arXiv:2302.02384v1 [cs.SE] 5 Feb 2023

CBMC
C有界模型检查器

Daniel Kroening、Peter Schrammel 和 Michael Tautschnig

1 引言

C有界模型检查器（CBMC） [9] 展示 C 程序中断言的违反情况，或证明在给定界限下断言的安全性。CBMC 将一个输入 C 程序进行位精确转换：该程序带有断言，并将循环展开至给定深度，然后转换为一个公式。如果该公式可满足，则存在一条会导致某个断言被违反的执行路径。

CBMC 是最成功的软件验证工具之一。它的主要优势在于精确性、稳健性和简便性。CBMC 已作为多个 Linux发行版 的一部分发布。数千名软件开发人员已使用它来验证现实世界中的软件，例如 Linux内核；它还为商业软件分析和测试生成工具提供支持。表 1 概述了 CBMC 的功能。

CBMC 还是一种用途广泛的工具，可用于解决许多实际的程序分析问题，例如错误查找、属性检查、测试输入生成、安全漏洞检测、等价性检查和程序综合。

本章将介绍 CBMC，包括实际示例以及进一步阅读的指针。此外，我们还将介绍 CBMC 本身的开发历程，展示其性能在过去十年中的演变。

第 2 节概述了 CBMC 实现的验证方法。第 3 节介绍如何使用 CBMC 解决各种验证问题。CBMC 的一大优势是已被证明适用于实际的 C 程序；第 4 节

Daniel Kroening 牛津大学，英国，以及 Diffblue Ltd，牛津，英国 电子邮箱：kroening@cs.ox.ac.ukPeter Schrammel 萨塞克斯大学，布莱顿，英国，以及 Diffblue Ltd，牛津，英国 电子邮箱：p.schrammel@sussex.ac.ukMichael Tautschnig 伦敦玛丽女王大学，英国 电子邮箱：michael.tautschnig@qmul.ac.uk

介绍了使这一点成为可能的功能。第 5 节描述了 CBMC 架构的组成部分。第 6 节概述了 CBMC 在过去 10 年中的性能演变，最后在第 7 节作总结。

语言	C语言、GOTO
性质	断言、内存安全、算术溢出、除零、内存泄漏
环境	Linux ，Mac OS ，Windows ，BSD
所用技术	符号执行、有界模型检测、SAT 和 SMT 求解
其他功能	通过 goto-cc 将整个项目编译并链接为 GOTO
优势	内存安全、浮点运算
劣势	对无界验证的支持有限

表 1：CBMC 的功能

软件项目。 CPROVER 框架（包括 CBMC）以 C语言++ 实现，约有 25 万行代码。CPROVER 由 Daniel Kroening 维护，另有 60 多位贡献者参与。它采用 BSD 风格许可证公开发布。可在https://www.cprover.org/cbmc 和 https://github.com/diffblue/cbmc获取适用于常见平台的源代码和二进制文件。详细的安装指南见https://www.cprover.org/cprover-manual/installation/。

2 验证方法

对于给定的 Kripke 结构，有界模型检测 [6, 7] 是一种半判定过程，它将转移关系的有界展开和 LTL 公式转换为命题可满足性问题。通过不断增加界限，直到找到见证，即可保证可靠性；但只有在到达所有状态所需的步数有限时，才能保证完备性。

CBMC 针对软件，特别是 C 程序，实现了有界模型检测。在这种情况下，转移关系由 C 程序以及 C 语言标准 [3],中规定的语义定义，初始状态则由程序的入口点确定。作为规约，程序使用断言进行注释，而不是使用 LTL 公式。转移关系的有界展开相当于程序执行的有限步数。不过，作为一种更实用的有界展开方式，通常通过对循环和递归过程调用进行有限次展开来实现。

通过这种有界展开，CBMC 将关于程序路径的问题转化为约束，并由现成的布尔可满足性（SAT）或可满足性模理论（SMT） [4]求解器求解。对于 SAT 后端，给定带有断言注释的程序，CBMC 会生成一个合取范式公式，其解描述了导致断言违例的程序路径。该公式的一个模型即为一个反例。

```
1 int 绝对值(int
x){2 int y = x;34
if(x < 0) {5 y = -
x;6 }78 return
y;9 }
```

清单 1：源代码（abs.c）

```
1 CBMC 版本 5.12 (cbmc-5.12-d8598f8-557-g1edf4d91f) 64 位 x86_64 macOS2
解析 abs.c3 转换4 类型检查 abs5 生成 GOTO 程序6 添加 CPROVER 库(x86_64)7 移除
函数指针和虚函数8 通用属性插桩9 以 8 个对象位数、56 个偏移位数（默认值）运行10
开始有界模型检验11 程序表达式大小：45 步12 简单切片移除了 0 个赋值13 生成了 0 个
VCC，简化后剩余 0 个14 验证成功
```

清单 2：对 abs.c（清单 1）执行命令 C1 的 CBMC 输出

在详细了解架构之前，我们先来看一个示例。清单 1 中的程序尝试计算一个整数类型输入的绝对值。要在此程序上运行 CBMC，我们需要指定源文件（abs.c）的名称以及入口点 `abs`：

```
cbmc --function abs abs.c
```
C1

输出如清单 2 所示。该输出提供了以下信息：

- 第 1 行报告了正在运行的 CBMC 版本（包括 CBMC 可执行文件的确切 Git 修订版本），以及其运行的平台。
- 第 2–5 行报告了 CBMC 的 C 前端正在处理的源代码。
- 第 6–8 行是插桩步骤的状态更新。
- 第 9 行确认了所使用的（可配置的）指针编码。
- 第 10–13 行是符号执行的状态和统计信息。值得注意的是，此处的输入未生成任何验证条件（VCC）。
- 第 14 行是 CBMC 给出的确凿结论：没有违反任何规约。然而，结合前一行的信息（零个验证条件），这个结论是空洞的：那里

10 开始有界模型检验11 程序表达式大小：46 步12 简单切片移除了 7 个赋值13 生成了 1 个验证条件，简化后剩余 1 个14 将问题提交给命题约简15 正在转换静态单赋值16 正在运行命题约简17 后处理18 使用带简化器的 MiniSAT 2.2.1 求解19 221 个变量，203 个子句20 SAT 检查器：实例可满足21 运行时判定过程：0.00414769 秒2223 ** 结果：24 abs.c 函数 abs25 [abs.overflow.1] 第 5 行：对 -x 中的有符号一元负号执行算术溢出检查：失败2627 ** 1 项失败（2 次迭代）28 验证失败

清单 3：CBMC 对 abs.c（清单 1）执行命令 C2 的输出

CBMC 没有可检查的断言。事实上，源代码中没有包含任何 断言语句。

现在，我们可以插入 `assert` 语句，也可以使用 CBMC 的内置规约。在这种情况下，我们尤其关注算术溢出。让我们再次调用 CBMC，并启用有符号整数溢出断言：

```
cbmc --function abs --signed-overflow-check abs.c                C2
```

现在，我们得到清单 3 所示的输出（省略了开头几行）：最终结论不仅有所不同（“验证失败”），我们还可以看到其他差异：

• 第 11 行现在报告多出一个步骤，第 13 行确认现在已生成一个验证条件，也就是说，我们有一个非空规约。
• 正在生成实际的公式（第 14–17 行）。该公式被传递给 SAT 求解器并由其处理（第 18–20 行），求解器判定该公式可满足。如第 21 行所示，SAT 求解器花费约 4 毫秒计算出一个模型。

• 可满足的公式意味着存在一个被违反的断言，相关信息见第 23–27 行。具体而言，检测到了算术溢出（第 25 行）。
• 总体判定结果总结在输出的最后一行。

布尔验证结果以及对违反的性质的一行摘要，通常不足以帮助软件工程师调试 CBMC 报告的问题。作为模型检查器，CBMC 会在内部计算出完整的反例。

```
** 结果：  abs.c 函数 abs  [abs.overflow.1] 第 8 行：对 -x 中的有符号一元负号执行算术
溢出：失败  abs.overflow.1 的执行轨迹：  状态 17，文件 abs.c，第 1 行，线程 0  ----------
输入 x：-2147483648 (1000000000000000 00000000 00000000)  状态 20，文件 abs.c,
第 1 行，线程 0  ----------  x=-2147483648 (10000000 00000000
00000000 00000000)  状态 21，文件 abs.c，第 2 行，函数 abs，线程 0  ----------
y=0 (00000000 0000000000000000 00000000)  状态 22，文件 abs.c，第 2 行，函数 abs,
线程 0  ----------  y=-2147483648 (10000000 00000000
00000000 00000000)  违反的性质：  文件 abs.c，第 5 行，函数 abs  对 -x 中的有符号一元
负号执行算术溢出    !(x ==-2147483648)  ** 1 项失败（2 次迭代）  验证失败
```

清单 4：CBMC 对 abs.c（清单 1）命令 C3 的输出

对于程序，反例相当于一条执行轨迹。CBMC 可以使用 --trace 命令行选项打印导致断言失败的步骤：

```
cbmc --function abs --signed-overflow-check --trace abs.c
```
C3

