# MÉTODOS FORMALES: TLA$^+$

**Juan Pablo Yamamoto Zazueta**
jpyamamoto@ciencias.unam.mx

Métodos Formales 2024-1
13 - diciembre - 2023

## 1 Introducción

El software actualmente controla gran parte de nuestras vidas, y los casos en los que un error durante el desarrollo ocasionó la pérdida de dinero, recursos e incluso vidas no son pocos. Desde la caída del servicio para llamadas a servicios de emergencia (911) en Estados Unidos en 2014 hasta los 4 pacientes muertos por un malfuncionamiento en la máquina Therac-25 para el tratamiento de cáncer.

La idea original para atacar estos problemas fue la ingeniería de software: forzarel uso de prácticas que obliguen a los equipos de desarrollo a escribir código seguro; enfatizando particularmente la realización de pruebas (unitarias, de integración, etc). Si bien el esfuerzo tuvo resultados positivos, tenemos ejemplos en los que vemos cómo esto no es suficiente.

Uno de esos ejemplos es el "Reto de Programación Extrema #14". En este se compartió un código concurrente y se invitó a la comunidad a aplicar las prácticas conocidas de la ingeniería de software para realizar pruebas que encontraran el error en el código. Únicamente tras haber descrito el autor original el error, fue que un desarrollador logró realizar pruebas que en pocos casos, caían en el error.

Todo esto deja en evidencia la necesidad de perfeccionar el cómo desarrollamos software y poner controles que verifiquen su correcto funcionamiento, más allá de las pruebas en el mismo código.

Además, con la velocidad a la que progresa el mundo hoy en día, es necesario que estos métodos formales de verificación no representen obstáculos en el proceso, sino que se adapten al flujo de trabajo de los desarrolladores y faciliten su adopción.

Una de las herramientas que permiten el rápido prototipado sin sacrificar por completo la formalidad, es TLA$^+$. Como veremos más adelante, esta herramienta es útil por la facilidad que proveé para desarrollar especificaciones de manera rápida, permitiendo ejecutar prontamente pruebas sobre esta, y facilitando el eventual formalizmo para tener mayores garantías.

## 2 TLA$^+$

TLA+ es un lenguaje para la especificación formal. A diferencia de la mayoría de las herramientas vistas durante el curso, el enfoque de TLA+ no está en verificar que el software sea correcto, sino dar un paso atrás y verificar que una especificación sea correcta; es decir: que los algoritmos o estructuras de datos mismas, no contengan errores que (tras ser implementados) ocasionen comportamientos indeseados.

TLA$^+$va a permitir al usuario definir qué transiciones de estados son válidas en el tiempo, dado un estado inicial. Además, podemos declarar mediante un lenguaje basado en la lógica de primer orden, cuáles son las propiedades que

deben cumplirse.

Otra de las ventajas de TLA$^+$es que, puesto que lidia con los algoritmos al nivel de su especificación, no estamos ligados a la implementación en particular de esta especificación. Es decir, la verificación sigue siendo igual de válida para poder implementarse eventualmente en cualquier lenguaje de programación.

### 2.1 TLA

La teoría que fundamenta a TLA+ es la conocida como *Temporal Logic of Actions*: una lógica desarrollada por Leslie Lamport que sirve para modelar el comportamiento de sistemas distribuidos y concurrentes, mediante acciones atómicas, estados y transiciones en el tiempo.

### 2.2 TLC

Por sí mismo, TLA$^+$como un lenguaje de especificación formal no proveé alguna garantía. Para ello se necesita de una manera para verificar que la especificación, bajo ciertas condiciones, se comporta de manera esperada.

Estas condiciones se conocen como *modelos*. Un modelo es una configuración que define el espacio de búsqueda sobre el cuál el verificador del modelo va a explorar, utilizando la especificación definida en TLA$^+$, hasta encontrar un comportamiento indebido, o explorar todo el espacio.

Esta herramienta que se encarga de la exploración se conoce como **TLC**. La característica que hace interesante a esta herramienta (y por la cuál su uso es popular en el área del cómputo distribuido y concurrente) es que genera todas las posibles ejecuciones posibles. Para un algoritmo secuencial, generalmente tenemos una sóla ejecución, dada por el orden de sus instrucciones. No obstante, para un algoritmo concurrente o distribuido, existen varias posibles permutaciones en el orden en que se ejecutan las instrucciones, dando lugar a espacios de búsqueda bastante amplios incluso para algoritmos sencillos, haciendo difícil su verificación con los métodos usuales de pruebas. **TLC** va a explorar todas estas posibles permutaciones de instrucciones y revisar exhaustivamente que todas se conformen a una especificación dada.

