Introducción a la Verificación de Programas

» Ricardo Rosenfeld
Centro de Altos Estudios en Tecnología Informática, Universidad Abierta Interamericana

RESUMEN

Iniciamos una serie de cuatro artículos introductorios sobre la verificación axiomática de programas, en el marco de los programas imperativos de entrada/salida. En este artículo introducimos el método axiomático de verificación para los programas secuenciales determinísticos, y desarrollamos ejemplos de aplicación. Si bien la verificación de programas se expone como una actividad a posteriori (dados un programa y una especificación, verificar que el programa satisface la especificación), la idea que se sostiene en el artículo, y en toda la serie, es tener en cuenta los axiomas y reglas del método para programar al mismo tiempo que verificar, de modo tal de obtener programas correctos por construcción. Con esta perspectiva, al final se ejemplifica un desarrollo sistemático de programa basado en la axiomática presentada.

PALABRAS CLAVE: PROGRAMA, VERIFICACIÓN, AXIOMÁTICA

Introduction to Program Verification

ABSTRACT

We start a series of four introductory articles regarding the axiomatic verification of programs, in the context of imperative input/output programs. In this article, we introduce the axiomatic verification method for deterministic sequential programs, and we show some application examples. Although program verification is presented as an a posteriori activity (given a program and a specification, verify that the program satisfies the specification), the main idea sustained in the article, and in the entire series, is to take into account the method axioms and rules both for programming as well as verifying, in order to obtain correct programs by construction. With this perspective, a systematic development of a program based on the studied axiomatics is presented at the end.

KEYWORDS: PROGRAM, VERIFICATION, AXIOMATICS

Recibido: enero 2022 Aceptado: Mayo 2022 Publicado: Junio 2022

1. Introducción

Este artículo es el primero de una serie de cuatro trabajos introductorios sobre la verificación de programas, elaborados en el ámbito de un proyecto en curso del Centro de Altos Estudios en Tecnología Informática (CAETI) de la Universidad Abierta Interamerica (UAI). El objetivo del proyecto es la creación de un *framework* para asistir en el desarrollo de software, centrado en conceptos avanzados de modularización y síntesis de comportamiento [MS11, ABKS13, DBPU13]. Acompañamos la implementación con estas publicaciones, con la idea de reforzar conceptos fundamentales de la teoría de correctitud de programas, teniendo en cuenta distintos paradigmas de programación.

Concretamente, describiremos el clásico *método axiomático de verificación de programas*. Verificar un programa consiste en probar formalmente sus propiedades con respecto a su especificación. En la verificación de un programa intervienen necesariamente dos artefactos formales, una especificación y un programa, sin ambigüedades, descriptos mediante lenguajes con sintaxis y semántica precisas. De esta manera la verificación contrasta con la validación de un programa, cuya actividad más representativa es el testing: tal como dijera E. Dijkstra, uno de los más importantes pioneros de la programación estructurada, el testing asegura la presencia de errores pero no su ausencia.

¿Cómo verificar un programa? La manera ideal es construirlo y verificarlo al mismo tiempo, es decir, partir de su especificación, y en base a una metodología bien definida, a un cálculo, derivarlo, de modo tal que satisfaga la especificación, y así obtener un programa correcto por construcción [Dij76, Gri81, CM88]. Otra aproximación aceptada consiste en ir transformando la especificación en artefactos equivalentes a través de la utilización de reglas de reescritura, hasta llegar al lenguaje objeto: es el caso de la verificación *a priori* [BD77, BW82]. Contrariamente, la verificación *a posteriori* (dados una especificación y un programa, probar que el programa es correcto con respecto a la especificación) no es el camino apropiado, pero como manera de exposición resulta altamente didáctico, lo que explica su profusión en la bibliografía existente. Por eso presentaremos el método axiomático con esta modalidad, pero acentuando la idea de utilizar los axiomas y reglas del método como guía para el desarrollo sistemático de programas.

Consideraremos una familia de programas muy representativos: los programas imperativos. Dichos programas se caracterizan por transformar estados (variables con sus contenidos) mediante las instrucciones que los componen. Además, acotaremos el estudio a los programas de entrada/salida. En esta primera entrega nos limitaremos a los programas secuenciales determinísticos, con los que se presentó originariamente el método axiomático [Flo67, Hoa69, Apt81]. El término axiomático se debe a que el método toma como marco de referencia los sistemas deductivos de la lógica de predicados [Ham88, Men15, PRS17]. Se plantea un sistema con axiomas y reglas de inferencia asociados a las instrucciones de los lenguajes de programación considerados, y así las pruebas son conducidas por la estructura de los programas. El alcance de esta metodología se amplió enseguida a los programas no determinísticos y concurrentes [OG76a, OG76b, AFdR80, Apt84], que trataremos en los trabajos siguientes. No mucho después surgió el *model checking*, alternativa al *approach* que desarrollaremos (identificado en la literatura como *theorem proving*), que consiste en la automatización de la verificación siempre que los programas sean modelizables con sistemas finitos de transiciones de estados [MP89, MP92, HR04, CES09]. En los últimos años también por el lado del *theorem proving* se han implementado numerosos productos que asisten

Revista Abierta de Informática Aplicada/vol 6 Nº1 (2022): 79-100
ISSN 2591-5320

computacionalmente en la verificación de software, tanto en la industria como en la academia [HS00, FS06, KPJ06, Jan07, WPN08, CHT15, ASFS18, SGS21].

El artículo continúa de la siguiente manera. En la sección 2 se presentan las características generales del método de verificación: se introducen sus componentes, se especifica la notación utilizada a lo largo del trabajo y se definen las propiedades de programas que se van a considerar, la *correctitud parcial* y la *terminación*. Las secciones 3 y 4 describen, respectivamente, los axiomas y reglas para las pruebas de dichas propiedades. En la sección 5 se desarrolla un ejemplo de programación sistemática basada en el método estudiado. Finalmente, la sección 6 cierra el artículo con observaciones finales.

2. Características generales del método de verificación

Trabajaremos con programas escritos en un lenguaje muy simple del tipo Pascal [HW73], para aprovechar las buenas prácticas de la programación estructurada. La sintaxis formal del lenguaje de programación, descripta en notación Backus-Naur (o BNF), es la siguiente:

$$S :: \text{skip} \mid x := e \mid S_1 ; S_2 \mid \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi} \mid \text{while } B \text{ do } S_1 \text{ od}$$

Los subíndices no forman parte del lenguaje, se utilizan para facilitar la definición. La expresión *e* de la asignación *x := e* es del tipo entero (por simplicidad, restringiremos así el dominio de las variables y usaremos variables simples, excepcionalmente arreglos), y la expresión B de las instrucciones de selección condicional *if then else* y de repetición *while* es del tipo booleano. Las expresiones tienen la siguiente forma (mostramos algunas):

$$e :: n \mid x \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 . e_2 \mid \ldots \mid \text{if } B \text{ then } e_1 \text{ else } e_2 \text{ fi}$$
$$B :: \text{true} \mid \text{false} \mid e_1 = e_2 \mid e_1 < e_2 \mid e_1 > e_2 \mid \ldots \mid \neg B \mid B_1 \vee B_2 \mid B_1 \wedge B_2 \mid \ldots$$

Nuevamente, los subíndices no forman parte del lenguaje. n es una constante entera, x es una variable entera, y *true* (por verdadero) y *false* (por falso) son las constantes booleanas. Completamos la descripción del lenguaje de programación presentando, informalmente, la semántica de sus instrucciones:

La instrucción *skip* es atómica (se consume en un paso), y no tiene ningún efecto sobre las variables. Se puede usar, por ejemplo, en una selección condicional sin instrucción alternativa para el *else*.

- La *asignación x := e* también es atómica. Asigna el valor de *e* a la variable *x*.
- La *secuencia* $S_1 ; S_2$ ejecuta S_1 y luego S_2.
- La *selección condicional if B then* S_1 *else* S_2 *fi* ejecuta S_1 si B es verdadera o S_2 si B es falsa.
- Finalmente, la *repetición while B do S od* ejecuta S mientras B sea verdadera (evalúa B, si B es verdadera ejecuta S y repite el ciclo, si B es falsa termina).

