Análisis de la herramienta TLA+ Proof System

Pablo Celayes, Giovanni Rescia, Ariel Wolfmann

Facultad de Matemática, Astronomía y Física
Universidad Nacional de Córdoba

Resumen Analizamos el lenguaje de especificación de alto nivel TLA+ y sus herramientas. Nos enfocamos particularmente en *TLA+ Proof System*, un entorno de verificación automática de pruebas formales. Estudiamos su contexto de creación, sus ventajas y desventajas y aplicaciones en la industria. Comparamos el lenguaje y sus herramientas con otros similares e ilustramos su forma de uso con un ejemplo algorítmico sencillo.

1. Contexto de creación de la herramienta

En 1977, Amir Pnueli introdujo el uso de lógica temporal para describir el comportamiento de un sistema, lo cual en principio podía hacerse en una sola fórmula pero no resultaba así en la práctica. La lógica temporal de Pnueli era ideal para describir ciertas propiedades, pero poco adecuada para muchas otras, por lo que normalmente se la combinaba con maneras más tradicionales de describir sistemas. [1] A fines de los 80's Leslie Lamport inventó el lenguaje TLA (Temporal Logic of Actions), una variante simple de la lógica temporal de Pnueli que facilita la descripción de un sistema completo en una sola fórmula usando mayormente Matemática ordinaria (no temporal) y reservando la lógica temporal sólo para aquellas propiedades en las que realmente es buena. [2]

En 2001, Lamport comenzó a trabajar en el centro *Microsoft Research* de Mountain View, Estados Unidos. De esta etapa surge el proyecto TLA+ Proof System (TLAPS) que actualmente se desarrolla como parte del proyecto *Tools for Proofs* en el centro conjunto de investigación *Microsoft Research - INRIA* de Palaiseau, Francia. El proyecto continúa en desarrollo activo y cuenta con una activa lista de mail y un sistema público de seguimiento de *bugs*.

2. Objetivo de la herramienta

TLAPS es una herramienta que verifica mecánicamente la correctitud de pruebas escritas en TLA+. Éste es un lenguaje de especificación de propósito general, orientado a sistemas concurrentes y distribuidos.[3] En general, una prueba de TLA+ es una colección de sentencias estructuradas jerárquicamente, donde cada sentencia tiene una afirmación, injustificada o justificada por una colección

de hechos citados. El propósito de TLAPS es verificar las pruebas de teoremas propuestas por el usuario, es decir, que la jerarquía de las sentencias de hecho establecen la veracidad del teorema si las afirmaciones fueran ciertas, y luego verificar que la afirmación de cada sentencia justificada es implicada por los hechos citados.[4] Si un teorema de TLA+ tiene una prueba con todas sus afirmaciones justificadas, entonces, como resultado de comprobar la prueba, TLAPS verifica que el teorema es cierto.

3. Descripción de la herramienta del lado del usuario

TLA+ es una herramienta de alto nivel para la descripción de los sistemas, especialmente sistemas concurrentes asíncronos y distribuidos. Fue diseñado para ser simple, muy expresivo, y permitir una formalización directa del razonamiento de aserciones tradicional. Se fundamenta en la idea de que la mejor manera de describir formalmente es con matemática simple, y que un lenguaje de especificación debe contener lo menos posible más allá de lo que se necesita para escribir matemática simple con precisión. Para cumplir con el objetivo de formalizar el razonamiento de aserciones, TLA+ está basado en TLA (*Temporal Logic of Actions*), una variante simple de la lógica temporal lineal.

TLA+ utiliza a **PlusCal** como lenguaje de especificación. Para el modelado se basa en el **Modelo Standard**. **TLC** es quien se encarga de verificar la corrección de los modelos. Finalmente, **TLA+ Proof System** es un asistente de pruebas, para verificar las especificaciones dadas en TLA+. En esta herramienta específica es donde nos enfocaremos el resto del documento. Cabe aclarar que TLA+ posee una interfaz gráfica donde el usuario puede utilizar todas estas herramientas de manera sencilla e integrada, llamada **Toolbox**.

