La Verificación Formal y sus Aplicaciones

Luna Rivera Carlos Alonso

Vázquez Dávila José Adolfo

Universidad Nacional Autónoma de México

1. Resumen

Con el paso del tiempo, el software ha evolucionando, haciendose cada vez más potente y más complejo, por ello la programación es cada vez más propensa a errores

También, el software se integra más y más a la vida cotidiana, con información personal privada, cofidencial gubernamental, económica, etc. haciendo uso de él, esto lleva a que los errores de software sean más peligrosos, entonces de qué forma aseguramos que no hayan errores en el software?

En este proyecto, haremos un breve estudio de los metodos y sus herramientas para asegurar la calidad del software y ponerlos en práctica

2. Preliminares

2.1. Definiendo la verificación formal

En dar una solucion a cualquier problema, es necesario asegurar la correctez de dicha solución, mientras más crítico sea el problema, más esencial se vuelve asegurar dicha correctez.

Considerando que muchos problemas críticos son llamados así por ser del mundo real, uno puede pensar que hacer simulaciones de la solución sería suficiente para probar su correctez; pero, si la solución es compleja, las simulaciones pueden volverse tardadas en ejecutar, o no considerar casos extremos o particulares que pueden causar que falle la solución propuesta.

Considerando esto, se desarrolla la verificación formal como una alternativa a la simulación y el estudio del comportamiento, que al consistir del uso de la demostración matemática, es más robusta que los otros métodos.

Al requerir demostración, la verificación formal requiere primero que nustro problema y solución se modelen de la forma adecuada, los modelos suelen ser fórmulas lógicas o autómatas de estado con los cuales es es posible especificar, diseñar, implementar y verificar los sistemas de información. Anteriormente los metodos formales eran considerados un ejercicio complejo e inviable en problemas reales, pero con el pasar de los años estos fueron tomando un mayor peso en la industria, pues estos iban mejorando hasta "parecerse.ª un lenguaje de programacion, además de que un sistema el cual tenia una çorreccion certificadaïmplicaba una mayor percepción de dinero.

2.2. Asistentes de Prueba

Para agilizar el proceso de la verificación formal, se han desarrollado varios asistentes de prueba, automatizan aspectos del proceso, pero su capacidad lógica es limitada, por lo que requieren intervención humana para dar los datos relevantes y guiar sus acciones, dicha intervención humana debe tomarse en cuenta a la hora de evaluar demostraciones presentadas por un asistente, ya que estan sujetas a error humano o uso malicioso.

Dos asistentes de prueba notables son:

- Mizar
  Desarrollado en los 70s, se basa en el lenguaje de las demostraciones matemáticas tradicionales y tomar ventaja del conocimiento previo en demostraciones
- HOL y sus distintas versiones
  Desarrollado desde los 80s, se basa en el cálculo-λ, derivó de LCF(Logic of Computable Functions) para implementar lógica de orden superior

Para la elaboracion de la solucion del problema en este proyecto se hara uso del asistente de pruebas Coq, este fue creado por INRIA en Francia, su primera version oficial salio en el año de 1989, los usarios se comunican con coq por medio de un lenguaje llamado "Gallina". La verificacion formal que usa coq esta basada en el calculo de construcciones inductivas, el cual solamente tiene 5 reglas de inferencia. Para facilitar el uso de este asistente, se cuentan con 2 interfaces graficas: CoqIDE y ProofGeneral, en nuestro caso estaremos haciendo uso de CoqIDE.

2.3. Aplicaciones de la verificación formal

Un gran ejemplo del uso de la verificación formal es la herramienta Infer, inicialmente desarrollado por Meta para uso en sus sitios (Facebook, Instagram, Whatsapp, etc.), usa la verificación formal para identificar posibles errores en sus códigos,y así evitar condiciones de carrera, conflictos de concurrencia, perdida de desempeño, etc.

Compañías de procesadores como Intel usan la verificación formal para asegurar y mejorar las arquitecturas de los mismos, en el caso de Intel, han usado el asistente de prueba HOL-light

Microsoft desarrolló SDV (static driver verifier), una herramienta para asisitir en el desarrollo de controladores, esto a raiz de que la complejidad del API de controladores de Windows resultaba en controladores defectuosos que vulneraban Windows XP

Amazon Web Services, que proporciona servicios como almacenamiento y proceso de datos en la nube, al manejar informacion sensible (privada, corporativa y gubernamental), ocupan el lenguage TLA+ para eliminar error humano en la seguridad de sus codigos

NASA utiliza una herramienta llamada Java Pathfinder para verificar la traducción de programas de Java a Promela, siendo Promela el lenguaje de otra herramienta de verificación, Spin, que se ha usado para asegurar sistemas operativos de diversos satelites y vehiculos de exploración

Otra de las ramas a las cuales se puede aplicar la verificación formal son las redes de comunicación, específicamente para protocolos de verificación, Manejo de la configuración, verificacion y seguridad de la red.