Por ejemplo, el siguiente programa obtiene en las variables c y r, respectivamente, el cociente y el resto de la división entera entre dos números naturales x e *y*, mediante la técnica de las restas

sucesivas (históricamente, este programa fue el primero que se probó utilizando el método axiomático):

$$S_{div} :: \; c := 0 \; ; \; r := x \; ; \; \text{while } r \geq y \text{ do } r := r - y \; ; \; c := c + 1 \; \text{od}$$

Por otro lado, para especificar formalmente los programas utilizaremos el lenguaje de la lógica de predicados, cuya sintaxis es (mostramos algunas formas de predicados):

$$p :: true \mid false \mid e_1 = e_2 \mid e_1 < e_2 \mid e_1 > e_2 \mid \ldots \mid \neg p \mid p_1 \vee p_2 \mid p_1 \wedge p_2 \mid \ldots \mid \exists x: p \mid \forall x: p$$

La especificación de un programa se va a expresar como un par de predicados (p, q), la *precondición* y la *postcondición*, asociados a la entrada y la salida del programa, respectivamente. La idea es que la especificación establezca la relación que debe existir entre los contenidos de las variables al inicio y al final del programa. Por ejemplo, la especificación:

$$(x = X, x = 2.X)$$

será satisfecha por un programa que duplique su entrada x. La variable X se conoce como *variable de especificación* (o *lógica*), no es una variable de programa, y está implícitamente cuantificada universalmente, es decir que contiene cualquier valor del dominio (en este caso un número entero). Sirve para fijar valores (lo que no es posible con las variables de programa), y así su uso permite vincular de alguna manera a la precondición con la postcondición. Supongamos ahora que se pretende especificar un programa que termine con la condición x > y. Una posible especificación sería:

$$(x = X \wedge y = Y, x > y)$$

y otra más simple, ya que por convención el predicado *true* representa cualquier condición, y en este caso no es necesario indicar qué valores iniciales tienen x e *y*:

$$(true, x > y)$$

Además de las especificaciones y los programas, en la presentación del método de verificación tenemos que introducir el concepto de *estado*. Los estados son las distintas configuraciones por las que transita la computación de un programa. En términos matemáticos, un estado es una función:

$$\sigma : Var \rightarrow Z$$

donde Var es el conjunto de variables del programa considerado y Z es el conjunto de los números enteros (recordar que nos limitamos a este dominio). De esta manera, por lo mencionado antes, el predicado *true* denota el conjunto de todos los estados (y por oposición, el predicado *false* denota el conjunto vacío de estados). Generalizando, un predicado denota un conjunto determinado de estados. Se suele utilizar la expresión:

Revista Abierta de Informática Aplicada/vol 6 N°1 (2022): 79-100
ISSN 2591-5320

$$\sigma \models p$$

para establecer que el estado σ pertenece al conjunto de estados denotado por el predicado p. Otra manera de expresar lo mismo es que σ satisface p, o que p evaluado en σ es verdadero. Por ejemplo, volviendo al predicado anterior p = x > y, si en un estado σ se cumple que x = 7 e y = 5, entonces σ |= p. En este caso, el predicado p denota el conjunto de todos los estados en que x es mayor que *y*.

Naturalmente, cuantos más estados iniciales denote una precondición (lo que equivale a decir en términos lógicos cuanto más *débil* sea la precondición), mayor alcance de aplicabilidad tendrá el programa en cuestión. A su vez, cuanto más *fuerte* en términos lógicos sea la postcondición, más precisa será la especificación del resultado del programa. De esta manera podemos referirnos a la *precondición más débil* y a la *postcondición más fuerte* de un programa.

Introducidos los componentes anteriores, llegamos a la definición de la verificación (de la correctitud) de un programa S con respecto a una especificación (p, q): verificar que S satisface (p, q) consiste en probar que a partir de la precondición p, el programa S termina en la postcondición q. En concreto, hay que probar dos cosas:

1. A partir de p, si S termina lo hace en q.
2. A partir de p, S termina.

La primera propiedad se conoce como *correctitud parcial*. La segunda propiedad es la t*erminación*. Si el programa cumple las dos propiedades, se dice que tiene *correctitud total* (en definitiva, lo que buscamos probar). Cabe destacar en este punto un par de comentarios:

- La separación en las dos propiedades no es caprichosa, porque se prueban de una manera distinta. La correctitud parcial (debe su nombre al concepto de función parcial) se prueba *inductivamente*, mientras que por el contrario la terminación no está asociada a una demostración por inducción. Esto quedará más claro en las secciones siguientes.

- Asumimos por simplicidad que cuando un programa termina lo hace en un estado correcto. Por ejemplo, en el caso de una división por cero consideraremos directamente que el valor resultante es un número entero y no un valor indefinido. Otra alternativa, que no adoptaremos en este trabajo, sería considerar un estado final incorrecto, lo que es usual cuando se trata con programas no determinísticos o concurrentes, en que la correctitud total reúne más propiedades (por ejemplo, un programa concurrente con *deadlock* se considera que termina incorrectamente).

Vamos a expresar las dos propiedades mediante las siguientes fórmulas de correctitud, conocidas como *ternas de Hoare*:

1. {p} S {q} para la correctitud parcial: a partir de la precondición p, si el programa S termina lo hace en la postcondición q.
2. ⟨p⟩ S ⟨true⟩ para la terminación: a partir de la precondición p, el programa S termina.

Consistentemente con lo anterior, la correctitud total de un programa S con respecto a una especificación (p, q) se expresa con la fórmula ⟨p⟩ S ⟨q⟩, la cual entonces es equivalente a ({p} S {q}) ∧ (⟨p⟩ S ⟨true⟩). Por ejemplo, son verdaderas (o correctas) las fórmulas:

$$\{x = 0\}\ x := x + 1\ \{x = 1\}$$

$$\langle x = 10\rangle \text{ while } x \neq 0 \text{ do } x := x - 1 \text{ od } \langle x = 0\rangle$$

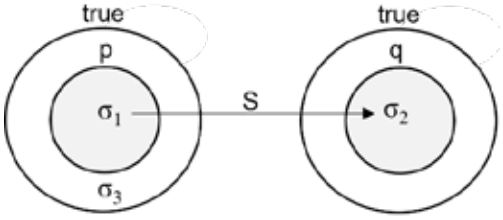
También la fórmula:

$$\{true\}\ S\ \{true\}$$

porque independientemente de la forma del programa S, si S no termina a partir de un determinado estado no se viola la correctitud parcial. En cambio, la fórmula:

$$\langle true\rangle\ S\ \langle true\rangle$$

no es correcta, lo que puede demostrarse con el siguiente contraejemplo. Dado el programa *S :: while x ≠ 0 do x := x – 1 od*, si al comienzo vale x < 0 el programa no termina. Con que haya un estado que satisface la precondición de un programa, a partir del cual el programa no termina o termina en un estado que no satisface la postcondición, alcanza para que el programa no sea totalmente correcto respecto de la especificación. En cambio, nada se puede afirmar cuando el estado inicial de un programa no satisface la precondición establecida, porque dicho estado está fuera del alcance de aplicación del programa. La siguiente figura ilustra cuándo se cumple la correctitud total:



El predicado *true* denota el conjunto de todos los estados. Desde un estado σ_1 del conjunto de estados denotado por la precondición p, S debe terminar en un estado σ_2 del conjunto de estados denotado por la postcondición q. No interesa el comportamiento de S a partir de un estado σ_3 que está fuera del alcance de p.

3. Verificación de la correctitud parcial

El método axiomático de verificación de programas se basa en reglas formales de prueba, que también se pueden utilizar como base para una metodología de construcción de programas correctos. Esto es extensible a la verificación de terminación tratada en la sección siguiente. Las reglas proveen el significado, la semántica de cada una de las instrucciones, estableciendo correspondencias apropiadas con pre y postcondiciones. Siendo un programa S una secuencia de instrucciones especificada por un par de predicados (p, q), aplicando las reglas instrucción por

Revista Abierta de Informática Aplicada/vol 6 N°1 (2022): 79-100
ISSN 2591-5320

instrucción y valiéndonos de la propiedad de *composicionalidad* del método llegamos a probar la fórmula {p} S {q} planteada.

Dicho lo último de otra manera, cada instrucción se comporta como un *transformador de predicados* (y por lo tanto de estados), y así el programa completo actúa como tal. Que el método sea composicional significa que la fórmula {p} S {q} se puede probar componiendo de determinada manera un conjunto de fórmulas $\{p_1\}\ S_1\ \{q_1\}$, $\{p_2\}\ S_2\ \{q_2\}$, …, $\{p_n\}\ S_n\ \{q_n\}$, ignorando el contenido de los subprogramas S_i de S (se toman como cajas negras). Lamentablemente esta propiedad se pierde en el paradigma concurrente.

