CBMC: Bounded Model Checking for C/C++

Ivan Devyaterikov M3307

Spring 2025

Введение в СВМС

CBMC

CBMC (C Bounded Model Checker) - это инструмент для автоматической формальной верификации программ, написанных на языках C и C++.

Bounded model checking - анализ всех возможных путей выполнения программы в пределах ограниченного числа шагов (итераций/циклов).

Проверяет корректность кода: Отсутствие ошибок (указатели, утечки памяти, переполнение и т.п.). Соответствие спецификациям (assertions, контракты).

Технические характеристики :

- почти любой m-SAT солвер
- C89, C99, C11, C17, C23
- Широкое использование в экосистеме AWS (популярная библиотека aws-c-common)

CBMC достаточно простой для обывателя, но под капотом очень сложно интересно. Давайте углубимся в дивный мир

$2 + 2 \neq 4$?

- `--no-signed-overflow-check` - выключает проверку на переполнение
- `--compact-trace` - в случае UNSAT, выводит контрпример

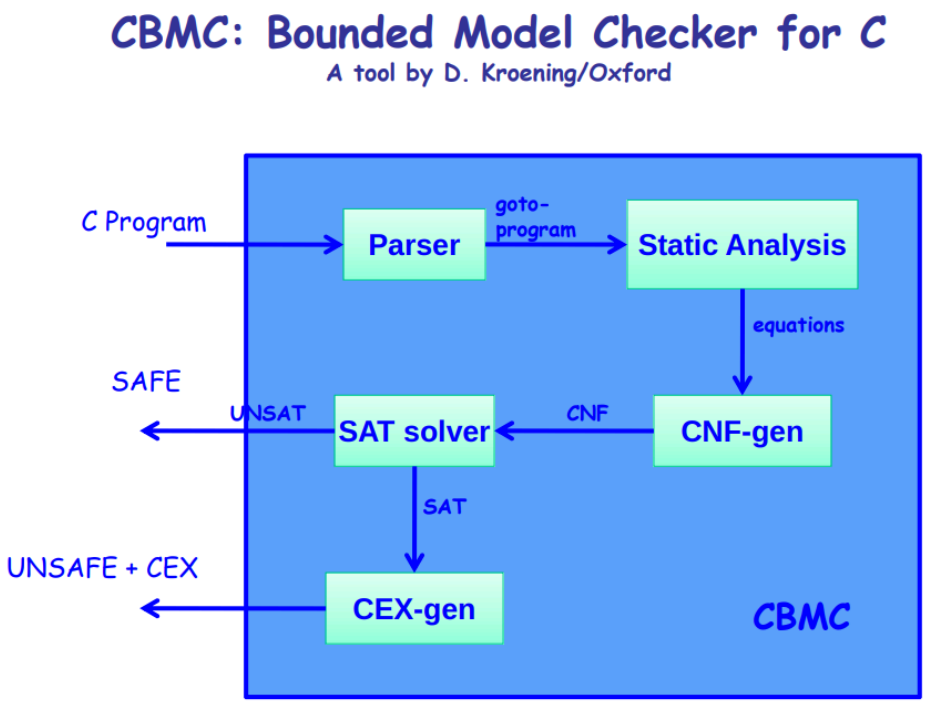
```c
#include <assert.h>

int sum(int a, int b) {
  return a + b;
}

int main() {
  int a = 2;
  int b = 2;
  assert(sum(a, b) != 4);
}
```

```shell
cbmc simple.c --verbosity 4 --no-signed-overflow-check --compact-trace
** Results:
kek.c function main
[main.assertion.1] line 10 assertion sum(a, b) != 4: FAILURE
Trace for main.assertion.1:
↳ kek.c:7 main()
  8: a=2 (00000000 00000000 00000000 00000010)
  9: b=2 (00000000 00000000 00000000 00000010)
↳ kek.c:10 sum(2, 2)
  4: goto_symex$$return_value$$sum=4 (00000000 00000000 00000000 00000100)
↵
  10: return_value_sum=4 (00000000 00000000 00000000 00000100)
Violated property:
  file kek.c function main line 10 thread 0
  assertion sum(a, b) != 4
  FALSE
VERIFICATION FAILED
```

Схема работы



Состояние программы

Состояние программы

Состояние программы — это совокупность всех значений переменных, регистров, памяти и других данных, которые определяют текущее положение программы в момент выполнения. Упрощено, состояние программы включает: Значения всех переменных, глобальные + локальные Текущую точку выполнения Состояние стека вызовов, чтобы понять куда вернуться из функции

```c
int x = 5; // {x=5}
int y = x + 1; // {x=5, y=6}
```

