Especificación y Verificación Formal de Sistemas Distribuidos con TLA+

02. Principios de TLA+ y PlusCal

Axel Suárez Polo

October 20, 2022

BUAP

Contenidos

Modelando programas con TLA+

Modelando programas con TLA+

- La especificación de un sistema puede ir desde simple prosa, hasta una especificación matemática, como ocurre en la ciencia y la ingeniería.
  - En la realidad, los planetas tienen montañas, océanos, olas, clima, etc
  - Pero los podemos modelar como puntos de masa con posición y momento
- TLA+ permite hacer lo mismo pero con los programas

El modelo de comportamientos

El modelo de comportamientos

- TLA+ utiliza el *modelo de comportamientos*
- La ejecución de un programa es representada por un **comportamiento**.
- Un **comportamiento** es una secuencia ordenada de **estados**, ya sea finita o infinita.
- Un **estado** es una asignación de valores a variables.
- Un **programa** es modelado por un conjunto de **comportamientos**.

El modelo de comportamientos

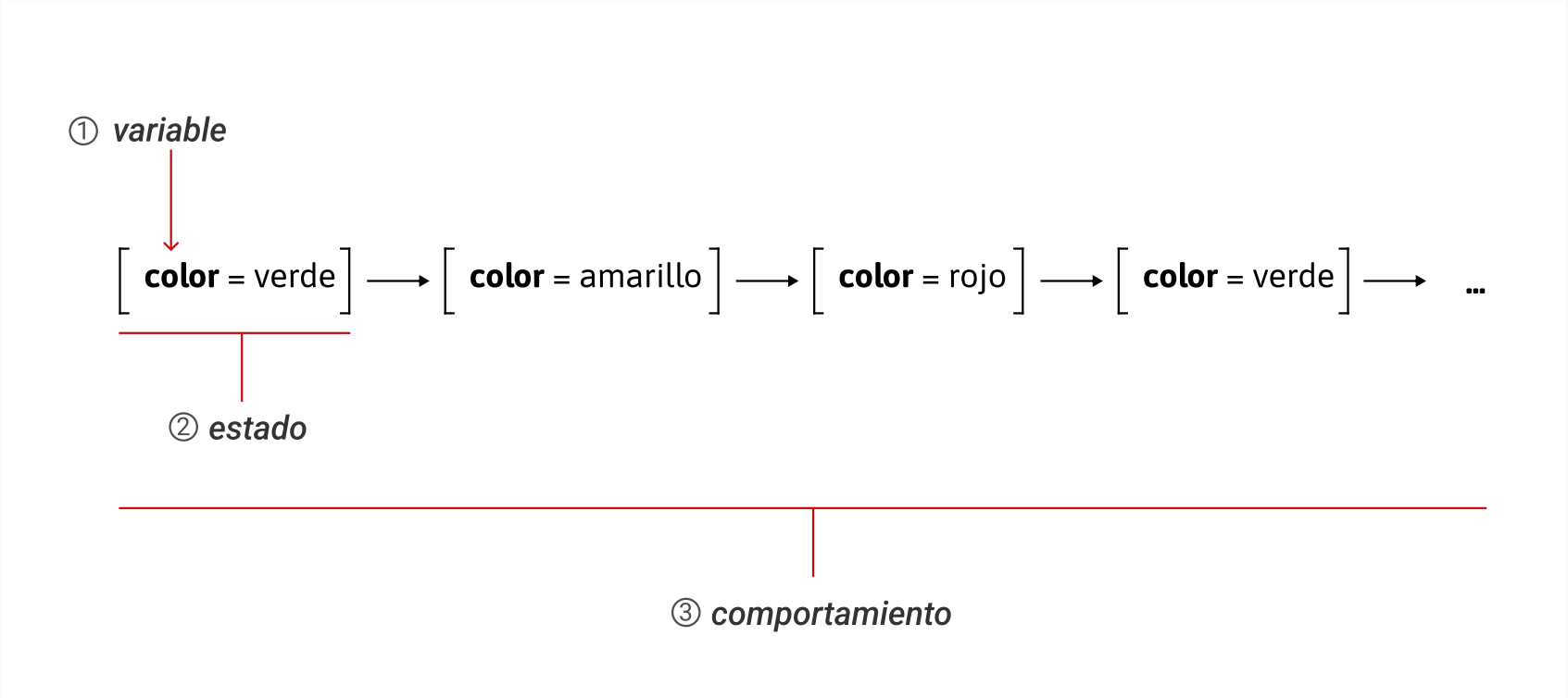

Figure 1: El comportamiento de un semáforo

El modelo de comportamientos

- *Un programa es modelado por todas sus posibles ejecuciones.*
- TLA+ nos permite especificar de forma precisa y concisa *todas* las ejecuciones posibles de un programa.

