

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

```

1  #include <assert.h>
2
3  int sum(int a, int b) {
4      return a + b;
5  }
6
7  int main() {
8      int a = 2;
9      int b = 2;
10     assert(sum(a, b) != 4);
11 }

```

```

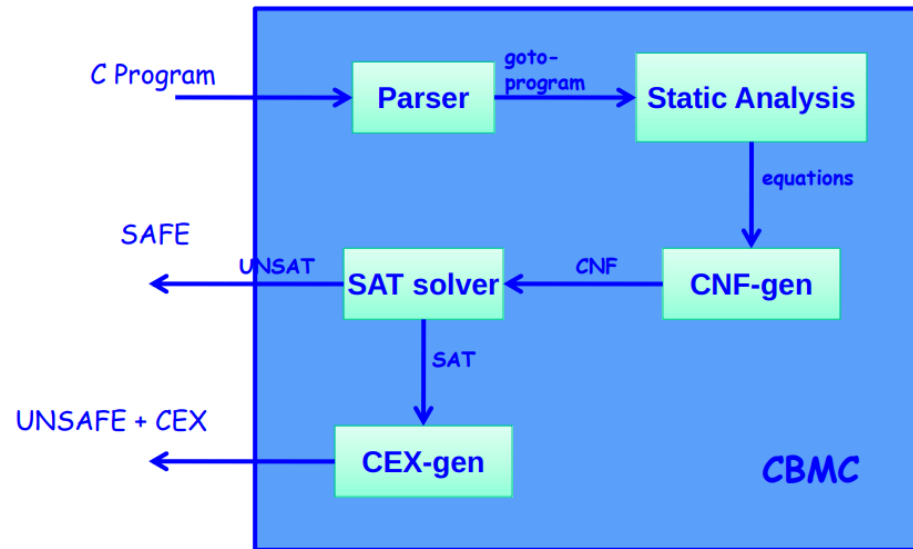
1  cbmc simple.c --verbosity 4 --no-signed-overflow-check --compact-trace
2  ** Results:
3  kek.c function main
4  [main.assertion.1] line 10 assertion sum(a, b) != 4: FAILURE
5  Trace for main.assertion.1:
6  ↳ kek.c:7 main()
7      8: a=2 (00000000 00000000 00000000 00000010)
8      9: b=2 (00000000 00000000 00000000 00000010)
9  ↳ kek.c:10 sum(2, 2)
10     4: goto_symex$$return_value$$sum=4 (00000000 00000000 00000000 00000100)
11  ↳
12     10: return_value_sum=4 (00000000 00000000 00000000 00000100)
13  Violated property:
14     file kek.c function main line 10 thread 0
15     assertion sum(a, b) != 4
16     FALSE
17  VERIFICATION FAILED

```

Схема работы

CBMC: Bounded Model Checker for C

A tool by D. Kroening/Oxford



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

```
1 int x = 5; // {x=5}
2 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 ∈ [INT_MIN, INT_MAX]`
- `x=rand(a, b)` $x ∈ [a, b]$
- `x=input()` $x ∈ \text{somewhere}$
- 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

```
1  while(cond) {
2      // BODY CODE
3  }
4
5  if(cond) {
6      // BODY CODE COPY 1
7      if(cond) {
8          // BODY CODE COPY 2
9          if(cond) {
10             // BODY CODE COPY 3
11             if(cond) {
12                 // BODY CODE COPY 4
13                 if(cond) {
14                     // BODY CODE COPY 5
15                 }
16             }
17         }
18     }
19 }
```

unwinding пример

```
1  #include<assert.h>
2
3  int main() {
4      unsigned bound;
5      unsigned array[bound];
6
7      for (int i = 0; i < bound; i++) {
8          array[i] = 0;
9      }
10
11     for (int i = 0; i < bound; i++) {
12         assert(array[i] == 0);
13     }
14     return 0;
15 }
```