- TLA+ utiliza **PlusCal** como lenguaje en el cual se debe realizar la especificación. PlusCal es un lenguaje algorítmico que, a primera vista, parece un lenguaje de programación imperativo pequeño. Sin embargo, una expresión PlusCal puede ser cualquier expresión de TLA+, lo que significa cualquier cosa que pueda ser expresada con matemática. Esto hace a PlusCal mucho más expresivo que cualquier lenguaje de programación. Un algoritmo PlusCal se traduce (compila) en una especificación TLA+, la cual puede ser chequeada con las herramientas de TLA+. Fue diseñado para reemplazar al pseudocódigo, conservando su sencillez y proporcionando al mismo tiempo un lenguaje formalmente definido y verificable, en el cual las personas sin tantos conocimientos profundos en matemática pueden definir su especificacion de forma sencilla y facil de entender, para luego traducirla a TLA+ usando la herramienta.
- En un **Modelo Standard**, un sistema abstracto se describe como un conjunto de comportamientos, cada uno representando una posible ejecución del sistema, donde un comportamiento es una secuencia de estados y un estado es una asignación de valores a las variables. En este modelo, un evento, también llamado un paso, es la transición de un estado a otro en un comportamiento.

- El chequeador de modelos **TLC** construye un modelo de estado finito de las especificaciones TLA+ para el control de las propiedades de invariancia. Genera un conjunto de estados iniciales que satisfacen la especificación y a continuación realiza una búsqueda en amplitud (Breadth-First Search) sobre todas las transiciones de estado definidas. La ejecución se detiene cuando todas las transiciones de estado conducen a estados que ya han sido descubiertos. Si TLC descubre un estado que viola un invariante del sistema, se detiene y ofrece una traza infractora. En caso de que haya alcanzado un estado que no tenga posibles acciones habilitadas, reporta un mensaje de error explicitando el posible *deadlock* (opcional). TLC ofrece un método de declarar simetrías del modelo para evitar el fenómeno de *explosión combinatoria*. También paraleliza el paso de la exploración del estado, y se puede ejecutar en modo distribuido para repartir la carga de trabajo a través de un gran número de computadoras. TLA+ es un ejemplo de lenguaje de mucha expresividad, puede ser fácilmente utilizado para especificar un programa que acepte una máquina de Turing arbitraria como entrada y puede determinar si se detendrá o no. Ningún chequeador de modelos puede manejar todas las especificaciones TLA+. TLC maneja un subconjunto de TLA+ que intenta incluir la mayoría de las especificaciones algorítmicas y propiedades de corrección, así como todas las especificaciones de diseño de protocolos y sistemas.
- **TLA+ Proof System**, el sistema de prueba de TLA+, es una plataforma que extiende a TLA+, para el desarrollo y verificación mecánica de demostraciones. El lenguaje de pruebas de TLA+ es declarativo y requiere cierto conocimiento previo de matemática elemental. Soporta desarrollo incremental y verifica la estructura jerárquica de la demostracion. La herramienta traduce una demostración en un conjunto de pruebas independientes, y llama a una colección de verificadores que se encargan de chequearlas, que incluyen demostradores de teoremas, asistentes de pruebas, chequeadores de satisfacibilidad y procedimientos de decisión. La versión actualmente disponible TLAPS maneja casi toda la parte no temporal de TLA+, como así también el razonamiento temporal necesario para probar propiedades de seguridad (*safety*) estándar, en particular invariantes y simulación de transiciones, pero no para propiedades de vitalidad (*liveness*). Desarrollar demostraciones es complejo y propenso a errores, por lo que previo a verificar la correctitud de una especificación, se recomienda chequear instancias finitas con TLC. Esto usualmente detecta varios errores sencilla y rápidamente. Una vez que TLC no encuentra errores, se puede intentar probar la correctitud de una demostración.

4. Aspectos técnicos de la herramienta

La arquitectura general de TLAPS (Fig. 1) se divide en: el administrador de pruebas de LTA (*LTAPM*) y tres *backends* que éste invoca: Isabelle, Zenon y SMT (Satisfiability Modulo Theories).