Un semáforo en TLA+

Semáforo en TLA+

```
1  VARIABLE color
2  SInit == color = "verde"
3  SNext == IF color = "verde"
4           THEN color' = "amarillo"
5           ELSE IF color = "amarillo"
6                THEN color' = "rojo"
7                ELSE color' = "verde"
8  Spec == SInit /\ [][SNext]_color
9
10 TypeInvariant == color \in {"verde", "amarillo", "rojo"}
11 ----
12 THEOREM Spec => []TypeInvariant
```

Listing 1: Semáforo en TLA+

Sintaxis

La sintaxis es la siguiente:

- `VARIABLE v1, v2, ...` declara las variables *v*1, *v*2, ...
- `Ident == e1` define el identificador *Ident* como la expresión *e*1. Es idéntico a `#define Ident e1` en C. [1]
- `e1 = e2` realiza la aserción de que *e*1 es *igual* a *e*2. Parecido a `e1 == e2` en C, pero también puede especificar el valor inicial de una variable.

[1] Permite la *Transparencia Referencial*.

Sintaxis

La sintaxis es la siguiente:

- `id' = e1` asigna al **siguiente estado** de la variable *id* el valor de la expresión *e*1.
- `e1 /\ e2` *and* lógico. Parecido a `e1 && e2` en C [2].
- `[][e1]_e2` aserción de que la propiedad *e*1 se cumple **siempre** o no ocurre cambio en las variables. [3]

[2] La diferencia es que `e1 /\ e2` no infiere control de flujo.

[3] Considerar que no haya cambio en las variables permite el *stuttering*.

Sintaxis

La sintaxis es la siguiente:

- `e1 \in e2` realiza la aserción de que *e*1 pertenece al conjunto *e*2. Si *e*1 es un identificador y es el estado inicial, entonces *e*1 toma todos los valores de *e*2.
- `{e1, e2, ...}` el conjunto conformado por *e*1, *e*2, etc.
- `e1 => e2` *e*1 implica lógicamente *e*2.

Transparencia Referencial

Una variable puede ser reemplazada por su valor sin alterar el programa.

No se cumple en lenguajes imperativos en general, pero sí en TLA+ y en lenguajes funcionales puros.

```c
#include <stdio.h>

int f() {
  printf("Hola\n");
  return 2;
}

int main() {
  int x = f();
  int valor = x + x;
  // int valor = f() + f();
  printf("%d\n", valor);
}
```

Listing 2: Transparencia Referencial no aplica en C

Stuttering

Permitir pasos que no cambian las variables (*stuttering steps*) permite una mejor **composición** (reutilización) de las especificaciones.

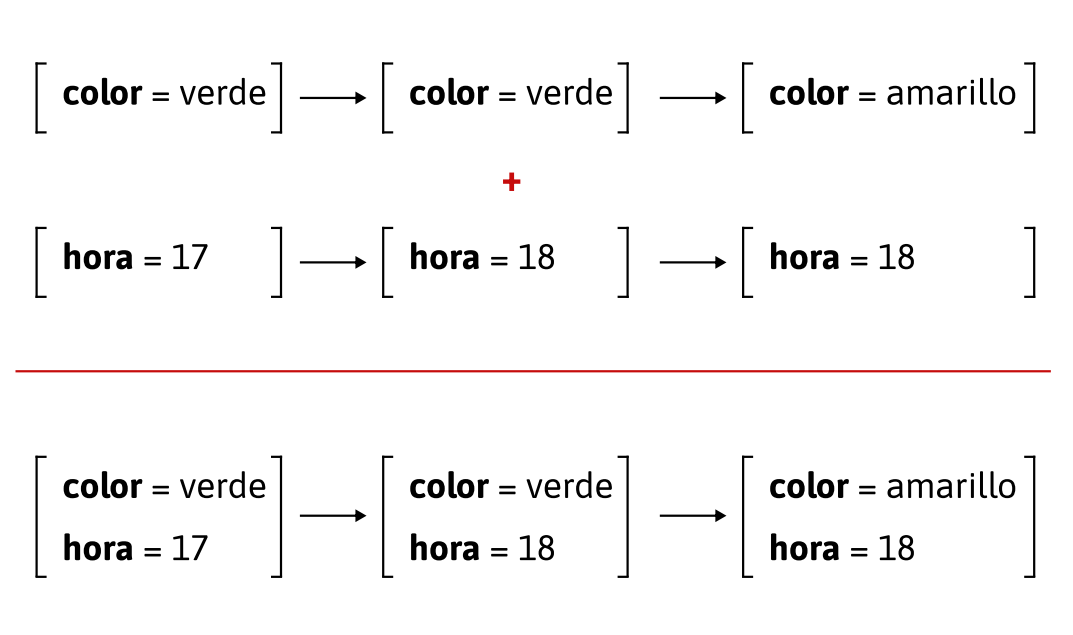

Figure 2: Composición de especificaciones con *stuttering*.

Un semáforo en PlusCal

Semáforo en PlusCal

```
1  variable color = "verde";
2
3  while TRUE do
4    if color = "verde" then
5      color := "amarillo";
6    elsif color = "amarillo" then
7      color := "rojo";
8    else
9      color := "verde";
10   end if;
11 end while;
```

Listing 3: Semáforo en PlusCal