现在，我们得到清单 4 所示的输出（同样，省略了开头几行）。

CBMC 现在会打印触发算术溢出的输入值（第 31 行）。该值同时以十进制表示法和二进制表示法打印，并按 8 位字节分组。在本例中，二进制值尤其能说明问题：它是采用 32 位二进制补码时可表示的最小负数。随后，算术溢出发生在 abs.c 的第 5 行，正如 CBMC 反例输出第 45–48 行所报告的那样。

CBMC 按照第 5 节图 2 所示的流水线实现上述步骤。在该节中，我们将详细解析 CBMC 的各个组成部分，以了解 CBMC 如何得出结果。

3 使用 CBMC

CBMC 使用断言来描述程序属性。断言是针对程序到达特定程序位置时程序状态所作的规约。程序员通常使用 `assert` 宏编写断言。

除了程序员编写的断言外，CBMC 还可以自动生成针对特定性质的断言，这通常可以免去程序员表达那些在任何行为良好的程序中都应成立的性质。该断言生成器会执行保守静态分析，以确定可能包含程序错误的程序位置。由于静态分析并不精确，必须强调，这些生成的断言仅表示潜在的程序错误，并且模型检查器还需要先确认它们确实是实际存在的程序错误。

断言生成器支持对以下性质进行后续验证：

• 缓冲区溢出。对于每次数组访问，检查是否违反上界和下界。

• 指针安全性。查找对 `NULL` 指针或其他无效指针的解引用。

• 内存泄漏。检查程序是否构造了随后无法访问的动态分配数据结构。

• 除以零。检查程序中是否存在除以零的情况。
• 非数。检查浮点计算是否可能产生非数值。
• 算术溢出。检查算术运算或类型转换期间是否发生数值溢出。

• 未定义移位。检查是否存在移位距离过大的情况。

上述所有性质都是可达性属性。它们始终采用以下形式：“程序中是否存在一条路径，使得某项性质遭到违反？”此类性质的反例始终是程序路径。逐步查看这些反例，类似于调试程序。

3.1 循环处理

由于 CBMC 执行有界模型检测，因此所有循环都必须具有有限的运行时上界，才能保证发现所有程序错误。CBMC 还可以选择检查展开是否充分。

例如，考虑清单 5 中的程序 binsearch.c。如果对这个函数运行 CBMC，你会发现展开不会自行停止。内置简化器无法确定此循环的运行时上界。必须通过命令行参数指定展开界限：

```
cbmc binsearch.c --function binsearch --unwind 6 --bounds-check --
unwinding-assertions
```

C4

1 int binsearch(int x) 2 { 3 int a[16]; 4 signed low = 0, high = 16; 5 6 while(low <high) 7 { 8 signed middle = low + ((high -low) >> 1); 9 10 if(a[middle] <x) 11 high =middle; 12 else if(a[middle] >x) 13 low =middle + 1; 14 else // a[middle]==x 15 return middle; 16 } 17 18 return -1; 19 } 清单 5：源代码（binsearch

.c)

1 ** 结果：2 binsearch.c 函数 binsearch3 [binsearch.unwind.0]第 6 行 展开断言 循环 0：成功4 [binsearch.array_bounds.1] 第 10 行5 数组 ‘a’ 下界 ina[(signed long int)middle]：成功6 [binsearch.array_bounds. 2] 第 10 行7 数组 ‘a’ 上界 ina[(signed long int)middle]：成功8 [binsearch.array_bounds.3] 第 12 行9 数组 ‘a’ 下界 ina[(signed long int)middle]：成功10 [binsearch.array_bounds.4]第 12 行11 数组 ‘a’ 上界 ina[(signed long int)middle]：成功12 ...13 验证成功

清单 6：CBMC 对 binsearch.c（清单 5）执行命令 C4 的输出

结果输出如清单 6 所示。CBMC 验证数组访问均在边界范围内；请注意，这实际上取决于程序第 8 行右移的结果。此外，由于为 CBMC 指定了选项 --unwinding-assertions，它还会检查是否进行了充分展开，即证明运行时上界。

对于任何较低的展开界限，都存在能够证明还可以进行更多循环迭代次数的执行轨迹。因此，CBMC 将报告展开断言失败。通常，可以使用选项 --trace 获取记录这一点的反例轨迹。

```
1 _Bool nondet_bool();2 unsigned int nondet_unsigned_int();3 _Bool 锁 = 0;45 _Bool 锁()6 {7 if(nondet_bool())8 {9 断言(!锁);10 锁 = 1;11 return 1;12 }1314 return 0;15 }1617 void解锁()18 {19 断言(锁);20 锁 = 0;21 }

22 int main()23 {24 unsignedgot_lock = 0;25 unsignedtimes = nondet_unsigned_int();2627 while(times > 0)28 {29 if(lock())30 {31 got_lock++;32 /* 临界区 */33 }3435 if(got_lock != 0)36 unlock();3738 got_lock--;39 times--;40 }41 }
```

清单 7：源代码（lock.c）

CBMC 也可用于包含无界循环的程序。在这种情况下，CBMC 仅用于错误查找；它不会尝试找出所有程序错误。清单 7 中的程序 lock.c 是一个具有用户指定属性的程序示例。主函数中的 while 循环没有（有用的）运行时上界。因此，必须为 CBMC 执行的展开次数设置上界。设置方法有两种：

1. 可以使用 --unwind 命令行参数限制循环展开的次数。2. 可以使用 --depth 命令行参数限制要处理的程序步骤数。

对于清单 7 中的示例，当循环展开界限为 1 时，没有发现程序错误。但当界限为 2 时，CBMC 会检测到一条违反断言的执行轨迹。如果不使用展开断言，或者使用 --depth 命令行开关，CBMC 不一定能证明程序正确，但这有助于发现程序错误。有关限制循环展开的更多信息，请访问 https://www.cprover.org/cprover-manual/cbmc/unwinding/。

3.2 使用内置检查

缓冲区溢出问题已受到广泛关注。缓冲区是连续分配的一块内存，在.....中由数组或指针表示

```
1 #include <stdio.h>23 int main (int argc, char** argv)4 {5
char密码[8] = {'s','e','c','r','e','t','!','\0'};6 char缓冲区[16] = {'\0', };
7 int tmp;8 int index = 0;910 printf("请输入␣你的␣姓名:␣");11
while ((tmp = getchar()) !='\n')12 {13 缓冲区[index] = tmp;
14 ++index;15 }1617 printf("%s\n",缓冲区);1819 return
0;20 }
```

清单 8：源代码（login.c）

C语言。用 C语言编写的程序不会自动对缓冲区执行边界检查，这意味着程序可能会意外或故意地写入缓冲区范围之外。清单 8 中的示例程序是语法有效的 C语言程序，可以编译并运行（表面上看）而不会出现任何错误。如果在栈向下增长的系统（例如 x86）上编译，则可以观察到以下现象：

```
1 >gcc login.c -o login2 >./login3 请输
入您的姓名：Daniel4 Daniel5 >./l
ogin6 请输入您的姓名：SimSala Bim ...
7 Sim Sala Bim ...secret!
```

发生了什么？在 C语言中，字符串的结尾由字符 '\0' 标记。当我们输入超过 15 个字符时，缓冲区 末尾就不会有 '\0' 字符， printf 并会继续打印超出为缓冲区 分配的内存范围的字符，直到遇到 '\0' 字符，或因内存访问违规（段错误）而崩溃。根据内存布局，这可能会导致机密数据泄露。在上面的例子中，密码被打印到了终端。

我们能否借助 CBMC 发现这个问题？可以，CBMC 能够检查是否可能访问已分配对象边界之外的内存。当我们运行

CBMC 报告

```
** 结果：
login.c 函数 main
[main.array_bounds.2]第13行
数组 ‘buffer’ 上界inbuffer[(signed long int)index]:
失败
[main.array_bounds.1]
第13行
数组 ‘buffer’ 下界inbuffer[(signed long int)index]:
成功
```

这意味着程序确实存在错误。如清单 9 所示，检查执行轨迹（--trace）可以确认，问题出现在 index 增加到 16 后，我们对 buffer 赋值时。

```
...
状态 251 文件 login.c函数 main 第13行 线程 0
----------
buffer[15l]=-1 (11111111)
状态 252 文件 login.c函数 main 第14行 线程 0
----------
index=16 (0000000000000000 00000000 00010000)
...
状态 259 文件 <内置库-getchar> 函数 getchar 第15行 线程 0
----------
输入 getchar：-1 (11111111 11111111 11111111 11111111)
...
违反的性质：
文件 login.c 函数 main 第13行 线程 0
数组 ‘buffer’ 上界，位于 buffer[(signed long int)index]
!((signed long int)index >= 16l)
```

清单 9：CBMC 对 login.c 的命令 C5 输出（带 --trace）（清单 8）

如果继续输入字符，就会写入 缓冲区 之外，从而覆盖栈上的数据。特别是，这类程序错误可能被利用来覆盖函数的返回地址，从而使任意代码得以执行。

CBMC 能够通过检查这些下界和上界来检测此类程序错误，即使对于大小动态变化的数组也是如此：选项 --bounds-check 和 --pointer-check 会指示 CBMC 查找与指针和数组边界相关的错误。使用 --show-properties 调用时，CBMC 会打印它检查的性质列表：

```
cbmc login.c --show-properties --bounds-check --pointer-check
```
C6