Emulando a los métodos deductivos de la lógica, los componentes del método de verificación se denominan *axiomas*, para definir la semántica de las instrucciones atómicas, y *reglas de inferencia*, o simplemente *reglas*, para definir la semántica de las instrucciones compuestas. Los presentamos a continuación:

1. *Axioma del skip* (SKIP): {p} skip {p}

El *skip* no tiene efecto alguno sobre el estado inicial, consume un paso y termina en el mismo estado. Así, si un predicado se cumple antes de su ejecución, sigue valiendo después.

2. *Axioma de la asignación* (ASI): {p[x|e]} x := e {p}

Este axioma se lee así: si se cumple el predicado p en términos de x después de la ejecución de una asignación *x := e*, significa que antes de la asignación se cumplía p en términos de la expresión *e*. p[x|e] denota el reemplazo de toda ocurrencia libre (no ligada a un cuantificador) de la variable x en el predicado p por la expresión *e*. Por ejemplo, una particular instanciación del axioma puede ser:

$$\{x + 1 \geq 0\}\ x := x + 1\ \{x \geq 0\}$$

De esta manera, el axioma ASI se lee de atrás para adelante, de derecha a izquierda, lo que impone en cierto modo una forma de probar desde la postcondición hasta la precondición. De hecho, está visión coincide con el cálculo de programas definido por E. Dijkstra: conociendo la postcondición que se espera de un programa, la idea es construirlo valiéndonos de los mecanismos de transformación de predicados del lenguaje de programación utilizado, y obtener su precondición más débil, es decir, el conjunto de estados iniciales más amplio correspondiente al programa.

Resulta más natural plantear un axioma ASI que se lea de izquierda a derecha. El más simple sería:

$$\{true\}\ x := e\ \{x = e\}$$

Pero esta fórmula no es verdadera. Notar que si la expresión *e* incluye a la variable x puede haber problemas. Por ejemplo, si e = x + 1, la fórmula obtenida es falsa:

$$\{true\}\ x := x + 1\ \{x = x + 1\}$$

El problema radica en que la x de la parte derecha de la postcondición se refiere a la variable antes de la asignación, mientras que la x de la parte izquierda se refiere a la variable luego de la asignación. Existe en la literatura una forma correcta del axioma ASI de adelante para atrás, que justamente distingue sintácticamente las dos partes de la asignación:

$$\{p\}\ x := e\ \{\exists z:\ p[x|z] \wedge x = e[x|z]\}$$

Consideraremos la primera forma del axioma, que es más simple y la más difundida.

3. *Regla de la secuencia* (SEC): $$\frac{\{p\}\ S_1\ \{r\}\ ,\ \{r\}\ S_2\ \{q\}}{\{p\}\ S_1\ ;\ S_2\ \{q\}}$$

Del cumplimiento de $\{p\}\ S_1\ \{r\}$ y $\{r\}\ S_2\ \{q\}$ se deriva $\{p\}\ S_1\ ;\ S_2\ \{q\}$. El predicado intermedio r actúa como nexo para probar la secuencia de S_1 con S_2 y luego se descarta, es decir que no se propaga a lo largo de la prueba. Mediante un razonamiento inductivo se puede utilizar también una generalización de la regla a cualquier número de premisas:

$$\frac{\{p\}\ S_1\ \{r_1\}\ ,\ \{r_1\}\ S_2\ \{r_2\}\ ,\ \ldots\ ,\ \{r_{n-1}\}\ S_n\ \{q\}}{\{p\}\ S_1\ ;\ S_2\ ;\ \ldots\ ;\ S_n\ \{q\}}$$

4. *Regla del condicional* (COND): $$\frac{\{p \wedge B\}\ S_1\ \{q\}\ ,\ \{p \wedge \neg B\}\ S_2\ \{q\}}{\{p\}\ \text{if B then } S_1 \text{ else } S_2 \text{ fi } \{q\}}$$

Esta regla impone una manera de verificar una selección condicional estableciendo un único punto de entrada y un único punto de salida, correspondientes a la precondición p y la postcondición q, respectivamente. A partir de p, se cumpla o no la expresión B, luego de la ejecución correspondiente de S_1 o S_2 vale q, según lo establecido por las premisas.

5. *Regla de la repetición* (REP): $$\frac{\{p \wedge B\}\ S\ \{p\}}{\{p\}\ \text{while B do S od } \{p \wedge \neg B\}}$$

REP se centra en un predicado *invariante*, que debe cumplirse al comienzo del *while* y luego de toda iteración de su cuerpo. Si mientras vale la expresión B, la ejecución de S preserva el predicado p, entonces si la repetición termina se cumple $p \wedge \neg B$. Claramente esta regla no asegura la terminación del *while*, fuente única de infinitud en los programas del lenguaje definido. Su forma explicita que la correctitud parcial es una propiedad que se prueba inductivamente. La inducción involucrada se conoce como *computacional*: si un predicado se cumple al comienzo de una computación y todos sus pasos lo preservan, entonces el predicado es un invariante de la computación y como tal se cumple hasta el final.

6. *Regla de consecuencia* (CONS): $$\frac{p \rightarrow p_1\ ,\ \{p_1\}\ S\ \{q_1\}\ ,\ q_1 \rightarrow q}{\{p\}\ S\ \{q\}}$$

Revista Abierta de Informática Aplicada/vol 6 N°1 (2022): 79-100
ISSN 2591-5320

Esta regla se diferencia de las anteriores porque no se relaciona con ninguna instrucción, sino directamente con el dominio semántico, en este caso los números enteros. Entenderemos mejor su uso en los ejemplos que desarrollaremos enseguida. Notar que por su forma, algunos pasos de una prueba serán directamente verdades matemáticas. Desde el punto de vista lógico, la regla permite reforzar precondiciones y debilitar postcondiciones, preservando la correctitud de una fórmula. Por ejemplo, si se cumplen la implicación $r \rightarrow p$ y la fórmula $\{p\}\ S\ \{q\}$, entonces aplicando la regla CONS también se va a cumplir $\{r\}\ S\ \{q\}$ (r es una precondición más fuerte que p, todo estado que satisface r también satisface p, o dicho de otro modo, el conjunto de estados denotado por r está incluido en el conjunto de estados denotado por p). En particular, la regla permite sustituir predicados por otros equivalentes, siempre interpretados en el dominio semántico de referencia.

7. *Regla de instanciación* (INST): $\dfrac{f(X)}{f(c)}$

La regla INST también es una regla semántica. f es una fórmula de correctitud. La variable X es una variable de especificación, representa cualquier valor del dominio, en este caso un número entero. c es algún elemento del dominio de X. La regla es necesaria para instanciar fórmulas de correctitud. Por ejemplo, $\{x = X\}\ x := x + 1\ \{x = X + 1\}$ se instancia en $\{x = 5\}\ x := x + 1\ \{x = 5 + 1\}$ cuando $X = 5$. Hay que tener en cuenta que si X aparece en la pre y postcondición, no puede instanciarse con una expresión que contenga una variable de programa. Por ejemplo, si en la fórmula anterior se reemplaza X con x, se obtiene la fórmula falsa $\{x = x\}\ x := x + 1\ \{x = x + 1\}$. Al igual que CONS, INST es una regla universal, y de hecho algunos autores las omiten asumiendo implícitamente su utilización.

El conjunto planteado de axiomas y reglas es suficiente para probar axiomáticamente cualquier programa con la sintaxis descripta. Se dice que el método es *completo*. Además, el método asegura que sólo se van a obtener fórmulas de correctitud verdaderas. En este caso se dice que el método es *sensato*. Sobre la sensatez y completitud del método diremos algo más en las observaciones finales. A pesar de la completitud, igual que como se acostumbra a hacer en la lógica, existen más reglas de verificación para facilitar las pruebas. Algunos ejemplos habituales en la bibliografía existente son los siguientes:

8. *Regla de la disyunción* (OR): $\dfrac{\{p\}\ S\ \{q\}\ ,\ \{r\}\ S\ \{q\}}{\{p \vee r\}\ S\ \{q\}}$

Se pueden considerar más premisas. Esta regla permite una prueba por casos. Evita tener que propagar información para establecer oportunamente una disyunción en la precondición. Una forma particular de la regla, bastante común, es:

$$\frac{\{p \wedge r\}\ S\ \{q\}\ ,\ \{p \wedge \neg r\}\ S\ \{q\}}{\{p\}\ S\ \{q\}}$$

9. *Regla de la conjunción* (AND): $\dfrac{\{p_1\}\ S\ \{q_1\}\ ,\ \{p_2\}\ S\ \{q_2\}}{\{p_1 \wedge p_2\}\ S\ \{q_1 \wedge q_2\}}$

También se pueden considerar más premisas. Esta regla permite una prueba incremental. Evita tener que propagar información para establecer oportunamente conjunciones.