```
1  cbmc --unwind 5 --verbosity 4 loop.c
2  ** Results:
3  loop.c function main
4  [main.overflow.1] line 7 arithmetic overflow on signed + in i + 1: SUCCESS
5  [main.unwind.0] line 7 unwinding assertion loop 0: FAILURE
6  [main.array_bounds.1] line 8 array 'array' lower bound in array[(signed long int)i]:
   SUCCESS
7  [main.array_bounds.2] line 8 array 'array' upper bound in array[(signed long int)i]:
   SUCCESS
8  [main.overflow.2] line 11 arithmetic overflow on signed + in i + 1: SUCCESS
9  [main.unwind.1] line 11 unwinding assertion loop 1: SUCCESS
10 [main.array_bounds.3] line 12 array 'array' lower bound in array[(signed long int)i]:
   SUCCESS
11 [main.array_bounds.4] line 12 array 'array' upper bound in array[(signed long int)i]:
   SUCCESS
12 [main.assertion.1] line 12 assertion array[i] == 0: SUCCESS
13
14 ** 1 of 9 failed (2 iterations)
15 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

```
1  #include <cassert>
2
3  class X {
4  public:
5      virtual bool ok() { return true; }
6  };
7
8  class Y : public X {
9  public:
10     virtual bool ok() { return false; }
11  };
```

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](#)

```

1  int foo() {
2      return 2;
3  }
4
5  int main() {
6      int a;
7      if (a) {
8          return 1;
9      } else {
10         return foo();
11     }
12 }
```

```

1  cbmc main.c --show-goto-functions
2  foo /* foo */
3      // 17 file main.c line 2 function foo
4      SET RETURN VALUE 2
5      // 18 file main.c line 3 function foo
6      END_FUNCTION
7      ~~~~~
8  main /* main */
9      // 0 file main.c line 6 function main
10     DECL main::l::a : signedbv[32]
11     // 1 file main.c line 7 function main
12     IF ¬(main::l::a ≠ 0) THEN GOTO 1
13     // 2 file main.c line 8 function main
14     SET RETURN VALUE 1
15     // 3 file main.c line 8 function main
16     DEAD main::l::a
17     // 4 file main.c line 8 function main
18     GOTO 2
19     // 5 file main.c line 10 function main
20     1: DECL main::$tmp::return_value_foo : signedbv[32]
21     // 6 file main.c line 10 function main
22     CALL main::$tmp::return_value_foo := foo()
23     // 7 file main.c line 10 function main
24     SET RETURN VALUE main::$tmp::return_value_foo
25     // 8 file main.c line 10 function main
26     DEAD main::$tmp::return_value_foo
27     // 9 file main.c line 10 function main
28     DEAD main::l::a
29     // 10 file main.c line 12 function main
30     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

```
1  cbmc simple.c --smt2 --outfile output.smt2
2  ; SMT 2
3  (set-info :source "Generated by CBMC 6.6.0 (cbmc-6.6.0)")
4  (set-option :produce-models true)
5  (set-logic QF_AUFBV)
6
7  ; set_to true (equal)
8  (define-fun |__CPROVER_dead_object#1| () (_ BitVec 64) (_ bv0 64))
9
10 ; set_to true (equal)
11 (define-fun |__CPROVER_deallocated#1| () (_ BitVec 64) (_ bv0 64))
12
13 ; set_to true (equal)
14 (define-fun |__CPROVER_memory_leak#1| () (_ BitVec 64) (_ bv0 64))
15
16 ; set_to true (equal)
17 (define-fun |__CPROVER_rounding_mode#1| () (_ BitVec 32) (_ bv0 32))
18
19 ; set_to true (equal)
20 (define-fun |__CPROVER::constant_infinity_uint#1| () (_ BitVec 32) (_ bv0
32))
21
22 ; set_to true (equal)
```

```
23 (define-fun |main::1::a!0@1#2| () (_ BitVec 32) (_ bv2 32))
24
25 ; set_to true (equal)
26 (define-fun |main::1::b!0@1#2| () (_ BitVec 32) (_ bv2 32))
27
28 ; find_symbols
29 (declare-fun |main::1::a!0@1#1| () (_ BitVec 32))
30 ; convert
31 ; Converting var_no 0 with expr ID of =
32 (define-fun B0 () Bool (= |main::1::a!0@1#1| |main::1::a!0@1#1|))
33
34 ; convert
35 ; Converting var_no 1 with expr ID of =
36 (define-fun B1 () Bool (= |main::1::a!0@1#1| |main::1::a!0@1#1|))
37
38 ; find_symbols
39 (declare-fun |main::1::b!0@1#1| () (_ BitVec 32))
40 ; convert
41 ; Converting var_no 2 with expr ID of =
42 (define-fun B2 () Bool (= |main::1::b!0@1#1| |main::1::b!0@1#1|))
43
44 ; convert
45 ; Converting var_no 3 with expr ID of =
```

m-SAT