请注意，其中列出了一个标记为“数组‘buffer’上界”的性质，以及错误数组访问的位置：

```
性质 main.array_bounds.1:
```

```
2 文件 login.c 第 13 行 函数 main3 数组‘buffer’下界 位于 buffer[(有符号
长整型)索引]4 (有符号长整型)索引 >= 0l5


6 性质 main.array_bounds.2: 7 文件 login.c 第 13 行，函数 main8 数
组‘buffer’在buffer[（有符号长整型）索引]处的上界9 !((有符号长整型)索
引 >= 16l)
```

如您所见，CBMC 在很大程度上会自行确定需要检查的性质。这是通过初步静态分析实现的，该分析依赖于在各种抽象域上计算不动点。这些自动生成的性质不一定对应程序错误——它们只是潜在缺陷，因为抽象解释可能并不精确。这些性质是否成立，或是否对应实际的程序错误，需要通过进一步分析来确定。

CBMC 使用符号模拟执行此分析，而将程序转换为公式为符号模拟提供了便利。随后，该公式会与性质结合。让我们看看 CBMC 的符号模拟生成的公式：

```
cbmc login.c --unwind 20 --show-vcc --bounds-check --pointer-check
```
C7

使用此选项时，CBMC 会执行符号模拟，并将验证条件打印为方程合取。判定过程必须证明验证条件有效，才能断言相应性质成立。让我们运行判定过程：

```
cbmc login.c --unwind 20 --bounds-check --pointer-check
```
C8

CBMC 将该方程（可使用 --show-vcc 打印）转换为合取范式，并将此公式传递给 SAT 求解器（参见 [15] 有关此类变换的背景介绍）。随后，它会确定为程序生成的哪些性质成立，哪些不成立。借助 SAT 求解器，CBMC 检测到 缓冲区的对象边界性质不成立，并会显示：

```
1 [main.array_bounds.2] 第 13 行2 缓冲区中数组 ‘buffer’ 的上界[
(signed long int)索引]:3 失败
```

为帮助理解问题，CBMC 可以为未通过的性质生成反例轨迹。要获取清单 9 的这条执行轨迹，请运行：

```
cbmc login.c --unwind 20 --bounds-check --pointer-check --trace
```
C9

随后，CBMC 会打印一条反例轨迹，也就是一条以 main 开始、并以违反该性质的状态结束的程序轨迹。在我们的示例中，程序轨迹以错误的数组访问结束。它还会给出导致程序错误发生时输入变量必须具有的值。在此示例中，`getchar`（重复调用）的返回值必须为 1，才能触发越界数组访问。如果在示例中添加一个分支，要求输入不超过 15 个字符，那么程序错误就会得到修复，CBMC 也会报告所有性质均已成功证明。

3.3 内置函数和类型

CPROVER 框架（其中包含 CBMC）提供了内置函数和类型，用于访问验证器的内部功能；借助这些函数和类型，可以实现源语言本身不提供的功能。

除了 assert.h 提供的 `assert(condition)` 函数外，还有一个 `__CPROVER_assert(condition, "description")`，它允许为性质附加自定义描述。

函数 `__CPROVER_assume(condition)` 将一个表达式作为约束添加到程序中。如果该表达式在某条路径上计算结果为 false，则该程序路径的执行会中止，但不会失败。假设用于限制程序所做的非确定性选择。例如，假设我们希望对返回 1 到 100 之间数字的非确定性选择进行建模。不存在取值范围为此区间的整数类型。因此，我们使用 `__CPROVER_assume` 来限制非确定性选择的整数范围：

```
1 unsigned intnondet_uint();23 unsigned
intone_to_hundred()4 {5 unsigned intresult=
nondet_uint();6 __CPROVER_assume(result>=1&&
result<=100);7 return result;8 }
```

此函数会返回 1 到 100 之间所需的整数。用户必须确保作为假设给出的条件确实能通过某种非确定性选择得到满足；否则，模型检测步骤将会空洞地通过。

还要注意，假设不会追溯生效。它们只会影响程序顺序中位于其后的断言（或其他性质）。下面的示例最能说明这一点。在以下这个变体程序中，假设对断言没有任何影响，这意味着断言将会失败：

```
1 unsigned intnondet_uint();23 unsigned
intone_to_hundred()4 {5 unsigned intresult=
nondet_uint();6 断言(result<100);7
__CPROVER_assume(result>=1&& result<=100);8
return result;9 }
```

假设确实会限制搜索空间，但只对其后的断言生效。例如，在这个程序中，将同一个断言放到假设之后（按程序顺序）后，程序就会通过：

```
1 unsigned int nondet_uint();23
unsigned int one_to_hundred()4
{
```

```
5 unsigned intresult=nondet_uint();6
__CPROVER_assume(result>=1&& result<=100);7
断言(result<100);8    return result;9 }
```

请注意，不能通过非确定性在假设中实现全称量化的效果。例如：

```
1 int main()2 {3 int a[10],x,y;45    x=nondet_int();6
y=nondet_int();7 __CPROVER_assume(x>=0 && x<10&&
y>=0 && y<10);89    __CPROVER_assume(a[x]>=0);1011
断言(a[y]>=0);12 }
```

第 11 行中的断言失败，因为x 和y 不一定具有相同的值。第 9 行只确保存在一个索引x ，使得 a[x]>=0。

内置类型。 __CPROVER_bitvector[size] 用于指定大小任意但固定的位向量。可以应用常见的整数类型修饰符 signed 和 unsigned。此类型的操作数将进行常见的算术提升。

__CPROVER_floatbv[total_size][mantissa_size] 用于指定大小任意但固定的 IEEE-754 浮点数。total_size 表示该数的总大小（以位为单位），而mantissa_size 表示尾数或有效数的大小（以位为单位，不包括隐藏位，因此单精度时应为 23）。可以通过给全局变量__CPROVER_rounding_mode赋值来设置 IEEE 浮点运算舍入模式。

__CPROVER_fixedbv[total_size][fraction_size] 指定一个大小任意但固定的定点位向量。total_size 是类型的总大小（以位为单位），而fraction_size是基数点后的位数。

并发。异步线程通过在指令前添加一个带有前缀 __CPROVER_ASYNC_的标签来创建。原子区段由__CPROVER_atomic_begin()和__CPROVER_atomic_end()界定。

完整的 CPROVER API 文档可在https://www.cprover.org/cprover-manual/api/找到。

```
#include <assert.h>
#include <pthread.h>
pthread_mutex_t mutex;
int 余额 = 1000;
void*事务(void*amount)
{
  // pthread_mutex_lock(&mutex);
  int current =余额;
  current +=*(int *)amount;
  余额 =current;
  // pthread_mutex_unlock(&mutex);
  return 0;
}
int main()
{
  pthread_t t1,t2;
  pthread_mutex_init(&mutex,0);
  int amount1 = 3000;
  pthread_create(&t1,0,transaction, &amount1);
  int amount2 = 9000;
  pthread_create(&t2,0,transaction, &amount2);
  pthread_join(t1, 0);
  pthread_join(t2, 0);
  断言(余额 == 6000);
  pthread_mutex_destroy(&mutex);
  return 0;
}
```

清单 10：源代码（account.c）

3.4 内置库

大多数 C 程序都会使用库提供的函数。例如，标准 ANSI-C 库中的函数，如 `malloc` 或 `printf`。验证使用此类函数的程序需要满足两个条件：

1. 必须提供适当的头文件。这些头文件包含所用函数的声明。2. 必须提供适当的定义。

大多数 C语言 编译器都附带 ANSI-C 库函数的头文件。CBMC 提供了常用函数的定义，例如内存分配或字符串操作。这些函数通常会对 C 标准规定的行为进行过度近似，以便进行可靠的验证。CBMC 提供的此类库函数的一个例子是 pthread 库的一个子集，以下示例将使用它。

清单 10 中的银行账户程序使用 `pthread` 库启动两个线程，以便在银行账户上执行交易。CBMC 支持 `pthread`库，因此只需运行 cbmc account.c，就可以对这个并发程序进行模型检验。CBMC 会很快报告：

```
** 结果：
...
account.c 函数 main
[main.断言.1]第 29 行断言 余额 == 6000：失败
...
验证失败
```

这个程序存在竞争条件。在多线程程序中，当计算结果取决于并发线程指令的交错执行顺序时，就可能发生竞争条件。通过检查执行轨迹，我们甚至可以看出竞争条件为何会发生：

```
1 ...2 状态 108 文件 account.c函数 transaction 第 12 行 线程 13 ----------------------------------------------------
4 余额=-2000 (11111111 1111111111111000 00110000)5 ...6 状态 137 文
件 account.c函数 transaction 第 12 行 线程 27 ----------------------------------------------------
8   余额=9000(00000000 00000000 00100011 00101000)9 ...
```