Рост состояний программы

К сожалению, очень много состояний получается из-за, очень простых конструкций :

- undefined behavior - в стандарте c/c++ неопределенное действие. Пример, деление на 0
- неинициализированная память. Пример : `int x;` $x \in [\text{INT\_MIN}, \text{INT\_MAX}]$
- `x=rand(a, b)` $x \in [a, b]$
- `x=input()` $x \in \text{somerange}$
- гонки в многопоточных/асинхронных программах. зависит от примера

Циклы

CBMC & unwinding

Как следует из название, bounded - ограничение. В CBMC это ограничение итераций цикла сверху. Для циклов принято решение повторить их неполное число раз. Проблемы, которые решает unwinding :

- Большое множество состояний до цикла
- Недетерминированное число шагов
- Halting Problem (цикл может не остановиться вовсе)
- Асимптотически цикл усложняет проверку

```c
while(cond) {
   // BODY CODE
}

if(cond) {
    // BODY CODE COPY 1
    if(cond) {
      // BODY CODE COPY 2
      if(cond) {
        // BODY CODE COPY 3
        if(cond) {
          // BODY CODE COPY 4
          if(cond) {
            // BODY CODE COPY 5
          }
        }
      }
    }
  }
```

unwinding пример

```c
#include<assert.h>

int main() {
  unsigned bound;
  unsigned array[bound];

  for (int i = 0; i < bound; i++) {
    array[i] = 0;
  }

  for (int i = 0; i < bound; i++) {
    assert(array[i] == 0);
  }
  return 0;
}
```

```
cbmc --unwind 5 --verbosity 4 loop.c
** Results:
loop.c function main
[main.overflow.1] line 7 arithmetic overflow on signed + in i + 1: SUCCESS
[main.unwind.0] line 7 unwinding assertion loop 0: FAILURE
[main.array_bounds.1] line 8 array 'array' lower bound in array[(signed long int)i]: SUCCESS
[main.array_bounds.2] line 8 array 'array' upper bound in array[(signed long int)i]: SUCCESS
[main.overflow.2] line 11 arithmetic overflow on signed + in i + 1: SUCCESS
[main.unwind.1] line 11 unwinding assertion loop 1: SUCCESS
[main.array_bounds.3] line 12 array 'array' lower bound in array[(signed long int)i]: SUCCESS
[main.array_bounds.4] line 12 array 'array' upper bound in array[(signed long int)i]: SUCCESS
[main.assertion.1] line 12 assertion array[i] == 0: SUCCESS

** 1 of 9 failed (2 iterations)
VERIFICATION FAILED
```

Неявные циклы

Циклы могут быть “спрятаны” в:

- Рекурсии (анализ требует учёта глубины вызовов),
- Библиотечных функциях (например, strcpy содержит цикл),
- Динамической диспетчеризации (полиморфизм в ООП).
- В пример на слайде можно добавить цикл в Y::ok() и уже будет сложно

C++

```cpp
#include <cassert>

class X {
public:
  virtual bool ok() { return true; }
};

class Y : public X {
public:
  virtual bool ok() { return false; }
};
```

goto program

goto-cc - утилита из набора cbmc. По сути делает обычное IR SSA представление. В это представление потом легко добавить дополнительные проверки и вообще IR переделать в SAT, собственно что и будет потом IR - промежуточное представление SSA - статическом одноадресном присваивании

Дополнительно про IR SSA Статья на habr.com

C

```c
int foo() {
    return 2;
}

int main() {
    int a;
    if (a) {
        return 1;
    } else {
        return foo();
    }
}
```

Shell

```
cbmc main.c --show-goto-functions
foo /* foo */
        // 17 file main.c line 2 function foo
        SET RETURN VALUE 2
        // 18 file main.c line 3 function foo
        END_FUNCTION
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
main /* main */
        // 0 file main.c line 6 function main
        DECL main::1::a : signedbv[32]
        // 1 file main.c line 7 function main
        IF ¬(main::1::a ≠ 0) THEN GOTO 1
        // 2 file main.c line 8 function main
        SET RETURN VALUE 1
        // 3 file main.c line 8 function main
        DEAD main::1::a
        // 4 file main.c line 8 function main
        GOTO 2
        // 5 file main.c line 10 function main
     1: DECL main::$tmp::return_value_foo : signedbv[32]
        // 6 file main.c line 10 function main
        CALL main::$tmp::return_value_foo := foo()
        // 7 file main.c line 10 function main
        SET RETURN VALUE main::$tmp::return_value_foo
        // 8 file main.c line 10 function main
        DEAD main::$tmp::return_value_foo
        // 9 file main.c line 10 function main
        DEAD main::1::a
        // 10 file main.c line 12 function main
     2: END_FUNCTION
```

m-SAT

Bit-blasting

Bit-blasting - это техника, используемая в SMT решателях для преобразования операций над битовыми векторами (bit-vectors) в эквивалентную булеву формулу, которую можно решить с помощью mSAT-решателя.