Es importante notar que en general, el dominio sobre el cuál definimos la especificación será infinito, o cuando menos muy grande para ser explorado en su totalidad. En ese sentido, es común que restrinjamos el modelo a un dominio suficientemente grande (según la opinión del desarrollador) para dar un buen nivel de confianza en su comportamiento. Esto ocasiona que TLA$^+$no sea un método completamente formal por sí mismo.

No obstante, lo anterior no significa que TLA$^+$no tiene utilidad como herramienta para el prototipado de especificaciones y como un método de "pruebas unitarias" (un poco más exhaustivas) sobre tales especificaciones.

### 2.3 TLAPS

Notando la carencia de TLA$^+$para dar completa certeza, se desarrolló el **TLA Proof System**. Este es un sistema que permite definir teoremas sobre las especificaciones en TLA$^+$, y demostrarlos formalmente.

Este sistema por sí mismo no es un asistente de pruebas, sino que internamente hace uso de otros solucionadores SMT como *Z3* e *Isabelle*. De cualquier forma, el sistema cuenta con la ventaja de ser relativamente sencillo de extender para utilizar tácticas más complejas o conectarse con otros solucionadores de pruebas. A pesar de carecer la expresividad de asistentes como *Coq*, el **TLAPS** es suficientemente bueno para la mayoría de los casos de uso que se presentan en el día a día de la industria.

### 2.4 PlusCal

Como ya mencioné, TLA$^+$en sí mismo es un lenguaje muy sencillo basado en la lógica de primer orden. Esto puede derivar en especificaciones muy largas y complejas para algoritmos relativamente sencillos. Debido a esto se desarrolló

**PlusCal**, un lenguaje con una sintaxis similar a la de lenguajes imperativos, que compila a TLA$^+$.

Además, tiene una buena interacción con TLA$^+$, pues es posible traducir el código de **PlusCal** dentro del *IDE* oficial de TLA$^+$, conocido como **TLA Toolbox**.

En adelante, me referiré a todos los conceptos introducidos anteriormente indistintamente como TLA$^+$, sin importar el subsistema en particular.

## 3 Máximo

A continuación muestro el desarrollo de la especificación y verificación de un algoritmo que obtiene el elemento máximo en un arreglo no vacío de números naturales.

### 3.1 Especificación

Primero damos la especificación del algoritmo para ello. Para tal fin utilizaré **PlusCal** por simplicidad. Adjunto a este artículo se puede encontrar el archivo `.tla` con el código original. Por ahora, presento la versión como PDF que se exportó directamente de la **Toolbox**.

MODULE *FindMax*

EXTENDS *Sequences*, *Naturals*, *Integers*, *TLAPS*

Especificación

```
 --fair algorithm Highest
variables
    array ∈ Seq(Nat) ;
    result = −1 ;
    index = 1 ;

define
    Max(a, b) ≜ IF a ≥ b THEN a ELSE b
end define ;

begin
    Iterate:
        while (index ≤ Len(array)) do
            result := Max(result, array[index]) ;
            index := index + 1 ;
        end while ;
end algorithm ;
```

Primero estamos importando algunas utilidades como lo son: secuencias (para modelar un arreglo), naturales y enteros; y **TLAPS** para hacer demostraciones formales (lo utilizaremos más adelante).

Primero definimos el algoritmo *Highest* dando que un `array` es una secuencia de naturales; el valor inicial para `result` es `-1` y el índice en el cuál vamos a comenzar es `index = 1`, pues en TLA$^+$los índices comienzan desde 1.

Luego, definimos el operador *Max*, con la definición canónica. Finalmente, damos el algoritmo que revisa elemento por elemento del arreglo, guardando el máximo en cada paso en `result`. Esto lo hacemos mediante un ciclo `while` que

etiquetamos con `Iterate`.

