Análisis de CBMC y su funcionamiento.

Mauricio D. Mazuecos Perez, Facundo Molina Heredia y Hernán J. Maina

Facultad de Matemática, Astronomía, Física y Computación
Universidad Nacional de Córdoba

Resumen En este documento presentamos un análisis de la herramienta CBMC, que usa model checking acotado (BMC, por sus siglas en inglés) para programas de C y C++. La idea del mismo es aportar datos relevantes de la herramienta y su desempeño en distintas aplicaciones. Luego de leer este documento, el lector tendría la información necesaria para saber si CBMC es el model checker que necesita.

1. Introducción

La herramienta CBMC (C Bounded Model Checker) fue nombrada por primera vez en el 2003 en un paper acerca de consistencia en comportamiento de programas en C y Verilog[1] y presentada formalmente en el 2004[2]. Originalmente se creó para corroborar la consistencia del modelo de nuevos dispositivos. Estos dispositivos estaban usualmente descritos en un lenguaje de programacíon como ANSI-C, y su implementación escrita en Verilog.

En este documento presentaremos un análisis de la herramienta, explicaremos su funcionamiento y mostraremos unos pequeños casos de uso de la misma. Compararemos la herramienta a otras existentes y daremos una breve reflexión de la utilidad y los campos de acción de la misma.

El documento se organizará de la siguiente manera: en la siguiente sección presentaremos algunos conceptos necesarios para entender el funcionamiento de las técnicas empleadas por CBMC; luego explicaremos la mecánica y el funcionamiento de CBMC; en la siguiente sección expondremos comparaciones con otros comprobadores de modelos y el uso de la herramienta, y finalizaremos con una breve conclusión.

2. Preliminares

Para poder introducir al lector a la mecánica de CBMC, primero introduciremos una serie de conceptos necesarios para la comprensión de la herramienta.

CBMC es una implementación de BCM. BMC [3](Bounded Model Checking) es una técnica de comprobación de modelos que trata al problema bajo consideración como una máquina de estados finitos, M, con un umbral de k estados para la ejecución. Tal máquina de estados está compuesta de n estados $s_1 ... s_n$ y m transiciones, donde cada transición φ_i es una terna (s,α,s') (i.e. si s es el estado actual y la condición α se cumple, entonces transiciona al estado s'). La

idea básica detrás de BCM es, dada una determinada propiedad P, buscar un contra ejemplo para P con k transiciones de estado como máximo. Si no lo encuentra, aumenta el umbral en uno hasta que bien encuentre un contra ejemplo o el problema se vuelva intratable.

Para representar el espacio de estados y posteriormente formar la fórmula que será pasada a un SAT Solver (solucionador de problemas de satisfacibilidad proposicional), CBMC transforma el programa a un CFG (Grafo de Control de Flujo, por sus siglas en inglés). Un CFG es una representación, usando notación de grafos, de todas las posibles trazas de un programa durante su ejecución.

Finalmente, CBMC genera una fórmula en CNF para pasarla a un SAT Solver y comprobar una propiedad. En lógica booleana, una fórmula está en CNF (Forma Normal Conjuntiva, por sus siglas en inglés) si es una conjunción de cláusulas. Los únicos conectivos que una fórmula puede tener en CNF son $\wedge$, $\vee$ y $\neg$.

En la siguiente sección se tratará la implementación y los detalles técnicos de la herramienta.

3. CBMC - Características y métodos

BMC transforma los programas de C y C++ en un CFG, con el cual representará el espacio de estados, incluyendo solo aquellos estados alcanzables en una ejecución. La idea detrás de esto es tomar caminos en el CFG hasta una aserción y construir la fórmula correspondiente a tal camino. Esta fórmula luego es pasada a un Sat Solver, que nos dirá si la misma es satisfacible o no, i.e. si esa ejecución es alcanzable o no.

Dado un sistema de transición $A = (S, s_0, T)$ (donde S es un conjunto de estados, s_0 un estado inicial y T una relación de transición), para evitar la explosión exponencial de caminos, CBMC simplemente concatena la relación T a cada paso. Es decir, busca una sucesión de k+1 estados $s_1 ... s_k$ tal que: s_0 sea el estado inicial; para cada dos estados adyascentes $s_i ... s_{i+1}$ exista una transición T, y P no se satisfaga en s_k. Luego, las asignaciones que satisfacen la ecuación (1) son trazas en A. Si queremos corroborar una propiedad $\mathbf{AG}p$, solo se tiene que encontrar un s_i que satisfaga $\neg p$. Encontrar una asignación que satisfaga la ecuación (2), es encontrar un contraejemplo para la propiedad $\mathbf{AG}p$.

$$S_0(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \tag{1}$$

$$S_0(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg p(s_i) \tag{2}$$

Para optimizar este proceso, las partes no alcanzables del programa no son generadas durante la fase de desenrrollamiento.