- `int` представляется как `32` битный вектор. Указатели на `64`-битной архитектуре, представляются как `64` битный вектор.
- операции (арифметические, побитовые, сравнения) разбиваются на булевы операции (И, ИЛИ, НЕ, XOR).
- арифметика реализуется через логические вентили с переносами, сравнения – через побитовые проверки.

Bit-blasting для CBMC

Хотя SMT-решатели поддерживают теорию целых чисел (например, в SMT-LIB2: (`declare-fun a () Int`)).

CBMC использует битовые векторы вместо теории целых чисел, потому что:

- Поддерживает переполнение (невозможное в теории целых).
- Точнее отражает поведение программ.
- Не все SMT-решатели хорошо поддерживают теорию целых чисел.
- Доступ к памяти моделируется через битовые векторы (адреса и данные).

SAT solver

CBMC по goto программе генерирует например smt2 представление и даёт его на вход z3, далее CBMC достаточно уметь интерпретировать UNSAT и выводить trace.

Поддерживаются : CVC3/4/5, MathSAT, Yices, Z3 (по умолчанию)

Никто не запрещает подключить свой солвер к CBMC.

SMT2 пример

```Shell
cbmc simple.c --smt2 --outfile output.smt2
```

```
; SMT 2
(set-info :source "Generated by CBMC 6.6.0 (cbmc-6.6.0)")
(set-option :produce-models true)
(set-logic QF_AUFBV)

; set_to true (equal)
(define-fun |__CPROVER_dead_object#1| () (_ BitVec 64) (_ bv0 64))

; set_to true (equal)
(define-fun |__CPROVER_deallocated#1| () (_ BitVec 64) (_ bv0 64))

; set_to true (equal)
(define-fun |__CPROVER_memory_leak#1| () (_ BitVec 64) (_ bv0 64))

; set_to true (equal)
(define-fun |__CPROVER_rounding_mode#1| () (_ BitVec 32) (_ bv0 32))

; set_to true (equal)
(define-fun |__CPROVER::constant_infinity_uint#1| () (_ BitVec 32) (_ bv0
32))

; set_to true (equal)
(define-fun |main::1::a!0@1#2| () (_ BitVec 32) (_ bv2 32))

; set_to true (equal)
(define-fun |main::1::b!0@1#2| () (_ BitVec 32) (_ bv2 32))

; find_symbols
(declare-fun |main::1::a!0@1#1| () (_ BitVec 32))
; convert
; Converting var_no 0 with expr ID of =
(define-fun B0 () Bool (= |main::1::a!0@1#1| |main::1::a!0@1#1|))

; convert
; Converting var_no 1 with expr ID of =
(define-fun B1 () Bool (= |main::1::a!0@1#1| |main::1::a!0@1#1|))

; find_symbols
(declare-fun |main::1::b!0@1#1| () (_ BitVec 32))
; convert
; Converting var_no 2 with expr ID of =
(define-fun B2 () Bool (= |main::1::b!0@1#1| |main::1::b!0@1#1|))

; convert
; Converting var_no 3 with expr ID of =
```

```
46 (define-fun B3 () Bool (= |main::1::b!0@1#1| |main::1::b!0@1#1|))
47
48 ; set_to false
49 (assert (not false))
50
51 ; convert
52 ; Converting var_no 4 with expr ID of not
53 (define-fun B4 () Bool (not false))
54
55 ; set_to true
56 (assert B4)
57
58 (check-sat)
59
60 (get-value (B0))
61 (get-value (B1))
62 (get-value (B2))
63 (get-value (B3))
64 (get-value (B4))
65 (get-value (|__CPROVER::constant_infinity_uint#1|))
66 (get-value (|__CPROVER_dead_object#1|))
67 (get-value (|__CPROVER_deallocated#1|))
68 (get-value (|__CPROVER_memory_leak#1|))
69 (get-value (|__CPROVER_rounding_mode#1|))
70 (get-value (|main::1::a!0@1#1|))
71 (get-value (|main::1::a!0@1#2|))
72 (get-value (|main::1::b!0@1#1|))
73 (get-value (|main::1::b!0@1#2|))
```

```
74
75 (exit)
76 ; end of SMT2 file
```

Память

Работа с памятью

```c
int main() {
  unsigned array[10];
  char *p;
  p = (char *)(array + 1);
  p++;
  assert(__CPROVER_POINTER_OFFSET(p) == 5);
  assert(__CPROVER_POINTER_OBJECT(array) == 2);
  assert(__CPROVER_POINTER_OBJECT(p) == 2);
  return 0;
}
```

Указатели и работа с памятью является самой опасной в c/c++. Не зря в c++ люди попытались отойти от них, применив умные указатели и другие техники.