```
46 (define-fun B3 () Bool (= |main::l::b!0@1#1| |main::l::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::l::a!0@1#1|))
71 (get-value (|main::l::a!0@1#2|))
72 (get-value (|main::l::b!0@1#1|))
73 (get-value (|main::l::b!0@1#2|))
```

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

Память

Работа с памятью

```
1  int main() {  
2      unsigned array[10];  
3      char *p;  
4      p = (char *) (array + 1);  
5      p++;  
6      assert(__CPROVER_POINTER_OFFSET(p) == 5);  
7      assert(__CPROVER_POINTER_OBJECT(array) == 2);  
8      assert(__CPROVER_POINTER_OBJECT(p) == 2);  
9      return 0;  
10 }
```

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

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

```

1  #include <assert.h>
2
3  int main() {
4      unsigned bound;
5      unsigned array[bound];
6
7      __CPROVER_assume(bound < 5);
8      for (int i = 0; i < bound; i++) {
9          array[i] = 0;
10     }
11
12     for (int i = 0; i < bound; i++) {
13         assert(array[i] == 0);
14     }
15     return 0;
16 }

```



```

1  cbmc --unwind 5 --verbosity 4 loop.c
2  ** Results:
3  loop.c function main
4  [main.overflow.1] line 8 arithmetic overflow on signed + in i + 1: SUCCESS
5  [main.unwind.0] line 8 unwinding assertion loop 0: SUCCESS
6  [main.array_bounds.1] line 9 array 'array' lower bound in array[(signed long
   int)i]: SUCCESS
7  [main.array_bounds.2] line 9 array 'array' upper bound in array[(signed long
   int)i]: SUCCESS
8  [main.overflow.2] line 12 arithmetic overflow on signed + in i + 1: SUCCESS
9  [main.unwind.1] line 12 unwinding assertion loop 1: SUCCESS
10 [main.array_bounds.3] line 13 array 'array' lower bound in array[(signed long
   int)i]: SUCCESS
11 [main.array_bounds.4] line 13 array 'array' upper bound in array[(signed long
   int)i]: SUCCESS
12 [main.assertion.1] line 13 assertion array[i] == 0: SUCCESS
13
14 ** 0 of 9 failed (1 iterations)
15 VERIFICATION SUCCESSFUL

```



__CPROVER_havoc_object

```

1  struct foo {
2      int x;
3      int y;
4  };
5  int main() {
6      struct foo thefoo = {.x = 1, .y = 2};
7      int *p = &thefoo.y;
8      __CPROVER_havoc_object(p); // makes the whole struct nondet
9      __CPROVER_assert(thefoo.x == 1, "fails because `thefoo.x` is now
   nondet");
10     __CPROVER_assert(thefoo.y == 2, "fails because `thefoo.y` is now
   nondet");
11     return 0;
12 }

```

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

```

1  [main.assertion.1] line 9 fails because `thefoo.x` is now
   nondet: FAILURE
2  [main.assertion.2] line 10 fails because `thefoo.y` is now nondet:
   FAILURE
3
4  Trace for main.assertion.1:
5
6  ↳ kek.c:5 main()
7      6: thefoo.x=1 (00000000 00000000 00000000 00000001)
8      6: thefoo.y=2 (00000000 00000000 00000000 00000010)
9      7: p=&thefoo!0@1.y (00000010 00000000 00000000 00000000 00000000
   00000000 00000000 00000100)
10     8: thefoo={ .x=536870913, .y=8388610 } ({ 00100000 00000000 00000000
   00000001, 00000000 10000000 00000000 00000010 })
11     8: thefoo.x=536870913 (00100000 00000000 00000000 00000001)
12     8: thefoo.y=8388610 (00000000 10000000 00000000 00000010)
13
14  Violated property:
15      file kek.c function main line 9 thread 0
16      fails because `thefoo.x` is now nondet
17      thefoo.x == 1
18      ...
19  VERIFICATION FAILED

```

__CPROVER_PRINT своими руками