3.1. Implementación

Obviamente, la implementación en sí no es tan simple, y requiere abordar muchos problemas pequeños, como bucles, azúcares sintácticos, modelo aritmético, etc.

En una primera etapa el programa es simplificado removiendo todos los azucares sintácticos y estructuras en el código (i.e. se sustituyen $i++$ por $i = i + 1$, ciclos for por $while$, etc). Luego de ello, se procede a 'desenrrollar' todos los bucles del código, i.e. se sustituyen $while$(condición) por if(condición), hasta una profundidad k, en la cual se introduce un $assert(\neg$condición).

Pasada esta etapa, lo siguiente es resolver las multi asignaciones de variables ya que pueden ocasionar problemas semánticos relacionados. Para ello cuando una variable es asignada más de una vez, se agrega una variable nueva para cada nueva asignación. Al código resultante, tras completar las anteriores etapas, se lo denomina SSA-Program (Asignaciones Estáticas Simples, por sus siglas en ingles) y es el código intermedio y precursor que será utilizado en la construcción de las fórmulas CNF para la corroboración de propiedades, de la manera en que anteriormente se ha descripto.

3.2. Fases de verificación

A contianuación explicaremos el flujo de la verificación.

1. El programa en C se introduce en el motor de análisis de CBMC junto con la propiedad a satisfacer P y un cota k.

2. Se genera una fórmula CNF mediante la adición de variables intermedias, junto con un término que describe la negación de la propidad $(C \wedge \neg P)$.

3. La fórmula en CNF resultante se introduce en un SAT Solver. Si la ecuación se satisface, encontramos una violación de la propiedad, caso contario, si no se satisface, la propiedad se satisface.

4. En caso de haber encontrado un contraejemplo, se vuelve a traducir a "lenguaje de programa"de manera de facilitar la visualización del error al programador.

5. En el caso de que el SAT solver crahsee, dependiendo de que haya generado el crasheo, el programa respondera de diversas maneras. Lo mas probable es que simplemente, se clave.

La herramienta no retorna *falsos negativos*, i.e. si encuentra un contraejemplo que no satisface la propiedad en estudio, éste es certero; pero sí puede otorgar *falsos positivos*, ya que CBMC comprueba la satisfacción de la ecuación (2) para ejecuciones de trazas de a lo sumo k estados, lo cual nada asegura la insatisfactibilidad de la ecuación mas allá del k-ésimo estado.

3.3. Descripción de la herramienta del lado del usuario

CBMC puede ser encontrada tanto para ser utilizada mediante línea de comandos como por interfaz gráfica. Una posible opción para ésta última modalidad, es por medio de CProver suite, un plug-in para Visual Estudio que le brinda soporte [6].

La herramienta trabaja directamente sobre el código del programa, específicamente sobre lenguaje ANSI-C y derivados. El lenguaje de especificación de propiedades es asercional.

Cabe destacar que en caso de encontrar un error en el programa, se reporta un contraejemplo. Para la opción de linea de comandos, este se visualiza mediante una sucesión de variables y los correspondientes valores que hacen falsa la propiedad en estudio. Además la misma no permite simulación.

3.4. Funcionalidades

Con CBMC se pueden verificar tanto propiedas particulares de interés semántico, como accesos correctos a arreglos y ausencia de memoria sin liberar entre otras propiedades deseables que todo programa debe satisfacer. CMBC también comprueba propiedades de una lista autogenerada en base a un previo análisis estático. Estas propiedades autogeneradas no corresponden necesariamente a bugs, sino que, comprueban posibles fallos. CBMC cuenta con comandos para ver las propiedades autogeneradas que verifica. Además se puede pedir que luego de verificar ciertas propiedades, CBMC devuelva una traza de contraejemplo para las propiedades que fallaron en la comprobación. A veces no queremos comprobar todo un programa, sino solamente una función, o un módulo, en estos casos también podemos usar CBMC, tanto con propiedades autogeneradas como con propiedades específicas.

3.5. Nota sobre compiladores y librerías de ANSI-C

La mayoría de los programas en C usan funciones provistas por librerias. Las comprobaciones para dichos programas requieren de dos cosas: 1) Que los archivos .h de dichas funciones sean provistos; 2) Y que las definiciones apropiadas de las funciones sean provistas.