CBMC представляет память, как абстрактное хранилище. Любой ненулевой указатель соотнесен с каким либо объектом. CBMC дополнительно хранит ещё и сдвиг относительно начала этого объекта. Получается указатель в CBMC - пара.

CPROVER

__CPROVER

CPROVER — это платформа для формальной верификации программного обеспечения, разрабатываемая командой Diffblue. CMBC и goto-cc утилиты из этого набора.

CPROVER - префикс макросов, для проверки. Компилятор обычный не знает про них и выдаёт ошибку, но для CMBC это очень важный инструмент, с помощью него можно помогать верификатору.

Например, `__CPROVER_assume(0 <= n && n <= 10)` даёт ограничение в самом MSAT

- `__CPROVER_POINTER_OBJECT` ранее рассматривали
- `__CPROVER_assume` и `__CPROVER_havoc_object` рассмотрим далее

__CPROVER_assume

```c
#include <assert.h>

int main() {
  unsigned bound;
  unsigned array[bound];

  __CPROVER_assume(bound < 5);
  for (int i = 0; i < bound; i++) {
    array[i] = 0;
  }

  for (int i = 0; i < bound; i++) {
    assert(array[i] == 0);
  }
  return 0;
}
```

```shell
cbmc --unwind 5 --verbosity 4 loop.c
** Results:
loop.c function main
[main.overflow.1] line 8 arithmetic overflow on signed + in i + 1: SUCCESS
[main.unwind.0] line 8 unwinding assertion loop 0: SUCCESS
[main.array_bounds.1] line 9 array 'array' lower bound in array[(signed long int)i]: SUCCESS
[main.array_bounds.2] line 9 array 'array' upper bound in array[(signed long int)i]: SUCCESS
[main.overflow.2] line 12 arithmetic overflow on signed + in i + 1: SUCCESS
[main.unwind.1] line 12 unwinding assertion loop 1: SUCCESS
[main.array_bounds.3] line 13 array 'array' lower bound in array[(signed long int)i]: SUCCESS
[main.array_bounds.4] line 13 array 'array' upper bound in array[(signed long int)i]: SUCCESS
[main.assertion.1] line 13 assertion array[i] == 0: SUCCESS

** 0 of 9 failed (1 iterations)
VERIFICATION SUCCESSFUL
```

__CPROVER_havoc_object

```c
struct foo {
  int x;
  int y;
};
int main() {
  struct foo thefoo = {.x = 1, .y = 2};
  int *p = &thefoo.y;
  __CPROVER_havoc_object(p); // makes the whole struct nondet
  __CPROVER_assert(thefoo.x == 1, "fails because `thefoo.x` is now nondet");
  __CPROVER_assert(thefoo.y == 2, "fails because `thefoo.y` is now nondet");
  return 0;
}
```

This function requires a valid pointer and updates all bytes of the underlying object with nondeterministic values.

```
[main.assertion.1] line 9 fails because `thefoo.x` is now nondet: FAILURE
[main.assertion.2] line 10 fails because `thefoo.y` is now nondet: FAILURE

Trace for main.assertion.1:

↳ kek.c:5 main()
  6: thefoo.x=1 (00000000 00000000 00000000 00000001)
  6: thefoo.y=2 (00000000 00000000 00000000 00000010)
  7: p=&thefoo!0@1.y (00000010 00000000 00000000 00000000 00000000 00000000 00000000 00000100)
  8: thefoo={ .x=536870913, .y=8388610 } ({ 00100000 00000000 00000000 00000001, 00000000 10000000 00000000 00000010 })
  8: thefoo.x=536870913 (00100000 00000000 00000000 00000001)
  8: thefoo.y=8388610 (00000000 10000000 00000000 00000010)

Violated property:
  file kek.c function main line 9 thread 0
  fails because `thefoo.x` is now nondet
  thefoo.x == 1
...
VERIFICATION FAILED
```

__CPROVER_PRINT своими руками

C

```c
#define __CPROVER_print(var) { int value_of_##var = (int) var; }

void foo(int x) {
  __CPROVER_print(x);
  assert(0);
}

int main() {
  foo(3);
}
```