线程 1 取出 3000 并将余额设为 -1000，但随后线程 2 用初始余额加上 10000 的结果覆盖了余额，最终得到 9000，而不是预期的 6000。

显然，只需将余额更新设为原子操作即可修复该程序。我们可以取消清单 10 第 9 行和第 13 行中的锁的注释，并使用 CBMC 验证程序现在是否能正确运行。

3.5 测试输入

CBMC 可用于自动生成满足特定代码覆盖标准的测试输入。常见的覆盖标准包括分支覆盖、条件覆盖和修正条件/判定覆盖（MC/DC）。其中，MC/DC 是若干航空电子软件开发指南的要求，用于确保对安全关键软件进行充分测试。简而言之，为满足 MC/DC，对于每个包含布尔判定的条件语句，每个布尔变量都应分别取值为“真”和“假”一次，并且这种取值应影响判定结果。

接下来，我们将演示如何在 CBMC 中使用测试套件生成功能。清单 11 中的程序 pid.c 摘录自实时嵌入式基准测试 PapaBench [19]，实现了无人驾驶飞行器（UAV）电传飞行自动驾驶仪的部分功能。我们为适应本文的目的对其略作调整。

函数 `climb_pid_run` 旨在控制无人驾驶飞行器的垂直爬升。它由 `main` 函数中的反应式循环调用。图 1 绘出了这个简单控制器在目标速度为每秒 0.5 米时的行为。

`main` 函数已扩展，用于模拟每个时间步获取的输入。函数 `__CPROVER_input` 和 `__CPROVER_output` 用于报告输入值或输出值。请注意，它们不会生成输入值或输出值，只会报告其值。第一个参数是字符串常量，用于区分多个输入和

28 void climb_pid_run() 29 { 30 floaterr=
estimator_z_dot-desired_climb; 31 32 float fgaz=
CLIMB_PGAIN*(err+CLIMB_IGAIN*climb_sum_err)+33
CLIMB_LEVEL_GAZ+CLIMB_GAZ_OF_CLIMB*desired_climb; 34
35 float pprz=fgaz*MAX_PPRZ; 36 desired_gaz=((pprz>=0 && pprz<=
MAX_PPRZ) ? pprz : (pprz>MAX_PPRZ ? MAX_PPRZ : 0)); 37 38 /**
爬升的俯仰偏移 */ 39 float pitch_of_vz=(desired_climb>0) ?
desired_climb*CLIMB_PITCH_OF_VZ_PGAIN : 0; 40 desired_pitch=
NAV_PITCH+pitch_of_vz; 41 42 climb_sum_err=err+
climb_sum_err; 43 if (climb_sum_err>MAX_CLIMB_SUM_ERR)
climb_sum_err=MAX_CLIMB_SUM_ERR; 44 if (climb_sum_err<-
MAX_CLIMB_SUM_ERR) climb_sum_err=-
MAX_CLIMB_SUM_ERR; 45 } 46 47 int main() 48 { 49 while(1) 50 {
51 /** 非确定性输入值 */ 52 desired_climb=nondet_float(); 53
estimator_z_dot=nondet_float(); 54 55 /** 输入值范围 */ 56
__CPROVER_assume(desired_climb>=-MAX_CLIMB &&
desired_climb<=MAX_CLIMB); 57
__CPROVER_assume(estimator_z_dot>=-MAX_CLIMB &&
estimator_z_dot<=MAX_CLIMB); 58 59 __CPROVER_input("
desired_climb", desired_climb); 60 __CPROVER_input("
estimator_z_dot", estimator_z_dot); 61 62 climb_pid_run(); 63 64
__CPROVER_output("desired_gaz", desired_gaz); 65
__CPROVER_output("desired_pitch", desired_pitch); 66 } 67
return 0; 68 } 清单 11：部分源代码（pid.c）

输出（输入通常使用非确定性生成，如此处所述）。字符串常量后面跟着任意数量的任意类型值。清单 12 展示了使用以下 CBMC 调用计算测试套件的格式化版本：

```
cbmc pid.c --cover mcdc --unwind 6
```

C10

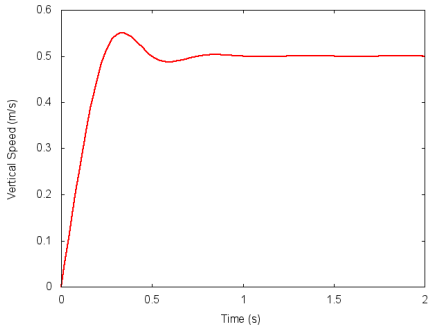

图 1：PID 控制器的行为

```
1 测试套件：2 测试 1。3 （迭代 1）desired_climb=-1.000000f,
estimator_z_dot=1.000000f45 测试 2。6 （迭代 1）desired_climb=-1.000000f,
estimator_z_dot=1.000000f7 （迭代 2）desired_climb=1.000000f,
estimator_z_dot=-1.000000f8

9 测试 3。
10（第 1 次迭代）期望爬升=0.000000f，估计器_z_dot=-1.000000f 11（第 2 次迭
代）期望爬升=1.000000f，估计器_z_dot=-1.000000f 12 13 测试 4。 14（第 1 次
迭代）期望爬升=1.000000f，估计器_z_dot=-1.000000f 15（第 2 次迭代）期望爬
升=1.000000f，估计器_z_dot=-1.000000f 16（第 3 次迭代）期望爬升=1.000000f，
估计器_z_dot=-1.000000f 17（第 4 次迭代）期望爬升=1.000000f，估计器_z_dot=-
1.000000f 18（第 5 次迭代）期望爬升=0.000000f，估计器_z_dot=-1.000000f 19
（第 6 次迭代）期望爬升=1.000000f，估计器_z_dot=-1.000000f 20 21 测试 5。
22（第 1 次迭代）期望爬升=-1.000000f，估计器_z_dot=1.000000f 23（第 2 次迭
代）期望爬升=-1.000000f，估计器_z_dot=1.000000f 24（第 3 次迭代）期望爬升=-
1.000000f，估计器_z_dot=1.000000f 25（第 4 次迭代）期望爬升=-1.000000f，估
计器_z_dot=1.000000f 26（第 5 次迭代）期望爬升=-1.000000f，估计器_z_dot=1.
000000f 27（第 6 次迭代）期望爬升=-1.000000f，估计器_z_dot=1.000000f
```

清单 12：使用命令 C10 为 pid.c（列表 11）生成的测试套件

这里展示了一个测试套件，其 MC/DC 覆盖率接近 100%（显然，`return` 语句未覆盖`main`）。每项测试所需的测试输入都按时间步列出。

除了 MC/DC，CBMC 还支持多种其他覆盖准则，例如分支覆盖和条件覆盖。此外，可以使用 `__CPROVER_cover(condition);` 语句定义自定义覆盖准则。

4 验证真实世界软件

现有软件项目通常并非只有一个可以直接交给模型检查器的源文件，而是包含大量分布在不同目录中的源文件，并引用外部库和系统范围的选项。随后，构建系统会从系统中收集配置选项，并按照构建规则编译软件。

使用软件验证工具检查这类项目时，如果有一个编译器先将所有必要的模型收集到单个模型文件中，整个过程就会大为简化。goto-cc 就是这样一种模型文件提取器。它利用编译器（例如 GCC）的预处理器将文本转换为实际的 C 代码。预处理的结果随后会交给内部 C 解析器（该解析器作为 CBMC 工具的一部分构建并发展，至今已有十多年）。此解析器支持多种 C 方言，包括 GCC 的扩展、Visual Studio、CodeWarrior 和 ARM-CC。除了 C 方言之外，goto-cc 还必须（也确实会）解析这些工具的所有相关命令行选项，因为它们可能会影响程序的语义。goto-cc 构建一种称为“GOTO 程序”的中间表示——一种类似控制流图的表示形式，而不是可执行二进制文件。

构建系统有时会先生成可执行文件，供构建过程中使用，或者调用检查目标文件的链接器。goto-cc 还可以构建混合二进制文件，其中既包含可执行代码，也包含用于验证的模型。要启用此模式，请创建一个指向 goto-cc 二进制文件、名称为 goto-gcc 的链接。在此模式下，首先会调用原始编译器或链接器，生成一个 ELF 格式的目标文件或可执行文件（例如，其中包含 x86/64 位指令）。接下来，goto-cc 会作为编译器或链接器被调用，并使用传递给原始编译器或链接器的相同命令行选项。如上所述，在编译时，此步骤会生成编译单元的中间表示。为了适应任意构建系统，生成的中间表示会作为新的节添加到 ELF 目标文件或可执行文件中。因此，使用 goto-cc 进行链接时，它会从各个输入文件中读取额外的节，执行链接，然后将链接结果添加到原始链接器生成的输出文件中。goto-cc 还支持一种适用于 OS X 的等效方法，OS X 使用不同的目标文件格式。在这种情况下，会构建所谓的胖二进制文件，以模拟上述行为。有关在此过程中支持链接器脚本的扩展，将在 [12] 中讨论。

请注意，将中间表示添加到原始目标文件中是一个关键步骤。这样可以确保该文件仍可执行，或仍可供原始链接器使用；重命名或构建归档文件等操作都会同时应用于未经修改的编译结果和中间表示，无需额外工作。这使得能够在