Otro detalle a notar de esta definición, es que le estamos indicando a TLA$^+$que este algoritmo es *fair*. Este tipo de algoritmos son aquellos en los que podemos garantizar que eventualmente habrá progreso. Es decir, no nos podemos quedar indefinidamente en un mismo paso. Para los algoritmos secuenciales, generalmente podemos garantizar que es así, no necesariamente para los distribuidos.

Deseamos definir ahora las propiedades que nos gustaría verificar para hablar de la correctez del algoritmo. Proponemos las siguientes:

1. *Tipado:*
   - El arreglo contiene números naturales.
   - El índice es un número natural.
   - El índice no se sale de los límites del arreglo.
   - El máximo en cualquier momento del algoritmo es un número natural o -1.
2. *Invariante de inducción:* Para todos los elementos antes del índice en que estamos iterando, los elementos en tales posiciones dentro del arreglo son menores o igual al máximo hasta el momento.
3. *Terminación:* Tras terminal, el índice es uno más que el tamaño del arreglo.
4. *Correctez:* Tras terminar, se mantiene la invariante de inducción para todas las posiciones en el arreglo.

Estas propiedades las codificamos de la siguiente manera en el bloque `define`:

MODULE $FindMax$

$$
\begin{array}{l}
TypeOK \triangleq \\
\quad \land \quad array \in Seq(Nat) \\
\quad \land \quad index \in 1 \,..\, (Len(array) + 1) \\
\quad \land \quad index \in Nat \\
\quad \land \quad result \in Nat \cup \{-1\} \\
\\
InductiveInvariant \triangleq \\
\quad \forall\, i \in 1 \,..\, (index - 1) : array[i] \leq result \\
\\
DoneIndexPos \triangleq pc = \text{“Done”} \Rightarrow index = Len(array) + 1 \\
\\
Correctness \triangleq \\
\quad pc = \text{“Done”} \Rightarrow \forall\, i \in \text{DOMAIN } array : array[i] \leq result
\end{array}
$$

### 3.2 Modelo

A continuación, deseamos verificar que las condiciones anteriores se cumplen. Puede ser mucho trabajo en vano directamente intentar demostrarlas formalmente antes de tener una idea de que efectivamente se satisfacen. Recordemos que muchos de los problemas más comúnes en este tipo de situaciones son debido a índices desplazados una posición, un acceso que se sale del límite, y otros errores que son fáciles de detectar con pruebas sencillas.

Si le pedimos al **TLC** que verifique las condiciones anteriores, vamos a obtener un error debido a que es imposible generar todos los casos de pruebas, pues son infinitos. Tenemos una infinidad de longitudes que el arreglo puede tomar, con una infinidad de instancias para cada natural en el arreglo.

Por lo tanto, al querer hacer una verificación exhaustica sobre todos los casos que se pueden presentar en la ejecución del algoritmo, encontramos que no es viable. Para solventar este error, y porque los errores en este tipo de algoritmos suelen encontrarse con ejemplares relativamente pequeños, vamos a restringir nuestro espacio de búsqueda:

MODULE $FindMax$

Verificación de *Modelos*

CONSTANTS $MaxLength$, $MaxNat$
ASSUME $MaxLength \in Nat$
ASSUME $MaxNat \in Nat$

$MCConstraint \triangleq Len(array) \leq MaxLength$
$MCNat \triangleq 0 \,..\, MaxNat$
$MCSeq(S) \triangleq \text{UNION } \{[1 \,..\, n \to S] : n \in Nat\}$

Lo que estamos haciendo es definir ciertas constantes que debemos definir al momento de ejecutar el modelo para indicar cuál es la longitud máxima para los arreglos a verificar exhaustivamente, y el natural máximo que contiene.

También ponemos una verificación de que los valores dados son naturales válidos, para asegurar que no ejecutamos pruebas incorrectas. Finalmente, definimos las restricciones para el espacio de búsqueda:

1. *MCConstaint:* La longitud de los arreglos que revisamos es menor o igual a la longitud máxima permitida.
2. *MCNat:* El dominio para generar naturales es desde 0 a *MaxNat*.
3. *MCSeq:* Los arreglos generados son todos aquellos de longitudes al menos 1, generados por los elementos dados en $S$. Eventualmente instanciaremos $S$ con *MCNat*.

Al ejecutar **TLC** con las condiciones anteriores, llegamos a que no se encontraron errores en la especificación. Esto nos da un grado de seguridad de que la especificación es correcta, y podemos pensar ahora en demostrar formalmente que es así.