Shell

```
** Results:
kek.c function foo
[foo.assertion.1] line 5 assertion 0: FAILURE

Trace for foo.assertion.1:

↳ kek.c:8 main()

↳ kek.c:9 foo(3)
  4: value_of_x=3 (00000000 00000000 00000000 00000011)

Violated property:
  file kek.c function foo line 5 thread 0
  assertion 0
  (__CPROVER_bool)0


VERIFICATION FAILED
```

Ограничения

Ограничения CBMC

Проверить код можно не весь. Есть нюансы, в которых проверка будет не эффективна :

1. Проверка может быть трудозатратная по времени
2. Раскрутка циклов или любые другие опущения плохо сказываются на проверке:

- Проверяются **не все варианты**, в которых возможно была бы ошибка.
- CBMC может давать false positive (ложное срабатывание) или true negative (не находить ошибок)

False positive

```c
#include <stdbool.h>

int main() {
  int bound;
  for (int i = 0; i < bound; i++) {
    if (i > 5) {
      assert(false);
    }
  }
}
```

```shell
cbmc --unwind 6 loop.c --no-unwinding-assertions
**** WARNING: Use --unwinding-assertions to obtain sound verification results
…
** Results:
loop.c function main
[main.overflow.1] line 5 arithmetic overflow on signed + in i + 1: SUCCESS
[main.assertion.1] line 7 assertion 0: SUCCESS

** 0 of 2 failed (1 iterations)
VERIFICATION SUCCESSFUL
~/cbmc> cbmc --unwind 7 loop.c --no-unwinding-assertions
**** WARNING: Use --unwinding-assertions to obtain sound verification results
…
** Results:
loop.c function main
[main.overflow.1] line 5 arithmetic overflow on signed + in i + 1: SUCCESS
[main.assertion.1] line 7 assertion 0: FAILURE

** 1 of 2 failed (2 iterations)
VERIFICATION FAILED
```

Применение

Верификация в `aws-c-common`

Разберём реализацию `memcpy` в `awc-c-common`. Ссылка на гитхаб

C

```c
void *memcpy_impl(void *dst, const void *src, size_t n) {
    __CPROVER_precondition(
        __CPROVER_POINTER_OBJECT(dst) != __CPROVER_POINTER_OBJECT(src) ||
            ((const char *)src >= (const char *)dst + n) || ((const char *)dst >= (const char *)src + n),
        "memcpy src/dst overlap");
    __CPROVER_precondition(src != NULL && __CPROVER_r_ok(src, n), "memcpy source region readable");
    __CPROVER_precondition(dst != NULL && __CPROVER_w_ok(dst, n), "memcpy destination region writeable");

    if (n > 0) {
        size_t index;
        __CPROVER_assume(index < n);
        ((uint8_t *)dst)[index] = nondet_uint8_t();
    }
    return dst;
}
```

Пример верификации

```c
int nondet_int();

int abs(int x) { return x < 0 ? -x : x; }

int abs_bits(int x) {
  int m = x >> 31;
  return (x ^ m) + ~m + 1;
}

int main() {
  int t = nondet_int();
  int t_abs = abs(t);
  int t_abs_bits = abs_bits(t);
  __CPROVER_assert(t_abs == t_abs_bits, "ans_bits=ans_ref");
}
```

```shell
cbmc bit.c --verbosity 4 --no-signed-overflow-check

** Results:
bit.c function main
[main.assertion.1] line 14 ans_bits=ans_ref: SUCCESS

** 0 of 1 failed (1 iterations)
VERIFICATION SUCCESSFUL
```

Пример с работы

Представим, язык X, на котором написаны критичные расчёты, например цены на услугу.

В момент редактирования формул, хочется понять, а не сломается что-либо на всевозможных входных вариантах.

Обычное решение : можно взять и написать юнитесты. К сожалению тут проверяется конкретные входные данные, а не всевозможные. Например, у вас есть входной параметр, как текущее время в секундах, вряд ли вы сможете написать $24 \cdot 60 \cdot 60$ тестов.

Решение с CBMC : Если сделать такой-же юнитест, но добавить `nondet_int` в параметр времени, то тест будет проверяться на все возможных вариантах.

Заключение

Заключение

- Рассмотрели базовое применение CBMC для задач верификации кода на `c/c++`.
- Поэтапно разобрали принцип работы CBMC
- Применение в реальных проектах.
- Рассмотрели случий с false positive
- Рассмотрели подель памяти
- `__CPROVER`