各种 Linux 发行版 [16, 17, 11] 的软件包上运行 CBMC。另一种做法是跟踪所有这些步骤，例如替换系统库，ECLAIR[1] 就采用了这种方式。

某些软件项目自带库。此外，目标也可能是单独分析某个库。为此，可以使用 goto-cc 将多个模型文件链接成模型文件库。随后，可以将目标文件与该模型库链接。为此，goto-cc 还提供了链接器模式。

要启用此链接器模式，请创建一个指向 goto-cc 二进制文件的链接，并将其命名为 goto-ld（Linux 和 Mac），或者将 goto-cc 二进制文件复制为 goto-link.exe（Windows）。现在，可以使用 goto-ld 工具，无缝替代大多数 Unix（及类 Unix）系统中的 ldtool，以及 Windows 上的 link 工具。

更多信息请参见https://www.cprover.org/cprover-manual/ goto-cc。

5 架构

CBMC 等有界模型检查器会将有关程序路径的问题转换为约束，供现成的 SAT 或 SMT 求解器求解。使用 SAT 后端时，给定一个带有断言注释的程序，CBMC 会输出一个合取范式公式，其解描述了导致断言违例的程序路径。为此，CBMC 会执行以下主要步骤，这些步骤在图 2 中概述，并将在下文中解释。

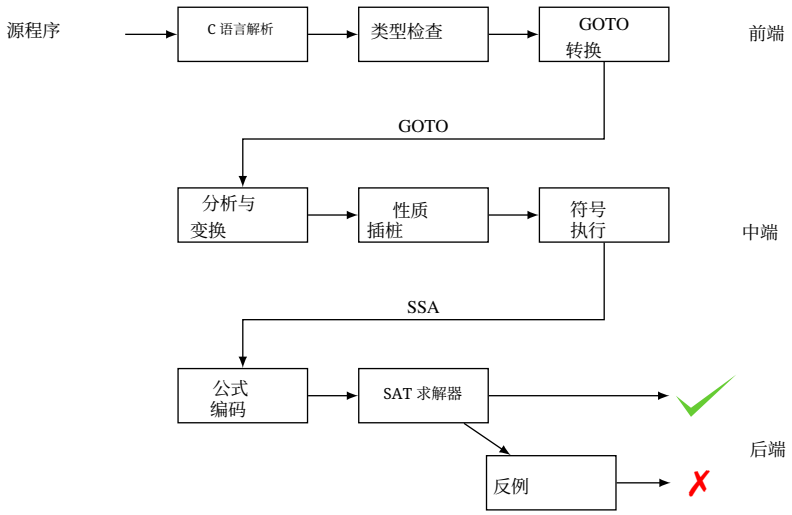

图 2：CBMC 概览

[1] https://bugseng.com/products/eclair/discover

5.1 命令行前端

命令行前端会处理用户提供的选项，并据此配置 CBMC。调用 CBMC 的通用命令语法为

```
cbmc [options ...] [file.c ...]
```
C11

其中 [options ...] 的说明见下文，而 [file.c ...] 表示零个或多个源文件名。用户通常提供的参数包括循环展开限制，或假定的系统架构，即基本数据类型的位宽和字节序，以及特定于操作系统的配置参数。如果用户选择不覆盖平台配置，则系统架构配置默认为 CBMC 编译时所用平台的规约。下面将介绍最常用的命令行选项；控制 CBMC 架构特定部分的其他选项将在后续章节中讨论。

- 常规选项：

  – --help、-h、-?: 显示版权信息和命令行选项列表。– --version:显示当前版本。

- 平台配置：

  – --function *f*：使用函数 *f* 作为程序入口点，而不是“main”。 – -I *path*：将 *path* 添加到 C 预处理器展开 #include指令时使用的搜索路径中。此选项可以多次指定，-D *macro* 也是如此： – -D *macro*：定义 C预处理器宏 *macro*，其中 *macro* 要么仅为宏名称，要么采用 *name=value*的形式。 – --16、--32、--64：分别将 C语言类型 `int` 的位宽设为 16、32 或 64 位。 – --LP64、--ILP64、--LLP64、--ILP32、--LP32：根据 [23]中的定义，设置 int、long 和指针的位宽。 – --little-endian、--big-endian：设置字与字节之间转换时使用的字节序。 – --unsigned-char：使 `char` 类型为无符号。 – --ppc-macos、--i386-macos、--i386-linux、--i386-win32、--win32 和 --winx64：根据给定的处理器和操作系统组合，设置特定于平台的定义、位宽和字节序。

• 控制用户界面的选项：

  – --json-ui、--xml-ui：分别将 CBMC 的输出更改为 JSON 或 XML 格式。--json-interface、--xml-interface 则扩展此功能，分别通过 JSON 或 XML 接收选项。此类输出（以及输入）更适合机器处理。

– --verbosity n：设置运行 CBMC 时打印的状态信息量。$n = 0$ 会禁用除测试用例以外的所有输出，$n \geq 1$ 会启用错误消息，$n \geq 2$ 会添加警告，$n \geq 6$ 会启用进度信息（$n = 6$ 为默认值），$n \geq 8$ 会添加统计信息，$n \geq 9$ 则会输出调试信息。

5.2 语言前端

CBMC 使用“GOTO 程序”作为中间表示。在这种语言中，所有非线性控制流，例如 if/switch 语句、循环和跳转，都会被转换为等价的受保护的 *goto* 语句。这些语句是带有可选保护条件的 goto，因此这些受保护的 goto 也足以表示 if/else 分支。在这一中间层中，最重要的语句类别包括赋值、goto 语句、函数调用、声明、断言和假设。

对于 C/C++输入源代码，CBMC 首先调用 C 预处理器（例如，在 Microsoft Windows 系统上使用 cl，或在类 Unix 系统上使用 gcc -E；如第 4 节所述，也支持其他编译器工具链），然后将结果传递给 CBMC 的 C 或 C++解析器，从而得到这种中间表示。调用 CBMC 时使用 --preprocess 可以查看预处理器的输出。CBMC 使用自己的 C 和 C++ 解析器，而不依赖现有工具链，因此能够支持多种 C/C++ 方言，包括各种 GCC 扩展。C 或 C++ 解析器会生成一棵带有源文件和行信息的解析树。调用 CBMC 时使用 --show-parse-tree，可以查看带有类型信息注释的解析树。其他语言前端（例如 Java 前端 [13], ）的解析工作流程有所不同，但最终也会进行类型检查：

类型检查会遍历解析树，收集所有类型名称和符号标识符，并为找到的每个符号和表达式分配位级类型信息，从而填充符号表。若要查看符号表，请使用选项 --show-symbol-table 调用 CBMC。如果类型检查检测到任何不一致，CBMC 就会中止。作为实验，比较 cbmc --16 abs.c 与 cbmc --32 abs.c 的输出，可以看出类型检查如何影响位级类型，进而影响常量扩展为不同位向量的方式：

类型检查成功后，CBMC 会为解析树中找到的每个过程或方法生成一个 GOTO 程序。此外，它还会添加一个新的主函数，先调用初始化函数，为具有静态生存期的对象进行初始化，然后调用原始程序入口函数。

对于前文清单 1 所示的 abs.c 示例，运行

```
cbmc --functionabs --signed-overflow-check --show-goto-functions C12
```

生成的输出如清单 13 所示。在此输出中，每个 GOTO 程序（即每个过程）都以其名称开头（第 1、19 和 26 行），并以 END_FUNCTION GOTO 程序指令结尾（第 17、24 和 46 行）。每条指令都包含其源代码位置（第 2、4 行等）。指令本身会被打印出来

```
常量 * 类型：signedbv * 位宽：16 * #c_type：有符号整数 * 值：0000000000000000 * #source_location：* 文件：abs.c * 行：4 * 函数：绝对值 * 工作目录：/home * #base：10
```

```
常量 * 类型：有符号位向量 * 位宽：32 * #C语言类型：有符号整数 * 值：00000000000000000000000000000000 * #源代码位置：* 文件：abs.c * 行：4 * 函数：绝对值 * 工作目录：/home * #基数：10
```

```
1 绝对值
2 // 0 文件 abs.c 行 2函数abs 3 有符号整数 y;
4 // 1 文件 abs.c 行 2函数abs 5 y = x; 6 // 2 文件 abs.c 行 4函数abs 7 IF !(x < 0) THEN GOTO 1 8 // 3 文件 abs.c 行 5 函数 abs 9 断言 !(x ==-2147483648) 10 // 4 文件 abs.c 行 5函数abs 11 y = -x; 12 // 5 文件 abs.c 行 8函数abs 13 1: abs#return_value =y; 14 // 6 文件 abs.c 行 8函数abs 15 dead y; 16 // 7 文件 abs.c 行 9函数abs 17 END_FUNCTION 18 19 __CPROVER_initialize 20 // 8 文件 <built-in-additions>行 20 21 // 标签：__CPROVER_HIDE 22 __CPROVER_rounding_mode = 0; 23 // 9 无位置 24 END_FUNCTION
```

```
25 __CPROVER__start 26 // 10 无位置信息 27 __CPROVER_initialize(); 28 // 11 文件 abs.c 第 1 行 29 有符号 intx; 30 // 12 文件 abs.c 第 1 行 31 x = NONDET(有符号 int); 32 // 13 文件 abs.c 第 1 行 33 输入("x", x); 34 // 14 文件 abs.c 第 1 行 35 绝对值(x); 36 // 15 文件 abs.c 第 1 行 37 return′=abs#return_value; 38 // 16 文件 abs.c 第 1 行 39 dead abs#return_value; 40 // 17 文件 abs.c 第 1 行 41 OUTPUT("return", return′); 42 // 18 无位置信息 43 dead x; 44 // 19 无位置信息 45 END_FUNCTION 46
```