4. Comparaciones y uso

Vamos a comparar CBMC con LLBMC [5] (Low-Level Bounded Model Checker). LLBMC lleva los programas de C a una representación intermedia de LLVM (Low-Level Virual Machine) que luego es convertido a una fórmula lógica, que después de ser simplificada, se pasa por un SMT Solver. En un estudio realizado en el Institute for Theoretical Computer Science KIT, en Alemania, compararon el desempeño de LLBC con las versiones 3.8 y 3.9 de CMBC usando benchmarks de distintos tipos. Para compersar las diferencias en los ajustes entre herramientas CBMC se corrió con - *-bounds-check*, - *-div-by-zero-check*, - *-pointer-check*,

y - *-overflow-check*, mientras que LLBMC se corrió con *llvm-gcc* para convertir los programas de C a LLVM, sin las optimizaciones del compilador y con la configuración de Boolector. Entre los benchmarks había programas en C, C++ y programas con bucles no infinitos. Como resultado de los 175 benchmarks:

- CBMC 3.9 resolvió satisfactoriamente 119(68 %), tomo mas tiempo del permitido en las pruebas o uso mas memoria de la permitida en 8 casos (4,6 %), falló en procesar el input en 12 casos (6,9 %) y dio resultados incorrectos en 36 casos (20,6 %).
- CBMC 3.8 resolvió satisfactoriamente 145(82,9 %), tomó mas tiempo del permitido en las pruebas o usó mas memoria de la permitida en 13 casos (7,4 %), fallo en procesar el input en 6 casos (3,4 %) y dio resultados incorrectos en 11 casos (6,3 %).
- LLBMC resolvió satisfactoriamente 172(98,3 %), tomo mas tiempo del permitido en las pruebas o uso mas memoria de la permitida en 1 caso (0,6 %), fallo en procesar el input en 0 casos y dio resultados incorrectos en 2 casos (1,1 %).

Si comparamos los resultados obtenidos con los tiempos en que cada herramienta los obtuvo, podemos apreciar que no solamente LLBMC obtuvo menos timeouts/lleno memoria, o dio menos errores, sino que además, obtuvo los resultados en menos tiempo que CBMC tanto en la version 3.8 como la 3.9. En aproximadamente 100 segundos, CBMC 3.9 obtuvo aproximadamente 100 resultados, CBMC 3.8 obtuvo aproximadamente 140 resultados, mientras que LLBMC obtuvo cerca de 180 resultados.

4.1. Uso de la herramienta

Para mostrar el uso de la herramienta, tomamos dos programas: uno que calcula la prefix sum en dos dimensiones y otro que usa pthreads para calcular el producto punto de vectores. Nos centraremos en el uso de la herramienta a través de su interfaz de linea de comandos. El código fuente estará disponible en un repositorio nombrado en las referencias[7].

Comenzaremos con el programa *scan2d.c*. CBMC facilita la forma de ver las propiedades de un programa. Escribiendo en consola *cbmc - -show-properties scan2d.c* nos dirá las aserciones que el programa tiene. Una porción de la salida de esta llamada:

```
Generic Property Instrumentation
Property main.assertion.1:
    file scan2d.c line 104 function main
    free argument is dynamic object
    DYNAMIC_OBJECT(ptr)
```

Si queremos comprobar que el programa cumple con las propiedades descriptas, escribimos en la consola *cbmc scan2d.c*. Al correr esto, el programa empezará a analizar todos los posibles estados del programa para verificar que las aserciones se cumplen. Esto puede llegar a ser intratable si el espacio de estados es

muy grande, pero podemos pedir que los bucles se corroboren para una cierta cantidad de iteraciones. Corriendo *cbmc - -unwind 6 - -no-unwinding-assertions scan2d.c* podemos analizar el programa para 6 iteraciones por bucle.

```
size of program expression: 4164 steps
simple slicing removed 22 assignments
Generated 7 VCC(s), 1 remaining after simplification
Passing problem to propositional reduction
converting SSA
Running propositional reduction
Post-processing
Solving with MiniSAT 2.2.0 with simplifier
23239 variables, 156 clauses
SAT checker inconsistent: negated claim is UNSATISFIABLE, i.e., holds
Runtime decision procedure: 0.009s
VERIFICATION SUCCESSFUL
```

El flag de - *-no-unwinding-assertions* permite ignorar problemas de la no satisfactibilidad de una propiedad debido a que no tuvo suficientes ciclos para encontrar una fórmula que la dé como satisfacible. En contraposición, - - *unwinding-assertions* buscará una cota en tiempo de ejecución para asegurar de que suficiente desenrollamiento sea realizado para probar la propiedad.

