correctitud.
- Usa como ejemplo un algoritmo probabilista que realiza un test de primalidad (dado un número entero retorna si es primo o no) llamado: Miller-Rabin; dado que utiliza componentes aleatorios argumentan que no es posible realizar una comprobación matemática (que es exacta por definición).
- Al existir software que son muy complejos y la verificación formal puede ser extensa en su definición. Significa que, no es aplicable al desarrollo de software en el mundo real.

[1] Invariante es el concepto fundamental en la verificación formal, nos permite comprobar si una expresión lógica mantiene su estado después de un conjunto de transformaciones.
[2] https://www.microsoft.com/en-us/research/project/boogie-an-intermediate-verification-language/
[3] http://why3.lri.fr/
[4] https://learntla.com/introduction/

- La verificación formal no puede dominar el diseño de software porque nada puede ser perfecto. En el entorno empresarial es normal encontrarse con *deadlines*; la búsqueda del costo/beneficios; grupos de trabajo con diversos estilos; la aceptación y limitación de posibles errores; hacen un peso más importante que la verificación.

La visión De Millo, Lipton y Perlis es bastante dura hacia la verificación formal, no le da ningún punto favorable y mencionan que debería evitarse porque la perfección no existe en el desarrollo de software del «mundo real».

4 LA IMPORTANCIA DE LA VERIFICACIÓN FORMAL

Dos grandes exponentes de este planteamiento son: Edsger Dijkstra y Leslie Lamport. Dos ganadores del prestigioso premio: Turing Award[5]. Los dos apelan a la importancia de las matemáticas en la programación; porque según ellos ésta es una habilidad fundamental para que un programador pueda pensar mejor un problema algorítmico, le da mayor rigurosidad, que se traduce en una disminución de errores en el software que se construya.

De hecho, el mismo Dijkstra fue critico del artículo presentado en la sección anterior (3). En la cual se realiza una critica a la verificación formal. Dijkstra en su artículo EWD638 «A political pamphlet from the Middle Ages» (Dijkstra, 1978) y a su propio estilo (que no era principalmente muy amable), comienza diciendo «Esta nota hace referencia a un artículo muy feo [...] ellos hacen una caricatura de la verificación formal [...] luego venden el gran mensaje que la comunidad de ciencia de la computación se ha equivocado. Eso es lo que yo llamo, el estilo de un folleto político.» Más adelante dice: «ellos simplemente ignoran como realizar una demostración matemática.» y «no distinguen entre el amor a la perfección y la pretensión de perfección, y culpan a la gente por la primera acusándola de la segunda.» concluyendo con:

«Muestran la misma actitud precientífica cuando argumentan como si un puente y un sistema de software fueran objetos significativamente similares: leyendo el texto solo se puede concluir que esta opinión ha sido inducida por la similitud verbal entre los términos "Ingeniería Mecánica" e "Ingeniería del Software".»

Dijkstra asume que debido a un error semántico todo el texto es un error y que ellos no saben lo que realmente *es* la formalización.

Este último párrafo es importante porque varios años después Lamport escribiría su artículo «Viewpoint who builds a house without drawing blueprints?» (Lamport, 2015) donde desarrolla la idea de construir software debería ser similar a un plano, es decir, el equivalente a lo que hace un arquitecto. Aunque él hace una salvedad: la verificación formal no elimina todos los problemas, por ejemplo, si los requerimientos son incorrectos nos llevaran a equivocaciones en nuestra verificación, eso significa que aún tenemos que tener un entorno de pruebas *testing* (pruebas), a pesar de eso: la verificación formal ayuda a *pensar*. Con respecto a esto último, termina su artículo con la siguiente frase: «Pensar no garantiza que no cometerás errores, pero no pensar si los asegura». Lamport durante su carrera a sido muy critico en decir que los problemas del desarrollo de software son por una falta de conocimiento matemático de los informáticos, es decir, una falta de rigurosidad en la definición de como elaboran su tarea: diseño e implementación de algoritmos. Esta idea fue detallada en su escrito: «If You're Not Writing a Program, Don't Use a Programming Language» (Lamport, 2018).

En general, Dijkstra y Lamport consideran que la verificación formal es importante, no perfecta, pero en ningún caso ésta debería omitirse como los expuesto por De Millo, Lipton y Perlis.

[5] Otorgado a personas que realizan un gran aporte a la ciencia de la computación, este es entregado por ACM y es el equivalente al Nobel en la computación.

5 CONCLUSIÓN

La verificación formal es un mecanismo que permite especificar rigurosamente lo que debe hacer un algoritmo, utilizando como herramienta las matemáticas. Es por esto que, primero explique qué es la verificación formal, utilizando el método presentado por (Hoare, 1969). Después la visión anti-verificación es analizada según el trabajo escrito por De Millo, Lipton y Perlis. Los cuales presentan una critica a ésta dando las siguientes razones: la formalidad matemática no es perfecta; no vale la pena porque puede ocupar demasiado tiempo haciéndola impracticable; diversos factores en el entorno de trabajo no hace factible que se utilice en el «mundo real». Por otro lado, tenemos a Dijkstra y Lamport que están a favor de una mayor rigurosidad en la programación. Mencionan que la verificación formal enseña a pensar y por lo mismo ayuda a reducir errores en el desarrollo de software.

Ahora, con respecto a mi opinión, la verificación formal en estos días no es ocupada en la mayoría de entornos de desarrollo de software, lo cual es lamentable, especialmente cuando se debe desarrollar algoritmos que realizan tareas criticas. A mayor nivel de importancia en la tarea que debe resolver un algoritmo mayor debería ser el uso de formalidad matemática previa a la implementación en un lenguaje de programación; y, del *testing* en la fase posterior. Por el contrario, según se puede ver en la industria todos los problemas se solucionan con entornos de *testing* sin pensar adecuadamente en como formalizar un algoritmo. Y, lo peor, es que en algunas empresas no realizan *testing*, por tanto, menos aún la verificación formal. Una pena.

REFERENCIAS

R. A. De Millo, R. J. Lipton, and A. J. Perlis. Social processes and proofs of theorems and programs. *Commun. ACM*, 22(5):271–280, May 1979. ISSN 0001-0782. doi: 10.1145/359104.359106. URL http://doi.acm.org/10.1145/359104.359106.

E. Dijkstra. On a political pamphlet from the middle ages. *SIGSOFT Softw. Eng. Notes*, 3(2):14–16, Apr. 1978. ISSN 0163-5948. doi: 10.1145/1005888.1005890. URL http://doi.acm.org/10.1145/1005888.1005890.

C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, Oct. 1969. ISSN 0001-0782. doi: 10.1145/363235.363259. URL http://doi.acm.org/10.1145/363235.363259.

L. Lamport. Who builds a house without drawing blueprints? *Commun. ACM*, 58(4):38–41, Mar. 2015. ISSN 0001-0782. doi: 10.1145/2736348. URL http://doi.acm.org/10.1145/2736348.

L. Lamport. If you're not writing a program, don't use a programming language. 2018. URL http://bulletin.eatcs.org/index.php/beatcs/article/view/539.

R. W. Floyd. Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967.