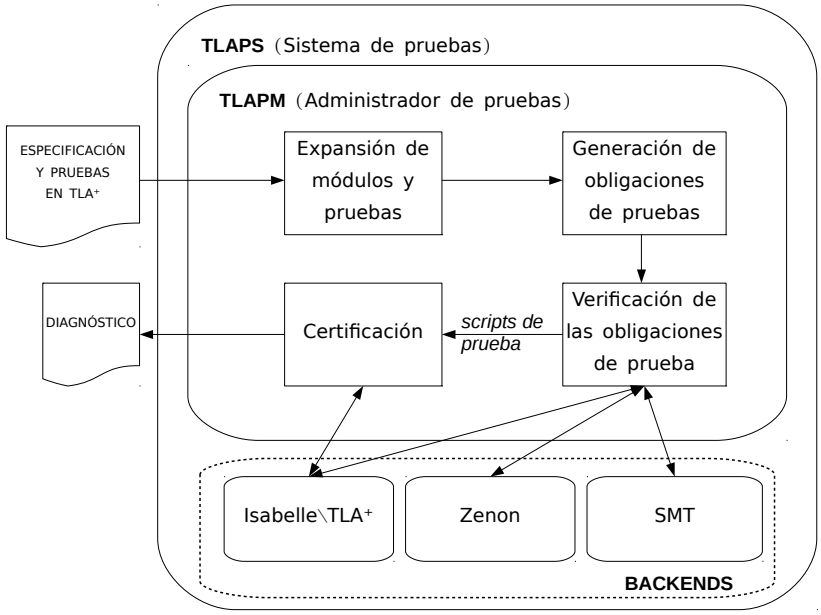

Figura 1. Arquitectura general del entorno de pruebas TLAPS

LTAPM: Una especificación en TLA+ consiste de un módulo raíz que puede (transitivamente), importar otros módulos por extensión e instanciación con sus respectivos parámetros. Cada módulo consiste en:

- Parámetros (variables, o estados, y constantes sin interpretar)
- Definiciones
- Teoremas, que pueden tener su prueba

Un parámetro de un módulo puede ser una variable (estado), o una constante; en las fórmulas de TLA+ no se hace referencia explícita a las variables, en cambio, se usan dos copias de una variable v: v y v', que hacen referencia a un valor antes, y después de la transición. TLAPS se ejecuta invocando a TLAPM en el módulo raíz e indicando qué pruebas chequear. El siguiente paso del LTAPM, es generar *obligaciones de prueba* para cada prueba terminal, es decir, las cosas que se tienen que demostrar, o corroborar, para probar la correctitud del teorema. Una vez generadas las obligaciones de prueba, organiza la ejecución de los backends para que las verifiquen. Cada obligación de prueba es independiente y puede ser probada por separado. Cuando los backends no pueden encontrar una prueba en un límite razonable de tiempo, el sistema informará cuál obligación es la que falló, junto con su contexto (falló, cancelada por el usuario, omitida) y objetivo de la prueba. El usuario debe entonces determinar si falló debido a que depende de hechos o definiciones ocultas, o si el objetivo es demasiado complejo y necesita

ser refinado con otro nivel de prueba. Esto indica que pueden haber obligaciones que son ciertas, y que por una cuestión de *timeout*, el backend las verifica como falsas. Si un backend encuentra una prueba de una obligación, es decir si la puede verificar, genera una traza de la prueba realizada, llamada *certificación de prueba*. LTAPM mediará la certificación de los scripts de prueba en un entorno lógico confiable, que en el diseño actual es Isabelle/TLA+. Finalmente, luego de certificar todas las obligaciones de prueba posibles generadas por las pruebas terminales, se procede a certificar el teorema en sí, en un proceso de dos pasos. En conclusión, primero, LTAPM genera la estructura de un lema (y sus pruebas en Isabelle/TLA+) que establece simplemente que la colección de obligaciones de pruebas terminales implican el teorema. Luego, LTAPM genera una prueba del teorema usando la estructura del lema y las obligaciones ya certificadas. Si Isabelle acepta la prueba, estamos seguros de que la versión traducida del teorema es verdadera en Isabelle/TLA+.[5]

4.1. Backends

Isabelle es un demostrador de teoremas genérico, diseñado para hacer razonamiento interactivo en una variedad de teorías formales. Provee un lenguaje para describir las lógicas-objeto y para probar sus teoremas. Es invocado cuando la prueba en Zenon falla.[5]

Zenon es el backend invocado por defecto por el TLAPM. Es un demostrador de teoremas automático para lógica clásica de primer orden (con igualdad), basado en el método tableau. Inicialmente diseñado para generar pruebas chequeables en el demostrador de teoremas Coq. Ahora extendido para generar scripts de pruebas chequeables en Isabelle. Uno de sus objetivos de diseño es resolver problemas simples de manera rápida.[6]

SMT es particularmente bueno en problemas difíciles que involucran aritmética, funciones sin interpretar y cuantificadores.[7]

5. Casos de aplicación de la herramienta

Mencionamos a continuación algunos ejemplos de aplicación de las herramientas analizadas. Fue difícil encontrar un ejemplo de la industria donde se use el *Proof System*, el enfoque más habitual resulta ser usar PlusCal y TLA+ para especificar el diseño del sistema y las propiedades deseadas (generalmente de *safety* o *liveness*), y luego usar el *model checker* TLC para intentar asegurar que no existan trazas de ejecución que violen las propiedades esperadas.