Podemos acceder a la fórmula que CBMC ingresará al SAT Solver con el flag - *-show-vcc*. Una porción de la misma tiene esta forma:

```
{-2438} __CPROVER_memory_leak#4 == NULL
{-2439} __CPROVER_memory_leak#5 == (\guard#3 ? NULL :
__CPROVER_memory_leak#3)
{-2440} ptr!0@1#5 == (void *)dynamic_object2
{-2441} \guard#4 == (__CPROVER_deallocated#5 == (void *)
dynamic_object2)
|------------------------
{1} !\guard#4
```

En caso de que un assert no se cumpla en una fórmula, CBMC nos dará un contraejemplo, podemos además probar solo una propiedad con el flag - - *property* y - *-function* nos dejará probar para una función en particular. Usando el programa *dotprod_mutex.c*, corremos *cbmc - -propierty dotprod.assertion.1 - -function dotprod dotprod_mutex.c* y nos encuentra un contraejemplo de que la propiedad *dotprod.assertion.1* no se cumple en la función *dotprod*.

```
Counterexample:

State 21 file dotprod_mutex.c line 53 thread 0
------------------------------------------------------------
  arg=NULL
  (0000000000000000000000000000000000000000000000000000000000000000)

Violated property:
  file dotprod_mutex.c line 58 function dotprod
  assertion arg != NULL
  arg != (void *)0

VERIFICATION FAILED
```

Además, CBMC cuenta con flags para verificar acceso a memoria, memory leaks, punteros colgantes y puede verificar programas que usan pthreads y probar programas concurrentes, modelando cada hilo.

4.2. Otros usos

Aparte de la aplicación básica de la verificación de programas C, CBMC se utiliza para una amplia variedad de aplicaciones, como:

- Explicación de errores: las versiones extendidas de CBMC pueden encontrar y explicar la causa de un error.
- BMC de programas concurrentes: verificación de programas C ejecutados por múltiples subprocesos.
- Comprobación de equivalencia: verificando que dos programas son equivalentes, en el sentido de que siempre calculan la misma salida.
- Verificación de programas embebidos y modelos que no son de programa: los modelos se formalizan utilizando código C y se verifican utilizando CBMC.
- Verificación de programas existentes, como los controladores de dispositivos Linux y Windows.
- Tiempo de ejecución del peor caso: analizar el tiempo de ejecución de los programas.
- Seguridad: medición de fugas de información en programas, búsqueda de errores de seguridad en binarios de Windows, entre otros.

5. Conclusiones

CBMC es un comprobador de modelos para programas de ANSI-C que usa BMC como método para probar propiedades de un programa dado. El mismo genera un CFG a partir del código, que luego recorre para generar una fórmula en CNF para combinarla con la negación de la propiedad a probar y pasarla a un SAT Solver. CBMC puede comprobar una gran cantidad de propiedades además de aserciones en un programa, lo cual lo hace útil a la hora de verificar la correctitud del mismo. Sin embargo, el espacio de estados puede ser muy extenso en algunas aplicaciones, lo cual lo hace muy costoso de emplear sin el correcto cuidado de las restricciones del problema.

La herramienta es poderosa, siendo capaz de encontrar numerósos problemas en sistemas dada la correcta instrumentación, pero la lectura de contraejemplos a propiedades que no se satisfacen puede ser muy compleja a simple vista, requiriendo un aprendizaje y familiarización con la misma. Cabe destacar que es posible que la herramienta no detecte errores en el código, por lo que esta no podrá afirmar la correctitud del programa completo, sino, hasta una cierta profundidad.

Si bien existen herramientas más eficientes, CBMC es una muy buena elección para adentrarse en el BMC y aprender acerca de ello, ya que facilita las estructuras internas con las que trabaja.

Referencias

1. Edmund Clark, Daniel Kroening y Karen Yorav. *Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking.*

2. Edmund Clarke, Daniel Kroening y Flavio Lerda. *A Tool for Checking ANSI-C Programs.*
3. Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman y Yunshan Zhu. *Bounded Model Checking.*
4. Daniel Kroening y Michael Tautschnig. *CBMC – C Bounded Model Checker (Competition Contribution).*
5. Florian Merz, Stephan Falke, and Carsten Sinz. *LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR.* Institute for Theoretical Computer Science Karlsruhe Institute of Technology (KIT) Germany.
6. Sitio web de plug-in CProver suite: http://www.cprover.org/visual-studio
7. Código fuente de los programas usados en este documento: https://github.com/maurygreen/CBMC-c-digo-de-ejemplos.