清单 13：由 C12 命令为 abs.c（清单 1）生成的 GOTO 程序

采用类 C 语法。例如，第 3 行和第 30 行表示变量声明，这些变量可能会离开作用域——每个这样的点都由相应的死指令表示，如第 15 行和第 44 行所示。第 5 行、第 13 行等处的指令表示赋值。此外，第 13 行带有一个标签：它是第 7 行（条件）`goto` 语句的分支目标。这个条件式`goto` 语句对清单 1 第 4 行 `if`语句所产生的控制流进行了编码。

分析与变换。 第13行的情况比较特殊，因为它是返回指令移除的结果：为了尽可能减少分析需要考虑的指令类别，我们将其转换为

通过 `return` 语句的效果模拟为 `goto` 以及对全局变量和线程局部变量的赋值。其他变换虽然不适用于此示例，但还包括替换函数指针。首先，通过轻量级静态分析解析函数指针，该分析会检查已声明函数的形式参数与通过函数指针调用时的实际参数之间的类型兼容性。所有匹配的目标都会合并为条件调用列表；当函数指针的实际值与目标函数的地址匹配时，就会执行相应分支。由此，我们得到一个静态调用图。

属性插桩。 在第 9 行，我们可以看到启用 --signed-overflow-check 后生成的断言。该断言用于防止第 11 行中的取反操作可能导致的未定义行为。正如第 3 节所述，这类断言是通过轻量级且偏向过近似的数据流分析生成的。

控制此插桩过程的选项包括：

• --bounds-check：确保每次数组索引访问都在其边界范围内。
• --pointer-check：确保仅使用指向存活对象的指针进行解引用。
• --div-by-zero-check：确保除数不为零。
• --signed-overflow-check, --unsigned-overflow-check：分别检查有符号整数或无符号整数是否不存在算术上溢和下溢。

• --conversion-check：确保类型转换不会应用于无法用目标类型表示的值。

• --undefined-shift-check：确保移位量不超过被移位对象类型的位宽。

• --float-overflow-check、--nan-check：分别确保浮点运算不会导致正无穷大或负无穷大，以及非数。
• --enum-range-check：确保枚举类型对象的取值始终是该类型已声明的枚举常量。

使用 *GOTO* 程序。 如上所述，GOTO 程序可以通过自定义二进制格式进行序列化和反序列化。这种方法支持使用 goto-cc，如第 4 节所述。其他可用于 GOTO 程序的工具还包括变换工具 goto-instrument，以及 goto-diff；后者为 GOTO 程序实现了与基于文本的 Unix 工具 diff 等效的功能。

5.3 符号执行

用于控制程序插桩和循环展开的选项：

• --no-library：默认情况下，CBMC 会附带系统库函数的抽象版本。此选项会禁用此类代码的包含。

• --show-goto-functions：显示插桩后得到的 GOTO 函数，如第 5.2 节所述。此选项主要用于调试。

• --no-assumptions：程序员可以使用内置函数 `__CPROVER_assume(`x`)` 向受检程序添加假设。也就是说，对于 CBMC 考虑的所有路径，在插入该内置函数的程序点处，性质 x 必须求值为 true true。如果设置了 --no-assumptions，则会忽略假设。

• --function f：使用函数 f 作为程序入口点，而不是“`main`”。
• --depth k：符号执行程序时，沿任意路径最多执行 k 步。除非 k 步足以到达所有状态，否则这会导致验证不可靠。

• --unwind k、--unwindset L:k,...、--show-loop-ids：循环展开、递归和向后跳转最多 k 次。使用 --unwindset L:k,... 时，展开界限 k 仅对 ID 为 L 的循环设置；使用 --show-loop-ids 可以找到该 ID，该选项会列出所有循环及其标识符。使用 --unwind 或 --unwindset 时，即使验证成功，也是不可靠的，除非指定的界限足以完成循环展开。要确保验证可靠（但可能不完整），请添加 --unwinding-assertions：
• --unwinding-assertions、--partial-loops：每当使用 --unwind 或 --unwindset 指定的循环展开界限达到时，CBMC 都会插入一个假设，认为循环条件确实不再成立，即循环确实会退出。这可能会排除可行的执行路径，从而导致验证不可靠，正如上文所述。为避免这种不可靠性的来源，可以指定 --unwinding-assertions，使其插入断言而不是假设。该断言会检查循环退出条件是否确实始终成立，也就是说，循环展开步数是否足够。使用参数 --partial-loops 时，循环展开期间既不会生成假设，也不会生成断言。这对于实验可能很有用，但确实会导致验证的程序可能与原程序大不相同，因为每个循环都会被替换为固定次数的循环体条件重复执行，且不会检查循环条件在末尾是否计算为 false。这可能会生成在原程序中不可能出现的执行轨迹。

由于 CBMC 实现的是一种有界模型检测变体，因此必须特别关注循环。与 [6], 中介绍的原始有界模型检测算法不同，CBMC 目前不会随着有界模型检测的进行而增加路径的最大长度，因此它并不完备。CBMC 会将循环积极展开至固定上限，该上限可以由用户针对每个循环单独指定，也可以对所有循环全局指定。在展开过程中，CBMC 还会将 GOTO 函数转换为静态单赋值（SSA）形式 [2, 22, 14]。这一过程结束后，程序便表示为一个关于重命名后的程序变量的数学方程，其中包含带守卫的语句。守卫决定了在给定一次具体的程序执行时，是否实际进行赋值。

CBMC 的基本思路是对程序执行进行建模，直到达到有界步数。从技术上讲，这通过一个本质上相当于展开循环的过程来实现。要理解这一概念，最好的方式是看一个通用示例：

```
int main(int argc,char **argv) {
while(cond) {      循环体代码    }  }
```

一个最多检查循环五次迭代、用于发现程序错误的有界模型检验实例，将包含五份循环体副本，本质上相当于检查以下无循环程序：

```
int main(int argc, char**argv)
{   if(cond) {  循环体代码副本 1
if(cond) {  循环体代码副本 2
if(cond) {  循环体代码副本 3
if(cond) {  循环体代码副本 4
if(cond) {  循环体代码副本 5
}            }          }       }     }  }
```

请注意，这里使用 if 语句来防止在循环执行次数不足五次时执行循环体。上述构造旨在生成一个程序，使其对于包含最多五次循环迭代的执行轨迹，与原程序的执行轨迹等价。

在许多情况下，CBMC 能够自动确定循环迭代次数的上界。即使循环展开次数不是常数，这种方法也可能有效。请看下面的示例：

```
_Boolf();  int main()
{  for(int i=0; i<100; i++)    {
if(f()) break;    }    断言(
0);  }
```

上面的程序中的循环有明显的迭代次数上界，但请注意，循环可能会根据

f() 的返回值提前终止。尽管如此，CBMC 仍能自动将循环展开至完成。

如果循环迭代次数高度依赖于数据，自动检测展开界限可能会失败。此外，任何给定循环执行的迭代次数都可能过大，甚至可能没有上界。针对这种情况，CBMC 提供了命令行选项 --unwind B，其中 B 表示一个数值，对应于 CBMC 对任意循环执行的最大循环展开次数。

请注意，展开次数是通过统计回跳次数来衡量的。在上面的示例中，请注意，循环终止前，条件 `i<100` 实际上被求值了 101 次。因此，循环的限制应为 101，而不是 100。

在 [1] 中，我们介绍了一种扩展，用于对并发程序进行高效的有界模型检测。该扩展以符号方式编码共享变量读写访问之间的偏序。

5.4 SAT/SMT 后端

通过对所有表达式以及布尔守卫进行位精确建模，可以将所得方程转换为合取范式公式（参见 [10]）。这里需要注意，CBMC 还支持将其他判定过程用作后端，例如 SMT（理论可满足性）求解器 [20, 21], ; 在这种情况下，会使用合取范式以外的编码方式。这些后端可以通过命令行选项进行选择，例如 --smt2（使用默认的 SMT2 求解器，目前为 Z3）、--z3（使用 Z3 [18], ）或 --cvc4（选择 CVC4 [5]）。

合取范式公式可通过 --dimacs 输出，并会传递给 SAT 求解器，由其尝试寻找满足赋值。此处，这样的赋值对应于一条违反了受检程序中至少一个断言的路径。反过来说，如果该公式不可满足，则在给定的展开界限下，任何断言都不会被违反。CBMC 支持程序中的多个性质，并会迭代查询求解器，以确定每个性质的结果。