10. *Axioma de invariancia* (AINV): $\{p\}\ S\ \{p\}$, si $change(S) \cap free(p) = \emptyset$

change(S) es el conjunto de variables modificables por S, y free(p) es el conjunto de variables libres de p. Naturalmente, si estos conjuntos son disjuntos la ejecución de S no altera p. Este axioma en realidad es más habitual que se emplee en combinación con la regla AND, para dar lugar a la siguiente regla:

11. *Regla de invariancia* (RINV): $\dfrac{\{p\}\ S\ \{q\}}{\{r \wedge p\}\ S\ \{r \wedge q\}}$, si $change(S) \cap free(r) = \emptyset$

Completamos la sección con una serie de ejemplos de pruebas de correctitud parcial de programas empleando los axiomas y reglas descriptos.

Ejemplo 1. Correctitud parcial de un programa de swap de dos variables

El programa siguiente intercambia los contenidos de las variables x e *y*:

$$S_{swap} :: z := x\ ;\ x := y\ ;\ y := z$$

Vamos a probar $\{x = X \wedge y = Y\}\ S_{swap}\ \{y = X \wedge x = Y\}$. Por la forma del programa, recurrimos al axioma de la asignación (ASI) tres veces, una por cada asignación, y al final completamos la prueba utilizando la (generalización de la) regla de la secuencia (SEC):

1. $\{z = X \wedge x = Y\}\ y := z\ \{y = X \wedge x = Y\}$	(ASI)
2. $\{z = X \wedge y = Y\}\ x := y\ \{z = X \wedge x = Y\}$	(ASI)
3. $\{x = X \wedge y = Y\}\ z := x\ \{z = X \wedge y = Y\}$	(ASI)
4. $\{x = X \wedge y = Y\}\ z := x\ ;\ x := y\ ;\ y := z\ \{y = X \wedge x = Y\}$	(1, 2, 3, SEC)

Al igual que en una prueba lógica, enumeramos los pasos e indicamos en cada uno qué axioma o regla utilizamos y sobre qué pasos anteriores se aplica. Notar cómo el axioma ASI impone una forma de prueba desde la postcondición hasta la precondición. Obviamente, también se cumple la misma fórmula pero permutando por ejemplo los operandos de la precondición, es decir:

$$\{y = Y \wedge x = X\}\ z := x\ ;\ x := y\ ;\ y := z\ \{y = X \wedge x = Y\}$$

Para probar esta fórmula tenemos que recurrir a la regla de consecuencia (CONS), agregándole a la prueba anterior los siguientes dos pasos:

Revista Abierta de Informática Aplicada/vol 6 N°1 (2022): 79-100
ISSN 2591-5320

5. $(y = Y \wedge x = X) \rightarrow (x = X \wedge y = Y)$ (MAT)
6. $\{y = Y \wedge x = X\}$ z := x ; x := y ; y := z $\{y = X \wedge x = Y\}$ (4, 5, CONS)

El paso 5 introduce directamente un predicado verdadero, por eso se justifica con el indicador MAT (matemática). Al estar las variables de especificación X e Y implícitamente cuantificadas universalmente, se establece entonces que el programa S_{swap} se comporta adecuadamente cualesquiera sean los valores iniciales de las variables x e *y*. Si quisiéramos instanciar la última fórmula con valores específicos, por ejemplo X = 1 e Y = 2, tenemos que aplicar la regla de instanciación (INST):

7. $\{y = 2 \wedge x = 1\}$ z := x ; x := y ; y := z $\{y = 1 \wedge x = 2\}$ (6, INST)

Ejemplo 2. Correctitud parcial de un programa que calcula el valor absoluto

El programa siguiente obtiene en la variable *y* el valor absoluto de la variable x:

$$S_{va} :: \text{ if } x > 0 \text{ then } y := x \text{ else } y := -x \text{ fi}$$

Probaremos $\{true\}\ S_{va}\ \{y \geq 0\}$. Notar que $(true, y \geq 0)$ no es una especificación correcta de un programa que calcula el valor absoluto. Por ejemplo, *S :: y := 0* satisface la especificación y no es el programa pretendido. No nos detendremos en esto, lo que nos interesa es ejemplificar el uso de la regla del condicional (COND). Considerando las asignaciones del programa S_{va}, efectuamos primero los siguientes dos pasos:

1. $\{x \geq 0\}$ y := x $\{y \geq 0\}$ (ASI)
2. $\{-x \geq 0\}$ y := – x $\{y \geq 0\}$ (ASI)

Para poder aplicar la regla COND se necesita contar con un par de fórmulas con precondiciones de la forma $p \wedge B$ y $p \wedge \neg B$, siendo B la expresión booleana del *if then else*. Haciendo $B = x > 0$, y p = true, la prueba se completa de la siguiente manera:

3. $(true \wedge x > 0) \rightarrow x \geq 0$ (MAT)
4. $(true \wedge \neg(x > 0)) \rightarrow -x \geq 0$ (MAT)
5. $\{true \wedge x > 0\}$ y := x $\{y \geq 0\}$ (1, 3, CONS)
6. $\{true \wedge \neg(x > 0)\}$ y := – x $\{y \geq 0\}$ (2, 4, CONS)
7. {true} if $x > 0$ then y := x else y := – x fi $\{y \geq 0\}$ (5, 6, COND)

A partir de la precondición *true* (cualquier estado), las alternativas $x > 0$ y $\neg(x > 0)$ relacionadas con la variable de entrada x determinan que se ejecute la asignación $y := x$ o la asignación $y := -x$, respectivamente, alcanzando siempre un estado final que satisface la postcondición $y \geq 0$.

Ejemplo 3. Correctitud parcial del programa que calcula la división entera

En este último ejemplo consideramos el uso de la regla de la repetición (REP), para verificar el programa de división entera entre dos números naturales presentado en una sección anterior. Su forma era:

$$S_{div} :: \; c := 0 \; ; \; r := x \; ; \; \text{while } r \geq y \text{ do } r := r - y \; ; \; c := c + 1 \text{ od}$$

S_{div} devuelve en las variables c y r, respectivamente, el cociente y el resto de la división entera entre las variables x e *y*, en base al método de las restas sucesivas. Se va a probar $\{x \geq 0 \wedge y > 0\}\ S_{div}\ \{x = y \,.\, c + r \wedge r \geq 0 \wedge r < y\}$. Nuevamente relajamos la especificación: en este caso la imprecisión es que no se asegura que al final x e *y* tengan los mismos valores que al comienzo, y así el resultado puede no ser el esperado.

Para probar el *while*, la regla REP requiere definir un invariante p, que debe valer al comienzo de la instrucción y después de cada iteración (desde la perspectiva metodológica de construcción de programas correctos, primero se debe concebir el invariante y luego escribir el *while* que lo plasma en términos del lenguaje de programación utilizado). Usaremos el predicado $p = (x = y \,.\, c + r \wedge r \geq 0)$. Como se observa, p generaliza (debilita) la postcondición, heurística habitual. Claramente el invariante refleja la esencia del algoritmo, la invariancia de $x = c \,.\, y + r$ toda vez que el cociente c se incrementa en 1 y el resto r se decrementa en *y*. Para mayor claridad, estructuraremos la verificación en tres partes. Además, para acortar la prueba (y las siguientes), en adelante en la aplicación de la regla CONS daremos por sobrentendido el enunciado verdadero utilizado. El plan de prueba es el siguiente:

a. Las inicializaciones del programa conducen al invariante por primera vez.
   $\{x \geq 0 \wedge y > 0\}$
   $c := 0 \; ; \; r := x \; ;$
   $\{x = y \,.\, c + r \wedge r \geq 0\}$

b. El cuerpo del *while* preserva el invariante, y al final, si se llega, no vale $r \geq y$.
   $\{x = y \,.\, c + r \wedge r \geq 0\}$
   $\text{while } r \geq y \text{ do } r := r - y \; ; \; c := c + 1 \text{ od}$
   $\{x = y \,.\, c + r \wedge r \geq 0 \wedge \neg(r \geq y)\}$

c. Aplicando las reglas SEC y CONS se completa la prueba.