En el caso de los protocolos de verificación pueden ser validados en cuanto a su implementacion y diseño, pues estos pueden caer en 3 errores: deadlock (situación en que los protocolos esperan por una condición que jamás sucederá), livelocks (cuando un protocolo se ejecuta indefinidamente) o cuando un protocolo termina sin haber cumplido las metas esperadas. Para la detección y corrección de estos han sido de gran utilidad los asistentes de prueba como Isabelle o coq.

El manejo de la configuración de la red puede crear problemas de conectividad, seguridad y de desempeño. Estos problemas se generan debido a fallos en el acceso de control o fallos en el alcance de los protocolos. Para la solución de estos se hace uso de un análisis estático con la ayuda de la herramienta rcc, pero al igual que otros asistentes de prueba, rcc puede presentar fallos a la hora de hacer su trabajo, pero a pesar de esto ha sido de gran ayuda en esta rama.

Por otra parte, la seguridad tambien es de gran importancia en las redes de comunicacion, hay bastantes subproblemas en la veridicacion del firewall, para la solucion de estas invluyen el analisis estatico, diagramas de decision o SAT solvers.

3. Implementación

Es sabido que, además del sistema decimal para la representación de los numeros naturales, existen otras formas de manejarlos, como estudiantes de ciencias de la computación sabemos que practicamente todas las computadoras trabajan con el sistema binario, es por ello que desarrollamos este proyecto con el objetivo de definir una representación de los numeros naturales en un sistema binario con la ayuda del sistema de pruebas coq, además de demostrar su equivalencia con el sistema convencional.

Para lograr esto primero se estable una definicion de tipo inductiva "bin.[el] cual establece a Zero como tipo bin (representacion del numero 0) además de tener dos constructores: Doble y Suc, los cuales dado un numero binario b, representan el doble de b y el sucesor de b respectivamente.

Durante la implementación del sistema binario nos dimos cuenta que no es posible definir una forma fácil de representar a los numeros binarios en coq, por ejemplo, no podemos definir al numero dos como 10, pues coq no lo permite, lo cual resulta inconveniente a la hora de representarlos. Un problema al que nos enfrentamos a la hora de probar la equivalencia entre ambos sistemas fue que el numero 0 tiene más de una representacion en forma binaria, por ejemplo $00 = 0$, pero en nuestra definicion esto podría representarse como Doble Zero = Zero, lo cual es falso, esto resulta inconveniente a la hora de convertir un numero binario a natural y luego pasarlo a binario, es por esto que se tienen que normalizar.

Por otra parte, aunque a lo largo de la solución no utilizamos la táctica omega (obsoleta desde la version 8.12, ahora se utiliza Lia), esta sirve para probar de una manera "directa"proposiciones aritmeticas del conjunto de los numeros naturales o de los numeros enteros, esto puede ser útil para no estar definiendo nuestros propios lemas o teoremas auxiliares que podrían resultar obvios.

3.1. Funciones utilizadas en la solución del problema

.

Incrementa

Se suma 1 a un numero binario dado manteniendo la forma binaria.

bin2nat

Se define una funcion recursiva el cual se encarga de transformar un numero binario a su forma decimal, el Zero es 0, el Doble de n es 2*n y Suc n es el Sucesor de n.

Conmuta

Establece que dado un numero binario, si lo transformamos a natural, el resultado es equivalente a primero transformar el numero binario a natural y sumarle 1. Para la prueba de esta propiedad se ocupan lemas auxiliares: suma_0_derecha y suma_suc, en este caso puede resultar util utilizar la tactica omega(lia) para reducir la demostración.

nat2bin

Hace una conversion de manera recursiva de un numero natural a un numero binario, el 0 es Zero y el sucesor de n es incrementa n, se utiliza incrementa en lugar de Suc para considerar todos los casos.

inversa

Se tiene que demostrar que si convertimos un numero natural n a binario y luego a natural, el resultado es el mismo natural n. Se ocupa el teorema conmuta definido anteriormente junto al lema suma_suc.