如果找到满足赋值，有界模型检验器便确定了一个针对以断言形式给出的规约的反例。为了将 SAT 公式的模型转换为对 CBMC 用户有用的信息，它会被转换成赋值列表。CBMC 通过查询带守卫的语句方程来找到这一序列：在计算得到的模型下，守卫条件求值为真的每条语句都构成反例中的一次赋值。赋予的实际值也可以从 SAT 公式的模型中找到。由此得到的反例输出如清单 4 所示。

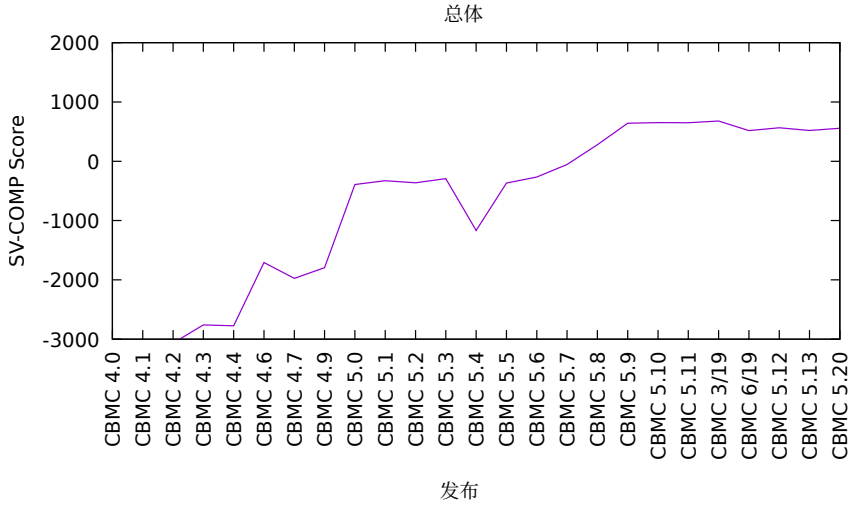

图 3：所有选定基准测试的得分时间线

6 性能历史

CBMC 已持续开发 15 年以上，但这些年来它真的变得更好了吗？为了回答这个问题，我们对 22 个 CBMC 版本进行了基准测试，版本范围从 4.0（2011 年 6 月）到 5.20（2020 年 12 月）。[2] 我们使用了 SV-COMP[3] 中的 9 个类别，并通过 BenchExec[4] 在一台运行 Ubuntu 16.04、配备 Intel Xeon Platinum 8175M 处理器（2.50 GHz）的机器上运行测试，资源限制为 15 GB 和 900 秒。

图 3 展示了 CBMC 在所有选定基准测试上的 SV-COMP 得分[5] 的变化。请注意，由于历年来规则和基准测试集均发生了变化，因此这些得分无法与官方 SV-COMP 结果进行比较。CBMC 凭借基于 4.5 的版本赢得了 SV-COMP 2014，并凭借基于 4.9 的版本在 SV-COMP 2015 中排名第三。

就 SV-COMP 分数而言，在这些版本跨度超过 9 年的时间里，CBMC 的表现有了显著提升。尤其是接近 CBMC 5.0 的版本，

[2] 旧版本的移植位于 https://github.com/ diffblue/cbmc 中的 cbmc-x.y-patch 分支。相应的 SV-COMP 包装脚本位于 https://github.com/diffblue/cprover-sv-comp 中的 cbmc-x.y 分支。[3] ReachSafety-Arrays、ReachSafety-BitVectors、ReachSafety-ControlFlow、ReachSafety- Floats、ReachSafety-Heap、ReachSafety-Loops、MemSafety-Arrays、MemSafety- Heap、MemSafety-LinkedLists；版本 https://github.com/sosy-lab/sv-benchmarks/tree/b8369a395d4749eb7eee1c3bd8149a3dc799e7f3[4] https://github.com/sosy-lab/benchexec/releases/tag/1.17[5] 在不安全基准测试中发现程序错误得 1 分；错误地声称其安全扣 32 分。在安全基准测试中证明其正确得 2 分；错误地报告程序错误扣 16 分。

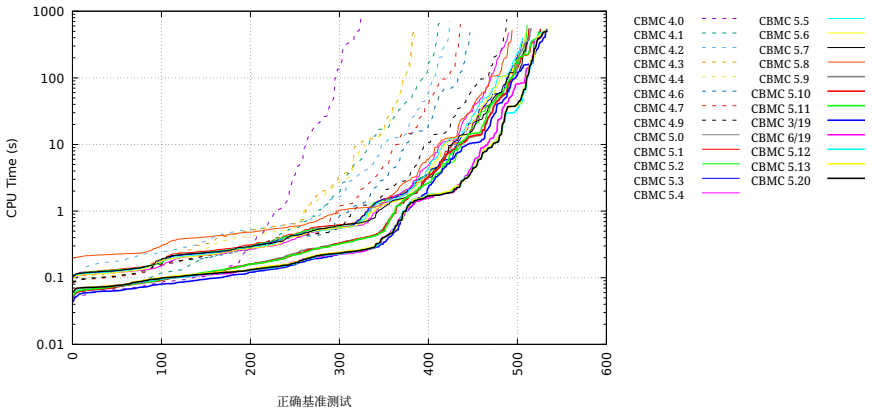

图 4：所有选定基准测试的 CPU 时间分位数图

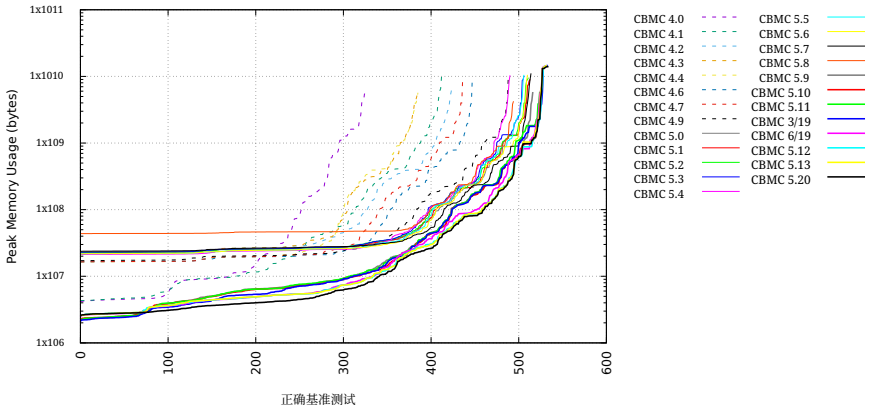

图 5：所有选定基准测试的峰值内存使用量分位数图

（2014年—2015年）与早期的 CBMC 4.x 版本（2011年—2013年）相比有了巨大改进。从 CBMC 5.7 到 5.9（2017年—2018年）可以看到第二轮改进。

从图 4 的 CPU 时间分位数图中，我们也可以观察到这些改进一直延续到 CBMC 5.0（4.4 到 4.6 以及 4.6 到 5.0 之间有显著提升），随后直到 CBMC 5.7 都基本停滞不前（且 5.4 出现了性能回退）。不过，从 CBMC 5.7 到 5.9 有了显著提速（但 5.8 出现了性能回退），而在最新的几个版本中，速度也进一步有所提升，不过幅度不太显著。从 CBMC 4.4 到 4.6、从 4.6 到 5.0 以及从 5.7 到 5.9 的显著改进，还大幅降低了内存消耗，如图 5 的峰值内存使用量分位数图所示。这些改进主要归功于增强的

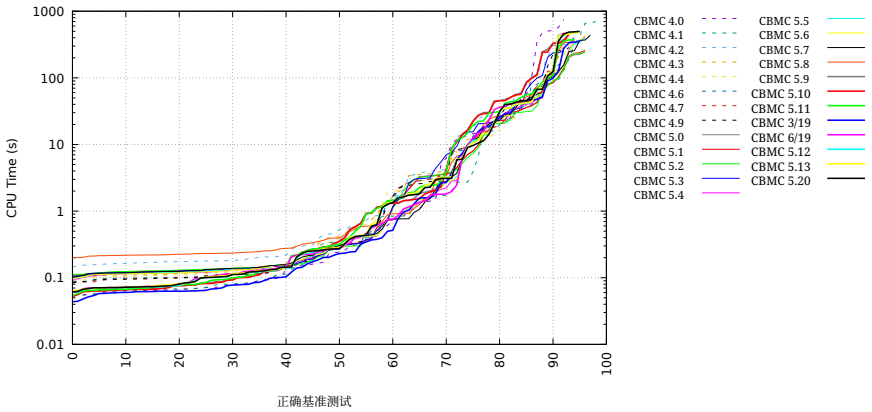

图 6：ReachSafety-Loops 上的 CPU 时间分位数图

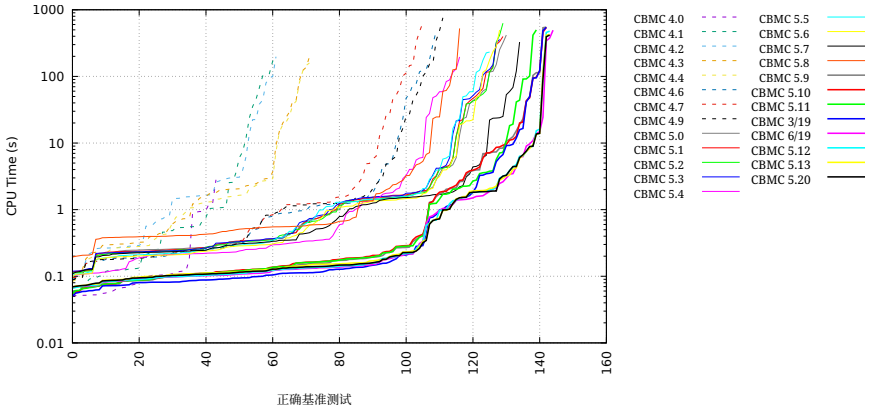

图 7：MemSafety-Heap 上的 CPU 时间分位数图

将静态单赋值编码为 SAT 公式之前进行表达式简化。CBMC 5.9 引入了编译器内置函数的按需定义，使运行时间恒定缩短了半秒。这一效果在运行时间较短的基准测试中尤为明显。