En lo que sigue desarrollamos cada parte de la prueba:

Prueba de (a).

1. $\{x = y \,.\, c + x \wedge x \geq 0\}\ r := x\ \{x = y \,.\, c + r \wedge r \geq 0\}$ (ASI)
2. $\{x = y \,.\, 0 + x \wedge x \geq 0\}\ c := 0\ \{x = y \,.\, c + x \wedge x \geq 0\}$ (ASI)
3. $\{x = y \,.\, 0 + x \wedge x \geq 0\}\ c := 0 \; ; \; r := x\ \{x = y \,.\, c + r \wedge r \geq 0\}$ (1, 2, SEC)
4. $\{x \geq 0 \wedge y > 0\}\ c := 0 \; ; \; r := x\ \{x = y \,.\, c + r \wedge r \geq 0\}$ (3, CONS)

Prueba de (b).

5. $\{x = y \,.\, (c + 1) + r \wedge r \geq 0\}\ c := c + 1\ \{x = y \,.\, c + r \wedge r \geq 0\}$ (ASI)
6. $\{x = y \,.\, (c + 1) + (r - y) \wedge (r - y) \geq 0\}\ r := r - y\ \{x = y \,.\, (c + 1) + r \wedge r \geq 0\}$ (ASI)
7. $\{x = y \,.\, (c + 1) + (r - y) \wedge (r - y) \geq 0\}\ r := r - y \; ; \; c := c + 1\ \{x = y \,.\, c + r \wedge r \geq 0\}$ (5, 6, SEC)
8. $\{x = y \,.\, c + r \wedge r \geq 0 \wedge r \geq y\}\ r := r - y \; ; \; c := c + 1\ \{x = y \,.\, c + r \wedge r \geq 0\}$ (7, CONS)
9. $\{x = y \,.\, c + r \wedge r \geq 0\}\ \text{while } r \geq y \text{ do } r := r - y \; ; \; c := c + 1 \text{ od}\ \{x = y \,.\, c + r \wedge r \geq 0 \wedge \neg(r \geq y)\}$ (8, REP)

Revista Abierta de Informática Aplicada/vol 6 N°1 (2022): 79-100
ISSN 2591-5320

Prueba de (c).

10. $\{x \geq 0 \wedge y > 0\}\ S_{div}\ \{x = y \,.\, c + r \wedge r \geq 0 \wedge \neg(r \geq y)\}$ (4, 9, SEC)
11. $\{x \geq 0 \wedge y > 0\}\ S_{div}\ \{x = y \,.\, c + r \wedge r \geq 0 \wedge r < y\}$ (10, CONS)

La prueba no asegura la terminación del programa, la cual trataremos en la sección siguiente. Aprovechamos este último ejemplo para describir una presentación alternativa de una prueba: la *proof outline* (o *esquema de prueba*). Consiste en intercalar los pasos de la prueba (algunos o todos) entre las instrucciones del programa considerado. Se obtiene así una prueba más estructurada, que documenta adecuadamente el programa. En la verificación de los programas concurrentes las *proof outlines* son imprescindibles. Por ejemplo, la siguiente es una *proof outline* (de correctitud parcial) del programa S_{div} que acabamos de verificar:

$\{x \geq 0 \wedge y > 0\}$
$c := 0 ; r := x ;$
$\{x = y \,.\, c + r \wedge r \geq 0\}$
while $r \geq y$ do
$\{x = y \,.\, c + r \wedge r \geq 0 \wedge r \geq y\}$
$r := r - y ; c := c + 1$
$\{x = y \,.\, c + r \wedge r \geq 0\}$
od
$\{x = y \,.\, c + r \wedge r \geq 0 \wedge r < y\}$

Una *proof outline* muestra qué predicados se cumplen en qué lugares de un programa (en los que se comportan entonces como invariantes, porque siempre se cumplen en esos lugares, independientemente del estado inicial del programa). Es común insertar sólo los predicados relevantes, mínimamente la precondición, los invariantes de las repeticiones y la postcondición.

4. Verificación de la terminación

El *while* es la única instrucción que puede provocar infinitud, y la regla de la repetición (REP) no asegura su terminación. Por eso el método de verificación tiene una regla más, que es en realidad una extensión de REP, tiene más premisas. La llamaremos *regla de la terminación* (REP*). Mantiene la idea de un invariante p, y agrega un *variante* t, que es una función entera definida en términos de las variables de programa. La forma de la regla es la siguiente (continuamos con la numeración de la sección anterior, y utilizamos los delimitadores ⟨ ⟩):

12. *Regla de la terminación* (REP*):
$$\frac{\begin{array}{l}1)\ \langle p \wedge B\rangle\ S\ \langle p\rangle \\ 2)\ \langle p \wedge B \wedge t = Z\rangle\ S\ \langle t < Z\rangle \\ 3)\ \ p \rightarrow t \geq 0\end{array}}{\langle p\rangle \text{ while } B \text{ do } S \text{ od } \langle p \wedge \neg B\rangle}$$

Z es una variable de especificación que no figura en p ni t. Su objetivo es guardar el valor de t previo a la ejecución de S. La primera premisa es la misma que la de REP. Las otras dos premisas son las que aseguran la terminación:

- Por la segunda premisa, la función t se decrementa en cada iteración.
- Por la tercera premisa, t arranca y se mantiene positiva después de cada iteración.

De esta manera el *while* necesariamente debe terminar, porque los valores de t se reducen de iteración en iteración y nunca son negativos. Claramente la postcondición es $p \wedge \neg B$, como en REP. Notar la relación entre el invariante p y el variante t: mientras valga $p \wedge B$, la ejecución de S decrementa t, y además p asegura que t nunca se hace negativo. El valor inicial de t representa la cantidad máxima de iteraciones. Su decrecimiento paulatino indica el acercamiento a la finalización del *while*. Por eso la terminación no se prueba por inducción, no hay una noción de propiedad que se preserva a lo largo de una computación, sino de un evento que se producirá a futuro. La prueba se fundamenta en la inexistencia de cadenas descendentes infinitas en el dominio N de los números naturales con respecto a la relación $<$. El par $(N, <)$ es un ejemplo de *orden parcial bien fundado*: conjunto con una relación antirreflexiva, antisimétrica y transitiva, sin cadenas que decrecen infinitamente. Diremos algo más sobre esto en la última sección.

Notar también que REP* permite verificar directamente $\langle p \rangle$ while B do S od $\langle p \wedge \neg B \rangle$, pero la aplicación de la regla siempre implica llevar a cabo dos pruebas, una prueba inductiva basada en la primera premisa y otra prueba que considera las dos premisas restantes y se relaciona con un orden parcial bien fundado. La recomendación es partir la prueba en dos, como hemos venido haciendo, probar $\{p\}$ while B do S od $\{p \wedge \neg B\}$ por un lado y por el otro $\langle p \rangle$ while B do S od $\langle true \rangle$, utilizando invariantes distintos. El invariante para la prueba de terminación suele ser más simple, en general no tiene que considerar todas las variables que incluye el invariante de la prueba de correctitud parcial. Esto lo podemos apreciar en el siguiente ejemplo, que cierra la sección:

Ejemplo 4. Terminación del programa que calcula la división entera

Completamos la prueba de correctitud total del programa de división entera S_{div}. Ya probamos:

$\{x \geq 0 \wedge y > 0\}$
$c := 0 ; r := x ;$ while $r \geq y$ do $r := r - y ; c := c + 1$ od
$\{x = y \,.\, c + r \wedge r \geq 0 \wedge r < y\}$

Falta probar la terminación del programa, desde la misma precondición $x \geq 0 \wedge y > 0$. Como antes, estructuramos la verificación en una parte asociada al fragmento de inicialización de variables y una parte correspondiente a la repetición. El predicado que enlaza las dos partes de la prueba es el invariante de la repetición. Delimitamos los predicados con $\langle\ \rangle$, y utilizamos el axioma SKIP de la forma $\langle p \rangle$ skip $\langle p \rangle$, el axioma ASI de la forma $\langle p[x|e] \rangle$ $x := e$ $\langle p \rangle$, y así todas las reglas salvo la de repetición. El plan de prueba queda de la siguiente manera (p es el invariante a determinar):

a. $\langle x \geq 0 \wedge y > 0 \rangle$ $c := 0 ; r := x$ $\langle p \rangle$
b. $\langle p \rangle$ while $r \geq y$ do $r := r - y ; c := c + 1$ od $\langle true \rangle$
c. Por SEC se llega a completar la prueba

Para la prueba de (b), la regla REP* requiere definir un invariante y un variante (desde la perspectiva metodológica, invariante y variante están primero, a partir de ellos se escribe la repetición, no al revés). Como variante usaremos t = r. Notar que r arranca con $x \geq 0$, y se decrementa en cada iteración en $y > 0$ unidades mientras no sea menor que *y*, por lo que en algún momento se

Revista Abierta de Informática Aplicada/vol 6 N°1 (2022): 79-100
ISSN 2591-5320

alcanza r < y, lo que provoca la salida del *while*. En cuanto al invariante, alcanza con que contenga información que contribuya a la prueba de terminación: p = (r ≥ 0 ∧ y > 0).