Amazon: *Amazon Web Services* utiliza TLA+ desde 2011. Haciendo *model checking* sobre especificaciones en TLA+ se han descubierto bugs en DynamoDB, S3, EBS, y en un *lock manager* interno distribuído. Muchos de los bugs encontrados eran críticos pero muy sutiles, requiriendo algunos trazas de hasta 35 pasos para ser detectados. También se ha usado model checking para verificar optimizaciones agresivas. Además, las especificaciones TLA+ han resultado valiosas como

documentación y ayudas de diseño. Inicialmente se intentó trabajar con Alloy, pero no resulto suficientemente expresivo. Se ha encontrado además que TLA+ es también muy práctico para modelado de datos y diseño de propiedades de *schemas*. [8]

XBOX 360: Microsoft usó TLA+ para encontrar y resolver un *bug* en el sistema de memoria de la consola *XBOX 360*. [10]

Byzantine Paxos: Leslie Lamport usó el TLAPS para derivar y chequear una prueba formal de la correctitud de la versión *bizantina* del *algoritmo de Paxos*, usado para garantizar tolerancia a fallas en sistemas distribuidos, y agregándosele en este caso tolerancia a un cierto número de procesos *maliciosos*. [11]

Farsite: Proyecto de investigación de Microsoft que buscaba desarrollar un sistema de archivos distribuido con la semántica de NTFS. Se desarrolló entre 2001 y 2006, y se usó activamente TLA+ para especificación y verificación de propiedades concurrentes. [12]

6. Comparación con otras herramientas

Para realizar la comparación con otras herramientas decidimos primero comparar TLA+ con otra herramienta ya conocida como Alloy, y luego abocarnos específicamente a TLAPS.

6.1. TLA+

TLA+ y Alloy poseen un concepto de modelado similar, pero TLA+ es mucho mas expresivo que Alloy. La expresividad limitada de Alloy aparece como consecuencia del enfoque particular que toma su herramienta analizadora a la hora de realizar el análisis. Alloy es limitado a las relaciones sobre los identificadores, no tiene estructuras de datos ni recursión, esto lo hace muy eficiente para verificar modelos pequeños, pero para modelos más realistas presenta problemas. Esta diferencia toma importancia en la práctica, ya que muchas especificaciones reales escritas en TLA+ son casi imposibles de escribir en Alloy.

6.2. TLAPS

Con respecto a TLAPS específicamente, las herramientas que encontramos como comparables son Isar, Focal y Coq.

- **Isar** es un lenguaje de prueba declarativo, que corre sobre Isabelle, pero tiene diferencias significativas, que llevan a un estilo de desarrollo de pruebas diferente. Por ejemplo, provee un acumulador para evitar referencias explícitas a los pasos de la prueba, lo cual es bueno para pruebas cortas, pero no tanto para demostraciones largas, que son típicas a la hora de verificar algoritmos. Además como Isabelle está diseñada para uso interactivo, los efectos de los comandos de Isar para demostraciones no son fácilmente predecibles, lo que estimula pruebas lineales en lugar de hacerlas jerárquicamente.

- El lenguaje **Focal** provee un conjunto de características funcionales y orientadas a objetos, que permiten expresar formalmente una especificación, avanzando incrementalmente, comprobando que la implementacion concuerda con la especificación y el diseño con los requerimientos. Es esencialmente un subconjunto de TLA+, que incluye el desarrollo de demostraciones jerárquicamente.
- **Coq** es un sistema asistente de demostraciones formales, que provee un lenguaje formal para escribir definiciones matemáticas, ejecutar algoritmos y teoremas en conjunto, con un ambiente para el desarrollo semi-interactivo de demostraciones verificadas automáticamente.

Básicamente, TLAPS realiza una integración entre las funcionalidades de Isar y Coq, ya que utiliza Isabelle y Zenon como backends.

7. Caso de estudio elegido

Ilustraremos la forma en que se trabaja con TLAPS a través de un ejemplo sencillo[9]: el desarrollo de una prueba de correctitud del clásico *Algoritmo de Euclides* para el cálculo del *máximo común divisor* entre pares de números naturales. A continuación, la arquitectura del código del algoritmo especificada en TLA+:

---------- Module *Algoritmo de Euclides* ----------

EXTENDS *Integers*

$p \,|\, q \triangleq \exists\, d \in 1..q : q = p * d$

$Divisors(q) \triangleq \{d \in 1..q : d \,|\, q\}$

$Maximum(S) \triangleq \text{CHOOSE}\, x \in S : \forall\, y \in S : x \geq y$

$GCD(p, q) \triangleq Maximum(Divisors(p) \cap Divisors(q))$

$Number \triangleq Nat - \{0\}$

CONSTANTS $M,\ N$

VARIABLES $x,\ y$

$Init \triangleq (x = M) \land (y = N)$

$$
\begin{aligned}
Next \triangleq \lor\ & \land\, x < y \\
& \quad \land\, y' = y - x \\
& \quad \land\, x' = x \\
& \lor\ \land\, y < x \\
& \quad \land\, x' = x - y \\
& \quad \land\, y' = y
\end{aligned}
$$