### 3.3 Verificación Formal

Aún si lo anterior es un buen indicador de que el algoritmo está bien especificado, aún no tenemos una completa certeza de que es correcto. Para obtener esa garantía, vamos a utilizar el **TLAPS**.

Notemos que por la misma naturaleza del lenguaje y de la definición de modelos, resulta bastante directo enunciar los teoremas de correctez:

MODULE $FindMax$

Teoremas

THEOREM $TypeInvariantHolds \triangleq Spec \Rightarrow \Box TypeOK$
OBVIOUS

THEOREM $InductiveInvariantHolds \triangleq Spec \Rightarrow \Box InductiveInvariant$
OBVIOUS

THEOREM $DoneIndexPosTheorem \triangleq Spec \Rightarrow \Box DoneIndexPos$
OBVIOUS

THEOREM $IsCorrect \triangleq Spec \Rightarrow \Box Correctness$
OBVIOUS

Todos los teoremas son de la forma: Dada la especificación definida, verifica que se cumple la propiedad de correctez. En este contexto, el símbolo $\Box$ (escrito `[]` en el código) se debe leer como "siempre se cumple ...". En la lógica TLA, se interpretaría como un cuantificador universal sobre el tiempo.

Ahora procedemos a demostrar los teoremas. Una de las ventajas de este sistema es que en gran medida las demostraciones son automáticas, utilizando tácticas en otros solucionadores externos. Debido a eso, para la mayoría de los casos bastará dar una vista general sobre cómo realizar la demostración.

Para estos 4 teoremas, bastará hacer inducción. Los casos que se verifican son:

1. *Init:* Los teoremas se satisfacen en el estado inicial.
2. *UNCHANGED vars:* Si en un estado se satisface el teorema, en todos los posibles siguientes estados en que no se modifican las variables, se satisface el teorema.
3. *TypeOK* $\wedge$ *Terminating:* Si en un estado se satisface el teorema, en todos los posibles siguientes estados que llegan a un estado terminal y el tipado es correcto, se satisface el teorema.
4. *TypeOK* $\wedge$ *Iterate:* Si en un estado se satisface el teorema, en todos los posibles siguientes estados en que iteramos y el tipado es correcto, se satisface el teorema.

Bastará con indicar a **TLAPS** que debe revisar tales casos, e indicar que use las definiciones de las hipótesis, para mostrar los casos inductivos:

MODULE *FindMax*

Demostraciones de *Teoremas*

THEOREM $TypeInvariantHolds \triangleq Spec \Rightarrow \Box TypeOK$
PROOF
⟨1⟩a. $Init \Rightarrow TypeOK$
BY DEFS $Init$, $TypeOK$
⟨1⟩b. $TypeOK \wedge$ UNCHANGED $vars \Rightarrow TypeOK'$
BY DEFS $TypeOK$, $vars$
⟨1⟩c. $TypeOK \wedge Next \Rightarrow TypeOK'$
⟨2⟩a. $TypeOK \wedge Iterate \Rightarrow TypeOK'$
BY DEFS $TypeOK$, $Iterate$, $Max$
⟨2⟩b. $TypeOK \wedge Terminating \Rightarrow TypeOK'$
BY DEFS $TypeOK$, $Terminating$, $vars$
⟨2⟩ QED BY ⟨2⟩a, ⟨2⟩b DEF $Next$
⟨1⟩ QED BY $PTL$, ⟨1⟩a, ⟨1⟩b, ⟨1⟩c DEF $Spec$