La prueba de (a) no reviste mayor dificultad. Para la prueba de (b) hay que aplicar la regla REP*, y así probar las siguientes premisas:

1) ⟨r ≥ 0 ∧ y > 0 ∧ r ≥ y⟩ r := r – y ; c := c + 1 ⟨r ≥ 0 ∧ y > 0⟩
2) ⟨r ≥ 0 ∧ y > 0 ∧ r ≥ y ∧ r = Z⟩ r := r – y ; c := c + 1 ⟨r < Z⟩
3) (r ≥ 0 ∧ y > 0) → r ≥ 0

lo que también se resuelve con facilidad. La conclusión de REP* determina:

⟨r ≥ 0 ∧ y > 0⟩ while r ≥ y do r := r – y ; c := c + 1 od ⟨r ≥ 0 ∧ y > 0 ∧ ¬(r ≥ y)⟩

y por CONS logramos:

⟨r ≥ 0 ∧ y > 0⟩ while r ≥ y do r := r – y ; c := c + 1 od ⟨true⟩

En las pruebas de correctitud parcial y de terminación del programa de división S_{div} hemos utilizado invariantes distintos. En el primer caso usamos $p_1 = (x = y \,.\, c + r \wedge r \geq 0)$, y en el segundo, $p_2 = (r \geq 0 \wedge y > 0)$. A ambos predicados se logró llegar a partir de la precondición y las instrucciones iniciales de S_{div}. Este mecanismo de prueba se puede generalizar a un programa con más de un *while*: para cada uno se definen los invariantes y el variante correspondiente para probar su correctitud total, los invariantes debiendo ser implicados por la postcondición de la instrucción previa al *while*.

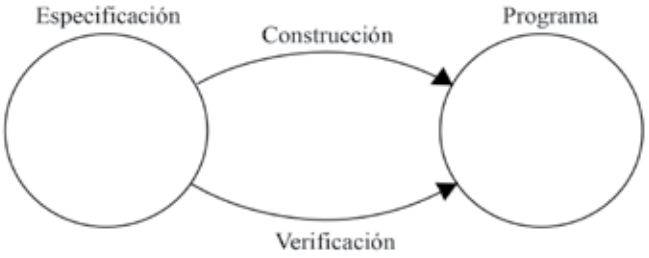
Las *proof outlines* de terminación se definen de la misma manera que las de correctitud parcial, pero cambiando los delimitadores { } por ⟨ ⟩ y agregando los variantes de los *while*. Una *proof outline* correspondiente a la prueba desarrollada recién es:

⟨x ≥ 0 ∧ y > 0⟩
c := 0 ; r := x ;
⟨inv: r ≥ 0 ∧ y > 0 , var: r⟩
while r ≥ y do
⟨r ≥ 0 ∧ y > 0 ∧ r ≥ y⟩
r := r – y ; c := c + 1
⟨r ≥ 0 ∧ y > 0⟩
od
⟨true⟩

Para que haya terminación debe cumplirse y > 0. Con precondición x ≥ 0 ∧ y ≥ 0 en lugar de x ≥ 0 ∧ y > 0, la correctitud parcial de S_{div} sigue valiendo. El predicado x ≥ 0 ∧ y ≥ 0 es la precondición *liberal* más débil (no se asegura terminación, sólo correctitud parcial), y el predicado x ≥ 0 ∧ y > 0 es la precondición *estricta* más débil (en este caso se asegura la correctitud total).

5. Desarrollo sistemático de programas

En los últimos 50 años se ha escrito, experimentado y evolucionado sobremanera, educativa e industrialmente, en las áreas de diseño y construcción de software. Distintas metodologías para derivar programas, como las basadas en los *refinamientos sucesivos* [Wir73], los *tipos abstractos de datos* [LZ74], las *estructuras de datos jerárquicas* [Jac75], la *precondición más débil* [Dij76], los *niveles de abstracción* [SP91], la *programación por ejemplos* [ASFS18], etc., todas ellas se han enfocado en estrategias que apuntan a obtener programas correctos por construcción, tratando de obedecer el paradigma de desarrollo sistemático simultáneo con la prueba de correctitud que ilustra la siguiente figura:



En las secciones anteriores hemos profundizado en conceptos que hacen a la esencia de este paradigma, la verificación se mostró como una actividad *a posteriori* por fines exclusivamente didácticos. Para reforzar este concepto, en esta sección presentamos un ejemplo muy sencillo de desarrollo sistemático de programa guiado por el método axiomático de verificación estudiado.

Nos concentraremos en la construcción de un *while*, a partir de un invariante p y un variante t. Suponiendo que se quiere construir un programa *P :: T ; while B do S od*, tal que satisfaga la fórmula de correctitud $\langle r \rangle\ P\ \langle q \rangle$, de acuerdo a lo que hemos planteado previamente deben cumplirse los siguientes requerimientos:

1. A partir de la precondición r, T debe establecer el invariante p: $\langle r \rangle\ T\ \langle p \rangle$.
2. El predicado p debe ser efectivamente un invariante del *while*: $\langle p \wedge B \rangle\ S\ \langle p \rangle$.
3. El variante t debe decrecer con cada iteración: $\langle p \wedge B \wedge t = Z \rangle\ S\ \langle t < Z \rangle$.
4. Mientras se cumpla el invariante p, el variante t no debe ser negativo: $p \rightarrow t \geq 0$.
5. Al finalizar el *while* debe valer la postcondición q, es decir: $(p \wedge \neg B) \rightarrow q$.

La siguiente *proof outline*, en este caso de correctitud total, resume los requerimientos:

$$\langle r \rangle\ T\ ;\ \langle \text{inv: } p\ ,\ \text{var: } t \rangle\ \text{while } B\ \text{do } \langle p \wedge B \rangle\ S\ \langle p \rangle\ \text{od } \langle q \rangle$$

Siguiendo la guía anterior, construiremos un programa S_{sum} que sume en x los elementos de un arreglo a[0:N – 1] de enteros, siendo $N \geq 0$. Por convención, si $N = 0$ la suma será cero. S_{sum} tendrá la forma:

$$S_{sum} :: T\ ;\ \text{while } B\ \text{do } S\ \text{od}$$

y deberá satisfacer la especificación (r, q) siguiente:

Revista Abierta de Informática Aplicada/vol 6 N°1 (2022): 79-100
ISSN 2591-5320

$$r = N \geq 0$$
$$q = (x = \sum_{i=0,N-1} a[i])$$

Como primer paso, definimos un invariante p para el *while*. Una estrategia conocida, ya mencionada, es generalizar la postcondición q. En este caso se logra reemplazando la constante N por una variable k:

$$p = (0 \leq k \leq N \wedge x = \sum_{i=0,k-1} a[i])$$

La idea reflejada por el invariante es que en la variable x se irán sumando iteración tras iteración los elementos del arreglo, en el orden a[0], a[1], ... Para que valga el invariante por primera vez (requerimiento 1), construimos el siguiente fragmento inicial:

$$T :: k := 0 \; ; \; x := 0$$

Es decir, inicialmente k apunta al primer elemento del arreglo, a[0], y la variable x tiene el valor cero. Por ASI, SEC y CONS se cumple efectivamente:

$$\langle N \geq 0 \rangle \; k := 0 \; ; \; x := 0 \; \langle 0 \leq k \leq N \wedge x = \sum_{i=0,k-1} a[i] \rangle$$