```
1  #define __CPROVER_print(var) { int value_of_##var = (int) var; }
2
3  void foo(int x) {
4      __CPROVER_print(x);
5      assert(0);
6  }
7
8  int main() {
9      foo(3);
10 }
```

```
1  ** Results:
2  kek.c function foo
3  [foo.assertion.1] line 5 assertion 0: FAILURE
4
5  Trace for foo.assertion.1:
6
7  ↳ kek.c:8 main()
8
9  ↳ kek.c:9 foo(3)
10  4: value_of_x=3 (00000000 00000000 00000000 00000011)
11
12  Violated property:
13  file kek.c function foo line 5 thread 0
14  assertion 0
15  (__CPROVER_bool)0
16
17
18  VERIFICATION FAILED
```

Ограничения

Ограничения СВМС

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.
 - СВМС может давать false positive (ложное срабатывание) или true negative (не находить ошибок)

False positive

```
1  #include <stdbool.h>
2
3  int main() {
4      int bound;
5      for (int i = 0; i < bound; i++) {
6          if (i > 5) {
7              assert(false);
8          }
9      }
10 }
```

```
1  cbmc --unwind 6 loop.c --no-unwinding-assertions
2  **** WARNING: Use --unwinding-assertions to obtain sound verification
   results
3  ...
4  ** Results:
5  loop.c function main
6  [main.overflow.1] line 5 arithmetic overflow on signed + in i + 1:
   SUCCESS
7  [main.assertion.1] line 7 assertion 0: SUCCESS
8
9  ** 0 of 2 failed (1 iterations)
10 VERIFICATION SUCCESSFUL
11 ~/cbmc> cbmc --unwind 7 loop.c --no-unwinding-assertions
12 **** WARNING: Use --unwinding-assertions to obtain sound verification
   results
13 ...
14 ** Results:
15 loop.c function main
16 [main.overflow.1] line 5 arithmetic overflow on signed + in i + 1:
   SUCCESS
17 [main.assertion.1] line 7 assertion 0: FAILURE
18
19 ** 1 of 2 failed (2 iterations)
20 VERIFICATION FAILED
```

Применение

Верификация в aws-c-common

Разберём реализацию memcpy в aws-c-common. [Ссылка на гитхаб](#)

```
1 void *memcpy_impl(void *dst, const void *src, size_t n) {
2     __CPROVER_precondition(
3         __CPROVER_POINTER_OBJECT(dst) != __CPROVER_POINTER_OBJECT(src) ||
4         ((const char *)src >= (const char *)dst + n) || ((const char *)dst >= (const char *)src +
5             n),
6         "memcpy src/dst overlap");
7     __CPROVER_precondition(src != NULL && __CPROVER_r_ok(src, n), "memcpy source region readable");
8     __CPROVER_precondition(dst != NULL && __CPROVER_w_ok(dst, n), "memcpy destination region
9         writeable");
10
11     if (n > 0) {
12         size_t index;
13         __CPROVER_assume(index < n);
14         ((uint8_t *)dst)[index] = nondet_uint8_t();
15     }
16     return dst;
17 }
```

Пример верификации

```
1  int nondet_int();
2
3  int abs(int x) { return x < 0 ? -x : x; }
4
5  int abs_bits(int x) {
6      int m = x >> 31;
7      return (x ^ m) + ~m + 1;
8  }
9
10 int main() {
11     int t = nondet_int();
12     int t_abs = abs(t);
13     int t_abs_bits = abs_bits(t);
14     __CPROVER_assert(t_abs == t_abs_bits, "ans_bits=ans_ref");
15 }
```



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



```
2  ** Results:
```

```
3  bit.c function main
```

```
4  [main.assertion.1] line 14 ans_bits=ans_ref:
   SUCCESS
```

```
5
```

```
6  ** 0 of 1 failed (1 iterations)
```

```
7  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 `pondet_int` в параметр времени, то тест будет проверяться на все возможных вариантах.

Заключение

Заключение

- Применение CBMC для базовых задач верификации программ
- Принципы работы CBMC
- Реальный пример из aws - c - common
- Написали несколько примеров самостоятельно, в том числе для битового хака
- Модель памяти
- Рассмотрели случай с false positive
- Рассмотрели модель памяти
- Разобрали __CPROVER, и даже написали свой __CPROVER_PRINT