THEOREM $InductiveInvariantHolds \triangleq Spec \Rightarrow \Box InductiveInvariant$
PROOF
⟨1⟩a. $Init \Rightarrow InductiveInvariant$
BY DEFS $Init$, $InductiveInvariant$
⟨1⟩b. $InductiveInvariant \wedge$ UNCHANGED $vars \Rightarrow InductiveInvariant'$
BY DEFS $InductiveInvariant$, $vars$
⟨1⟩c. $InductiveInvariant \wedge TypeOK \wedge TypeOK' \wedge Next \Rightarrow InductiveInvariant'$
⟨2⟩a. $InductiveInvariant \wedge Terminating \Rightarrow InductiveInvariant'$
BY DEFS $InductiveInvariant$, $Terminating$, $vars$
⟨2⟩b. $InductiveInvariant \wedge TypeOK \wedge Iterate \Rightarrow InductiveInvariant'$
BY DEFS $InductiveInvariant$, $TypeOK$, $Iterate$, $Max$
⟨2⟩ QED BY ⟨2⟩a, ⟨2⟩b DEF $Next$
⟨1⟩ QED BY $PTL$, ⟨1⟩a, ⟨1⟩b, ⟨1⟩c, $TypeInvariantHolds$ DEF $Spec$

```
THEOREM DoneIndexPosTheorem ≜ Spec ⇒ □DoneIndexPos
PROOF
  ⟨1⟩a. Init ⇒ DoneIndexPos
    BY DEF Init, DoneIndexPos
  ⟨1⟩b. DoneIndexPos ∧ UNCHANGED vars ⇒ DoneIndexPos'
    BY DEFS DoneIndexPos, vars
  ⟨1⟩c. DoneIndexPos ∧ TypeOK ∧ Next ⇒ DoneIndexPos'
    ⟨2⟩a. DoneIndexPos ∧ Terminating ⇒ DoneIndexPos'
      BY DEFS DoneIndexPos, Terminating, vars
    ⟨2⟩b. DoneIndexPos ∧ TypeOK ∧ Iterate ⇒ DoneIndexPos'
      BY DEFS DoneIndexPos, TypeOK, Iterate
    ⟨2⟩ QED BY ⟨2⟩a, ⟨2⟩b DEF Next
  ⟨1⟩ QED BY PTL, ⟨1⟩a, ⟨1⟩b, ⟨1⟩c, TypeInvariantHolds DEF Spec

THEOREM IsCorrect ≜ Spec ⇒ □Correctness
PROOF
  ⟨1⟩a. Init ⇒ Correctness
    BY DEF Init, Correctness
  ⟨1⟩b. Correctness ∧ UNCHANGED vars ⇒ Correctness'
    BY DEF Correctness, vars
  ⟨1⟩c. ∧ Correctness
        ∧ InductiveInvariant'
        ∧ DoneIndexPos'
        ∧ Next
        ⇒ Correctness'
    ⟨2⟩a. Correctness ∧ Terminating ⇒ Correctness'
      BY DEF Correctness, Terminating, vars
    ⟨2⟩b.
         ∧ Correctness
         ∧ InductiveInvariant'
         ∧ DoneIndexPos'
         ∧ Iterate
         ⇒ Correctness'
      BY DEFS Correctness, InductiveInvariant, DoneIndexPos, Iterate
    ⟨2⟩ QED BY ⟨2⟩a, ⟨2⟩b DEF Next
  ⟨1⟩ QED
    BY
      ⟨1⟩a, ⟨1⟩b, ⟨1⟩c,
      InductiveInvariantHolds, DoneIndexPosTheorem, PTL
     DEF Spec
```

Tras ejecutar **TLAPS**, nos notifica que los teoremas se pudieron demostrar, y con eso tenemos una verificación formal de la especificación.

## 4 Conclusiones

A partir de este pequeño ejemplo hemos podido identificar algunas de las posibilidades que brinda la herramienta TLA$^+$. Esta es una muy pequeña introducción en donde no llegamos a abordar los temas más relevantes en TLA$^+$, que son el cómputo distribuido y concurrente. No obstante, tenemos un panorama bastante general de cómo funciona la herramienta y cuál es su utilidad.

## References

[1] Adam Fabio. Killed by a machine: The therac-25. https://hackaday.com/2015/10/26/killed-by-a-machine-the-therac-25/, Oct 2015.

[2] Microsoft Research Inria. Tla+ proof system. https://tla.msr-inria.inria.fr/tlaps/content/Home.html.

[3] LambdaConf. Specifying distributed systems with tla+ workshop. https://youtu.be/H7yBYoY7ILc?si=7p6stiS_ccftfIMN, Dec 2021.

[4] Code Sync. A beginner's guide to tla+ exploring state machines and proving correctness. https://youtu.be/dtevi8lI4I8?si=Wa8QSutNmNHQ3lei, Apr 2020.

[5] Hillel Wayne. Learn tla+. https://learntla.com/.

[6] wiki.c2.com. Extreme programming challenge fourteen. https://wiki.c2.com/?ExtremeProgrammingChallengeFourteen, May 2009.