En segundo término, definimos un variante t para el *while*. Con la idea mencionada en el paso anterior, hacemos:

$$t = N - k$$

al tiempo que incluimos k := k + 1 en el cuerpo del *while* y definimos B = (k ≠ N): el arreglo se irá recorriendo desde a[0], y mientras no se lo haya barrido todo (mientras no se alcance k = N) el *while* proseguirá. Así, considerando el invariante definido, t debería ir decreciendo con cada iteración (requerimiento 3), que habrá que corroborar cuando se tenga el cuerpo S completo, y se mantendrá no negativo (requerimiento 4). El requerimiento 5 también se cumple, la postcondición del *while* implica q:

$$(0 \leq k \leq N \wedge x = \sum_{i=0,k-1} a[i] \wedge \neg(k \neq N)) \rightarrow (x = \sum_{i=0,N-1} a[i])$$

La *proof outline* tiene hasta el momento la siguiente forma:

$\langle N \geq 0 \rangle$
$k := 0 \; ; \; x := 0 \; ;$
$\langle \text{inv: } 0 \leq \; k \leq N \wedge x = \sum_{i=0,k-1} a[i] \text{ , var: } N - k \rangle$
while $k \neq N$ do
$\langle 0 \leq \; k \leq N \wedge x = \sum_{i=0,k-1} a[i] \wedge k \neq N \rangle$
S' ;
$\langle (0 \leq k + 1 \leq N \wedge x = \sum_{i=0,k} a[i]) \rangle$
$k \; := k + 1$
$\langle 0 \leq k \leq N \wedge x = \sum_{i=0,k-1} a[i] \rangle$
od
$\langle x = \sum_{i=0,N-1} a[i] \rangle$

El cuerpo del *while* es por ahora S :: S´ ; k := k + 1. El predicado entre S' y k := k + 1 se obtuvo aplicando el axioma ASI. Para terminar el desarrollo del programa hay que calcular S´. Según la *proof outline* observamos que debe cumplirse la siguiente fórmula:

$$\langle 0 \leq k \leq N \wedge x = \sum_{i=0,k-1} a[i] \wedge k \neq N \rangle \; S' \; \langle 0 \leq k + 1 \leq N \wedge x = \sum_{i=0,k} a[i] \rangle$$

Que S´ sea una asignación que sume en la variable x el elemento corriente del arreglo, es decir x := x + a[k], se corresponde con el algoritmo que venimos concibiendo. Notar que de esta manera, por ASI y CONS, se verifica la fórmula anterior:

$$\langle 0 \leq k \leq N \wedge x = \sum_{i=0,k-1} a[i] \wedge k \neq N \rangle \; x := x + a[k] \; \langle 0 \leq k + 1 \leq N \wedge x = \sum_{i=0,k} a[i] \rangle$$

Considerando ahora el cuerpo completo del *while*, por SEC llegamos a:

$\langle 0 \leq k \leq N \wedge x = \sum_{i=0,k-1} a[i] \wedge k \neq N \rangle$
$x := x + a[k] \; ; \; k := k + 1$
$\langle 0 \leq k \leq N \wedge x = \sum_{i=0,k-1} a[i] \rangle$

que es el requerimiento que faltaba satisfacer (requerimiento 2): el predicado p es efectivamente un invariante del *while*. Además, corroboramos el cumplimiento del requerimiento 3, porque el agregado a S de la asignación x := x + a[k] no altera el valor del variante t.

La *proof outline* del programa S_{sum} queda en definitiva de la siguiente forma:

$\langle N \geq 0 \rangle$
$k := 0 \; ; \; x := 0 \; ;$
$\langle \text{inv: } 0 \leq k \leq N \wedge x = \sum_{i=0,k-1} a[i] \text{ , var: } N - k \rangle$
while $k \neq N$ do
$\langle 0 \leq k \leq N \wedge x = \sum_{i=0,k-1} a[i] \wedge k \neq N \rangle$
$x := x + a[k] \; ; \; k := k + 1$
$\langle 0 \leq k \leq N \wedge x = \sum_{i=0,k-1} a[i] \rangle$
od
$\langle x = \Sigma_{i=0,N-1} a[i] \rangle$

Para la construcción del programa hemos tenido que considerar, naturalmente, un solo invariante que sustente tanto la correctitud parcial como la terminación. Para la prueba de terminación se puede comprobar que alcanza con el invariante más débil $0 \leq k \leq N$.

6. Observaciones finales

Teniendo en cuenta las reglas de verificación descriptas, no es difícil plantear reglas para otras instrucciones de un lenguaje como el que presentamos, por ejemplo para una selección condicional sin *else*, una selección condicional múltiple del tipo *case*, una repetición en que se evalúa al final del tipo *repeat until*, etc. Más aún, los principios de la metodología estudiada aplican a extensiones del lenguaje de mayor envergadura (procedimientos, recursión, estructuras de datos más complejas) y a otros paradigmas de programación (no determinismo, concurrencia, programas

reactivos), sumando conceptos como *asunciones*, *variables auxiliares*, *invariantes globales*, etc. [RI10, RI13]. En todos los casos se recurre a las nociones fundamentales que hemos presentado en este artículo: invariante, variante, inducción, orden parcial bien fundado, composicionalidad, abstracción de estados por medio de predicados, etc.

La especificación de un programa es una conjunción de propiedades que el programa debe satisfacer. Por esta razón, una metodología de verificación de programas debe considerar las pruebas propiedad por propiedad. En este trabajo introductorio nos hemos limitado a estudiar la verificación de la correctitud parcial y la terminación, sugiriendo la práctica de atacarlas por separado. En paradigmas más complejos esta modalidad es insoslayable. Por ejemplo, en los programas concurrentes se deben probar propiedades adicionales como la ausencia de *deadlock*, la ausencia de interferencia (o exclusión mutua) y la ausencia de inanición (o *nonstarvation*). En realidad, el conjunto de propiedades que se pueden plantear cualquiera sea el paradigma es ilimitado. Hay una clasificación básica que considera dos familias de propiedades: propiedades *safety* (seguridad), que se prueban inductivamente y se refieren a situaciones indeseables que no pueden suceder (que no haya correctitud parcial, que haya *deadlock*, que haya interferencia, etc.), y propiedades *liveness* (vivacidad, progreso), que se prueban en base a órdenes parciales bien fundados y se refieren a situaciones deseables que deben suceder (terminación, y en general cualquier evento deseable a futuro, como por ejemplo que una impresora imprima o que un proceso ejecute una determinada instrucción). Así, en este artículo hemos cubierto el tratamiento de una propiedad representante de cada familia.

El método de verificación presentado es *sensato (sound),* lo que prueba tiene su correlato en la semántica asociada. En el caso de la correctitud parcial incluso lo es independientemente de la interpretación considerada (notar por ejemplo que la prueba del programa de división que desarrollamos aplica también para el dominio semántico de los números reales). El método además tiene la propiedad inversa, es *completo*, es capaz de probar todas las fórmulas de correctitud verdaderas. En otro trabajo tratamos estas consideraciones de la metateoría de correctitud de programas.

Nuestra última observación se relaciona con la enseñanza de la programación: ¡qué importante es tener conocimientos fundamentales en lógica y matemática!, lo que no debe sorprendernos por ser éstas las disciplinas originarias de las ciencias de la computación. En el desarrollo sistemático de software manejamos fórmulas compuestas por especificaciones y programas, en base a mecanismos formales construidos a partir de reglas y lenguajes rigurosamente definidos. Parafraseando a E. Dijkstra, a quien nos hemos referido intencionalmente varias veces por ser uno de los pensadores más lúcidos de las ciencias de la computación desde la segunda mitad del siglo XX, los programas son fórmulas de un sistema formal, y la tarea del programador consiste en derivarlas mediante la manipulación de los símbolos del sistema. Por si hiciera falta aclararlo, esta idea no se contrapone en absoluto al talento y la creatividad del programador: disciplina y arte, dos términos que popularizaron en este marco, respectivamente, E. Dijkstra y D. Knuth (referente fundamental de la algorítmica), son elementos necesarios y complementarios en la programación. Formalidad y rigurosidad por un lado, habilidad e ingenio por el otro (convergiendo idealmente en objetos de belleza tal como lo expresara D. Knuth), requerimos de ambas caras de la moneda para resolver los problemas computacionales, cada vez más diversos y complejos.