$Spec \triangleq Init \land \Box[Next]_{\langle x,y \rangle}$

$ResultCorrect \triangleq (x = y) \Rightarrow x = GCD(M, N)$

$InductiveInvariant \triangleq \land\, x \in Number$

$$\wedge\, y \in Number$$
$$\wedge\, GCD(x, y) = GCD(M, N)$$

ASSUME $NumberAssumption \triangleq M \in Number \wedge N \in Number$

THEOREM $InitProperty \triangleq Init \Rightarrow InductiveInvariant$

OBVIOUS

Ahora bien, si nosotros quisiéramos corroborar la correctitud de la prueba ejecutando TLAPS, éste nos indicaría un error ya que los *backends* fallarían al intentar demostrar la obligación de prueba. Ésto se debe a que TLAPS hace uso de los símbolos *Init* e *InductiveInvariant* como identificadores opacos, a menos que de manera explícita se les indique que expandan sus respectivas definiciones. Esto se logra con la directiva ***DEF***. Si cambiamos la última línea de código (**OBVIOUS**), por:

- **BY DEF** *Init, InductiveInvariant*
  , cuando las obligaciones de pruebas se envíen a los *backends*, éstas se expandirán según sus definiciones. Desafortunadamente, todavía no es suficiente para probar la correctitud del teorema, si hacemos la verificación de prueba, nos seguirá indicando que ninguno de los *backends* pudo completar la prueba. Tanto como con las definiciones, debemos especificar cuáles son los hechos que son *usables*. En este caso en particular, tenemos que hacer que el hecho *NumberAssumption* sea usable. Ésto se logra cambiando la última línea de código por:
- **BY** *NumberAssumption* **DEF** *Init, InductiveInvariant* Finalmente, SMT tiene éxito a la hora de poder verificar la correctitud de la prueba.

8. Conclusiones particulares

Buen balance entre expresivo y chequeable.

En la industria parece tener más aceptación el model checking que la verificación de pruebas.

Cooperación industria-academia muy beneficiosa (Microsoft / INRIA)

Basic TLA+ is simple enough to be quickly understood (actions, sets and functions): its standard mathematical notation helps.

Complex notions (e.g. fairness, non-determinism) are expressible and accessible.

Tools (TLC) are essential to our success

Referencias

1. Pnueli, Amir. **The temporal logic of programs**. Proceedings of the 18th IEEE Symposium on Foundation of Computer Science, 1977

2. Lamport, Leslie. **Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers**. Addison-Wesley. ISBN 0-321-14306-X, 2002
3. Brannon Batson, Leslie Lamport. **High-Level Specifications: Lessons from Industry**.
4. Kaustuv Chaudhuri ,Damien Doligez, Leslie Lamport, Stephan Merz. **A TLA + Proof System**.
5. Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, Stephan Merz. **Verifying Safety Properties With the TLA+ Proof System**. Fifth International Joint Conference on Automated Reasoning 2010
6. Richard Bonichon, David Delahaye, Damien Doligez. **Zenon: an Extensible Automated Theorem Prover Producing Checkable Proofs**. Paris, France
7. Martín Abadi, Frank McSherry, Derek G. Murray, and Thomas L. Rodeheffer. **Formal Techniques for Distributed Systems**. Joint IFIP WG 6.1 International Conference, FMOODS/FORTE 2013
8. Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, Michael Deardeuff. **How Amazon Web Services Uses Formal Methods**. Communications of the ACM, Vol. 58 No. 4, Pages 66-73
9. **TLA+ PS: A simple proof** `http://tla.msr-inria.inria.fr/tlaps/content/Documentation/Tutorial/A_simple_proof.html`. Microsoft Research - Inria Joint Centre
10. Lamport, Leslie **Thinking for Programmers** (at 21m46s) (Grabación de charla técnica). `http://channel9.msdn.com/Events/Build/2014/3-642#time=21m46s` Microsoft San Francisco
11. Lamport, Leslie **Byzantizing Paxos by Refinement** Distributed Computing: 25th International Symposium: DISC 2011,
12. William J. Bolosky, John R. Douceur, Jon Howell **The Farsite Project: A Retrospective** Microsoft Research Redmond