inversa2: nat2bin (bin2nat n) = n

Esta propiedad es falsa, consideremos n= Doble Zero, entonces nat2bin(bin2nat n)=Zero el cual es distinto de Doble Zero, esto se debe a las multiples representaciones del 0 en binario, es por esto que se debe definir una normalizacion de los numeros binaios.

doble

Se encarga de calcular el doble de un numero binario, para el Zero simplemente se regresa Zero, para el Doble de n, es el Doble (doble n), mientras que para Suc n, es Suc(Suc(doble n)), el ultimo caso resulta de la multiplicacion 2(n+1)=2n+2, el cual representa el sucesor del sucesor del doble de n.

incrementa_incrementa_doble

El resultado de incrementar 2 veces el doble de un numero es igual al resultado de primero incrementar el numero y luego su doble. La demostración de esta propiedad en todos los casos es por definición, se hace reflexivity directamente.

n_mas_n : nat2bin (n + n) = doble (nat2bin n)

Se hace inducción sobre n haciendo uso de lemas auxiliares definidos anteriormente.

n_mas_n_mas_1

Se establece que nat2bin(n + n + 1) = incrementa (doble (nat2bin n)). Al igual que algunas de las propiedades anteriores se hace induccion sobre n y se utilizan reglas definidas anteriormente.

normaliza

Se encarga de normalizar un numero binario n de manera recursiva, el unico caso que se encarga de esto es Doble Zero el cual es reducido a simplemente Zero, el resto de la estructura del numero se mantiene igual.

normaliza_correcta

Ahora el resultado de transformar un numero binario a natural y luego a binario da el mismo numero binario (normalizado).

4. Conclusiones

A lo largo del desarrollo del proyecto hemos logrado una representación binaria de los numeros naturales en coq, además de demostrar su equivalencia con el sistema decimal. En el caso particular de coq fue necesaria definir una forma de normalizar un numero en su representación binaria. Consideramos que la solución propuesta es eficiente, pues la recursion es usada solo en los casos que es necesaria. La complejidad a lo mucho es O(n), por ejemplo las funciones normaliza y nat2bin, mientras que incrementa tiene una complejidad constante. Por otro lado, según la documentación de la biblioteca Omega, el uso de esta táctica puede llegar a ser lento, pero nosotros al definir nuestros propios lemas a utilizar, la prueba de ciertos teoremas tiene una mayor eficiencia.
[6] [12] [11] [3] [1] [8] [5] [10] [4] [7] [2] [9]

Referencias

[1] lowicz Adam Grabowski Adam Naumowicz Artur Korni. *Mizar Hands-on Tutorial*. 2016. URL: https://mizar.uwb.edu.pl/cicm_tutorial/mizar.pdf.

[2] Fan Zhang Chris Newcomb Tim Rath. *How Amazon Web Services Uses Formal Methods*. 2015. URL: https://cacm.acm.org/research/how-amazon-web-services-uses-formal-methods/.

[3] Coq. URL: https://coq.inria.fr/doc/v8.13/refman/addendum/omega.html.

[4] Microsoft Corporation. *Thorough Static Analysis of Device Drivers*. 2006. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/eurosys2006-1.pdf.

[5] fbinfer. *About Infer*. URL: https://fbinfer.com/docs/next/about-Infer/.

[6] Ian Grout. *Digital Systems Design with FPGAs and CPLDs*. 2008. URL: https://doi.org/10.1016/B978-0-7506-8397-5.X0001-3.

[7] John Harrison. *Formal verification of IA-64 division algorithms*. 200. URL: https://www.cl.cam.ac.uk/~jrh13/papers/hol00.pdf.

[8] John Harrison. *HOL Light Tutorial*. 2017. URL: https://www.cl.cam.ac.uk/~jrh13/hol-light/tutorial.pdf.

[9] Thomas Pressburger Klaus Havelund. *Model Checking Java Programs Using Java PathFinder*. 1999. URL: https://ntrs.nasa.gov/api/citations/20000068918/downloads/20000068918.pdf.

[10] ligurio. *A List of companies that use formal verification methods in software engineering*. 2024. URL: https://github.com/ligurio/practical-fm.

[11] Patrick Schnider. *An Introduction to Proof Assistants*. URL: https://people.inf.ethz.ch/fukudak/lect/mssemi/reports/09_rep_PatrickSchnider.pdf.

[12] Roberto Sebastiani. *Introduction to Formal Methods*. 2020. URL: http://disi.unitn.it/rseba/DIDATTICA/fm2020/.