Referencias

Además de las referencias bibliográficas del artículo, sugerimos al lector curioso consultar el siguiente material adicional:

[Fra92, AO97] compendian métodos axiomáticos de verificación de programas para distintos paradigmas de programación. En [HVB88] se modeliza el proceso de desarrollo de software. [Jac95] reúne numerosos artículos referidos a principios y técnicas de análisis de requerimientos, especificación y diseño. [Vel86, Per81] describen varios métodos de construcción de programas. Los artículos [Dij68, Dij70, Dij72, Knu74a, Knu74b, Dij88] atestiguan parte del pensamiento de E. Dijkstra y D. Knuth durante los años del surgimiento y consolidación de la programación estructurada. Y [WGBF09] es una amplia muestra del estado del arte del soporte herramental a los métodos formales en la industria.

Detalle de todas las referencias:

» [ABKS13] Apel, S., Batory, D., Kästner, C. & Saake, G. *Feature-oriented software product lines*. Springer, 2013.

» [AFdR80] Apt, F., Francez, N. & de Roever, W. *A proof system for communicating sequential processes*. ACM Trans. Prog. Lang. Syst., 2, 3, 359-385, 1980.

» [AO97] Apt, K. & Olderog, E. *Verification of secuencial and concurrent programs, second edition*. Springer-Verlag, 1997.

» [Apt81] Apt, K. *Ten years of Hoare's logic, a survey, part 1*. ACM Trans. Prog. Lang. Syst., 3, 431-483, 1981.

» [Apt84] Apt, K. *Ten years of Hoare's logic, a survey, part 2: nondeterminism.* Theoretical Computer Science, 28, 83-109, 1984.

» [ASFS18] Alur, R., Singh, R., Fisman, D. & Solar-Lezama, A. *Search-based program synthesis.* Comm. ACM, 61, 12, 84-93, 2018.

» [BD77] Burstall, R. & Darlington, J. *A transformation system for developing recursive programs.* J. ACM, 24, 44-61, 1977.

» [BW82] Bauer, L. & Wössner, H. *Algorithmic language and program development.* Springer-Verlag, 1982.

» [CES09] Clarke, E., Emerson, A. & Sifakis, J. *Model checking: algorithmic verification and debugging*. Comm. ACM, 52, 11, 74-84, 2009.

» [CHT15] Cohen, E., Hillebrand, M. & Tobies, S. *Verifying C programs: a VCC tutorial*. European Microsoft Innovation Center & Microsoft Research Edmond, 2015.

» [CM88] Chandy, K. & Misra J. *Parallel program design: a foundation*. Addison-Wesley, 1988.

» [DBPU13] D´ippolito, N., Braberman, V., Piterman, N. & Uchitel, S. *Synthesising nonanomalous event-based controllers for liveness goals*. ACM Trans. Soft. Eng. and Meth., 22, 9, 1–36, 2013.

» [Dij68] Dijkstra, E. *Go to statement considered harmful.* Comm. ACM, 11, 3, 147-148, 1968.

» [Dij70] Dijkstra, E. *Notes on structured programming.* Eindhoven University of Technology, 1970.

» [Dij72] Dijkstra, E. *The humble programmer.* ACM Annual Conference, Boston, 1972.

» [Dij76] Dijkstra, E. *A discipline of programming*. Prentice-Hall, 1976.

» [Dij88] Dijkstra, E. *On the cruelty of really teaching computing science.* The University of Texas, Austin, 1988.

» [Flo67] Floyd, R. *Assigning meaning to programs*. Proc. American Mathematical Society, Symposium on Applied Mathematics, 19, 19-32, 1967.

» [Fra92] Francez, N. *Program verification*. Addison-Wesley, 1992.

» [FS06] Feinerer, I. & Salzer, G. *A comparition of tools for teaching formal software verification.* Conference Teaching Formal Methods: Practice and Experience (TFM), 2006.

» [Gri81] Gries, D. *The science of programming*. Springer-Verlag, 1981.

» [Ham88] Hamilton, A. *Logic for mathematicians, 2nd edition*. Cambridge University Press, 1988.

» [Hoa69] Hoare, C. *An axiomatic basis for computer programming*. Comm. ACM, 12, 576-580, 1969.

» [HR04] Huth, M. & Ryan, M. *Logic in computer science*. Cambridge University Press, 2004.

» [HS00] Holzmann, G. & Smith, M. *Automatic software feature verification.* Bell Labs Technical Journal, 2000.

» [HVB88] Haeberer, A., Veloso, P. & Baum, G. *Formalización del proceso de Desarrollo de software.* Editorial Kapelusz, 1988.

» [HW73] Hoare, C. & Wirth, N. *An axiomatic definition of the programming language PASCAL*. Acta Informatica, 2, 335-355, 1973.

» [Jac75] Jackson, M. *Principles of program design*. Academic Press, London, 1975.

» [Jac95] Jackson, M. *Software Requirements and Specifications: A Lexicon of Practice, Principles and Prejudices*. ACM Press, 1995.

» [Jan07] Janota, M. *Assertion-based loop invariant generation*. Proceedings of 1st International Workshop on Invariant Generation (WING 2007) collocated with the 14th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning, 2007.

» [Knu74a] Knuth, D. *Structured programming with go to statements.* Computing Surveys, 6, 4, 1974.

» [Knu74b] Knuth, D. *Computing programming as an art.* ACM Annual Conference, San Diego, 1974.

» [KPJ06] Kovács, L, Popov, N. & Jebelean, T. *Combining logical and algebraic techniques for program verification in Theorema.* Conference Leveraging Applications of Formal Methods, Verification and Validation, IsoLA, 2006.

» [LZ74] Liskov, B. & Zilles, S. *Programming with abstract data types*. SIGPLAN Notices, 4, 4, 50-59, 1974.

» [Men15] Mendelson, E. *Introduction to mathematical logic, sixth edition*. CRC Press, 2015.

» [MP89] Manna, Z. & Pnueli, A. *The anchored version of the temporal framework.* Proceeding linear time, branching time and partial order in logics and models for concurrency, school/workshop, 201-284. Springer-Verlag, 1989.

» [MP92] Manna, Z. & Pnueli, A. *The temporal logic of reactive and concurrent systems. Specifications*. Springer-Verlag, 1992.

» [MS11] Maoz, S. & Sa'ar, Y. *AspectItl: an aspect language for ltl specifications*. AOSD ´11, Proceedings of the tenth international conference on Aspect-oriented software development, 19–30, 2011.

» [OG76a] Owicki, S. & Gries, D. *An axiomatic proof technique for parallel programs*. Acta Informatica, 6, 319-340, 1976.

» [OG76b] Owicki, S. & Gries, D. *Verifying properties of parallel programs: an axiomatic approach*. Comm. ACM, 19, 279-285, 1976.

» [Per81] Pereira de Lucena, C. *Analise e sintese de programas: uma introducao*. Editora da Unicamp, 1981.

» [PRS17] Pons, C., Rosenfeld, R. & Smith, C. *Lógica para informática*. EDULP, 2017.

» [RI10] Rosenfeld, R. & Irazábal, J. *Teoría de la computación y verificación de programas*. McGraw-Hill y EDULP, 2010.

» [RI13] Rosenfeld, R. & Irazábal, J. *Computabilidad, complejidad computacional y verificación de programas*. EDULP, 2013.

» [SGS21] Sitnikovski, B., Goracinova-Ilieva, L. & Stojsevska, B. *Models for software verification: proving program correctness*. UTMS Journal of Economics 12, 1, 32–39, 2021.

» [SP91] Scholl, P. & Peyrin, J. *Esquemas algorítmicos fundamentales: secuencias e iteración.* Masson, Barcelona, 1991.

» [Vel86] Veloso, P. *Verificacao e construcao de programas*. Editora da Unicamp, 1986.

» [WGBF09] Woodcock, J., Gorm Larsen, P., Bicarregui, J. & Fitzgerald, J. *Formal methods: practice and experience*. ACM Computing Surveys, 41, 4, 1-40, 2009.

» [Wir73] Wirth, N. *Systematic programming: an introduction*. Prentice-Hall, 1973.

» [WPN08] Wenzel, M., Paulson, L. & Nipkow, T. *The Isabelle framework.* 21st International Conference, TPHOLs 2008, Montreal, Canadá, 2008.