各类别的改进幅度并不一致。例如，CBMC 在 ReachSafety-Loops 类别中的性能变化很小（见图 6）。这些基准测试都是相当简单的整数程序，CBMC 4.0 已能非常高效地支持它们，并且以相近的速度解决了大致相同数量的基准测试。

在图 7 中，MemSafety-Heap 基准测试呈现出截然不同的情况。这些基准测试使用了更复杂的语言特性，例如动态内存分配、指针和结构体。CBMC 不仅变得显著

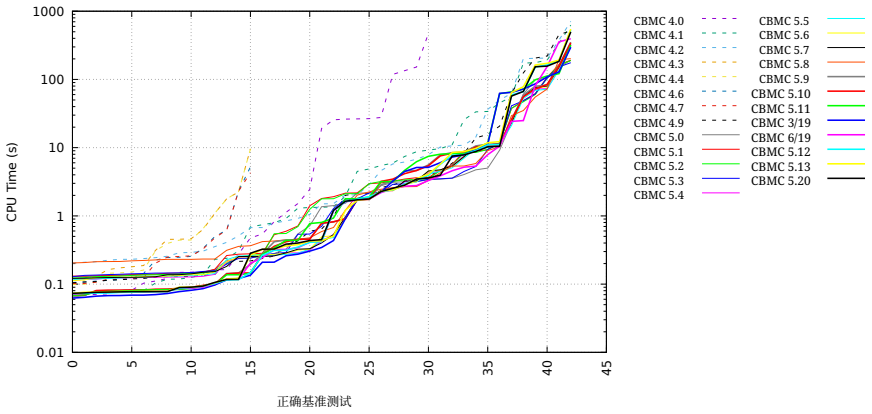

图 8：ReachSafety-Floats 基准测试的 CPU 时间分位数图

更快，而且产生的错误结果也少得多（从 CBMC 4.6 中的六个错误结果减少到 CBMC 5.9 以来的一个错误结果）。CBMC 4.4 与 4.6 之间的差距源于引入了内存泄漏插桩，使得证明这些性质成为可能。

3/19 至 5.20 期间的最新改进得益于符号执行方面的重大增强。例如，经过优化的数据结构用于执行常量传播和存储指向集，以避免不必要的复制。此外，还引入了针对结构体的字段敏感常量传播和针对数组的单元敏感常量传播，并引入了条件传播，以筛选指向集并避免探索不可行分支。

图 8 展示了 ReachSafety-Floats 基准测试中 CPU 时间的变化。CBMC 的浮点决策过程经历了一段持续修复程序错误和优化的时期，尤其是在 CBMC 4.2 至 5.0 之间。CBMC 5.8 引入了更完善的 math.h 内置库，这解释了为何后续的 5.x 版本能够解决比早期版本多得多的基准测试。

总体而言，我们观察到，CBMC 的性能演进并非主要得益于少数几个带来巨大性能提升的重要功能，而是通过持续进行渐进式改进和修复程序错误逐步提升的。

7 未来方向

CBMC 已证明能够验证大型真实软件项目并找出其中的程序错误。尽管取得了这些成功 [8], ，但在此类场景中使用软件验证工具绝非易事，往往需要专家用户。此外，这还需要

投入大量人工工作，以模块化方式拆分应用程序、分而治之，并编写测试框架和桩函数，同时对程序运行环境作出切合实际的假设。因此，除了持续努力提升 CBMC 的性能外，提高 CBMC 用户使用该工具的便利性也是开发工作的主要重点。

参考资料

1. Alglave, J., Kroening, D., Tautschnig：用于高效并发软件有界模型检测的偏序。载于：计算机辅助验证，*LNCS*，第8044卷，第141–157页（2013）2. Alpern, B., Wegman, M.N., Zadeck, F.K.：检测程序中的变量相等性。载于：POPL，第1–11页（1988）3. 美国国家标准学会：ANSI/ISO/IEC 9899-1999：程序设计语言——C。美国国家标准学会，美国纽约州纽约市百老汇大道1430号，邮编10018（1999）4. Barrett, C., Sebastiani, R., Seshia, S.A., Tinelli, C.：理论可满足性，人工智能与应用前沿，第185卷，第26章，第825–885页。IOS出版社（2009）5. Barrett, C.W., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.：CVC4。载于：G. Gopalakrishnan, S. Qadeer（编）计算机辅助验证——第23届国际会议，CAV 2011，美国犹他州Snowbird，2011年7月14日至20日。论文集，计算机科学讲义，第6806卷，第171–177页。施普林格（2011）。DOI 10.1007/ 9781 14 6 3 642 22110 1 14101007978364222110- - - - \。URL https://doi.org/./----。Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.：无需二元决策图的符号模型检测。载于：系统构造与分析工具和算法，计算机科学讲义，第1579卷，第193–207页。施普林格（1999）7. Biere, A., Heule, M., van Maaren, H., Walsh, T.（编）：可满足性手册，人工智能与应用前沿，第185卷。IOS出版社（2009）8. Chong, N., Cook, B., Eidelman, J., Kallas, K., Khazem, K., Monteiro, F.R., Schwartz- Narbonne, D., Tasiran, S., Tautschnig, M., Tuttle, M.R.：亚马逊云科技软件开发工作流中的代码级模型检测。《软件实践与经验》**51**(4)，772–797（20212949 9 10 1002 2949101002）。DOI . /spe. 。URL https://doi.org/./spe.。Clarke, E.M., Kroening, D., Lerda, F.：用于检查ANSI-C程序的工具。载于：计算机辅助系统分析与转换，*LNCS*，第2988卷，第168–176页（2004）10. Clarke, E.M., Kroening, D., Yorav, K.：使用有界模型检测验证C语言和Verilog程序的行为一致性。载于：设计自动化会议，第368–371页。美国计算机协会（2003）11. Cook, B., Döbel, B., Kroening, D., Manthey, N., Pohlack, M., Polgreen, E., Tautschnig, M., Wieczorkiewicz, P.：使用模型检测工具评估Xen虚拟机监控器中安全漏洞的严重程度。载于：2020年计算机辅助设计形式化方法，FMCAD 2020，以色列海法，2020年9月21日至24日，第185–193页。电气电子工程师学会（2020）。DOI 10.34727/2020/isbn. 9786 26 12 3 85448 042 6 26103472720209783854480 42- - - - \。URL https://doi.org/.//isbn.----。Cook, B., Khazem, K., Kroening, D., Tasiran, S., Tautschnig, M., Tuttle, M.R.：对亚马逊云科技数据中心的引导代码进行模型检测。载于：计算机辅助验证——第30届国际会议，CAV 2018，作为逻辑联合会议FloC 2018的一部分举行，英国牛津，2018年7月14日至17日。论文集，第二部分，计算机科学讲义，第10982卷，第467–486页。施普林格（2018）。DOI 10.1007/978-3-319-96142-2\ 28。URL https://doi.org/10.1007/978-3-319-96142-2 2813. Cordeiro, L.C., Kesseli, P., Kroening, D., Schrammel, P., Trtík, M.：JBMC：用于验证Java字节码的有界模型检测工具。载于：计算机辅助验证，CAV，*LNCS*，第10981卷，第183–190页。施普林格（2018）14. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.：计算静态单赋值形式的高效方法。载于：POPL，第25–35页（1989）

15. Kroening, D., Strichman, O.：决策过程——算法视角，第二版。理论计算机科学丛书：欧洲理论计算机科学协会系列。施普林格（2016）。DOI 100 16 1007 978 3 662 50497 0101007978366250497. / - - - - 。URL https://doi.org/./----。Kroening, D., Tautschnig, M.：大规模自动化软件分析。载于：MEMICS，计算机科学讲义，第8934卷，第30–39页。施普林格（2014）。DOI 10.1007/ 9780 3 17 3 319 14896 0 3101007978331914896- - - - 。URLhttp://dx.doi.org/./----。Malacaria, P., Tautschnig, M., Distefano, D.：复杂C代码的信息泄漏分析及其在OpenSSL中的应用。载于：ISoLA，计算机科学讲义，第9952卷，第909–925页（2016）。DOI 10.1007/978-3-319-47166-2 63。URL http://dx.doi.org/10.1007/978-3-319-47166-2 6318. de Moura, L.M., Bjørner, N.：Z3：高效的SMT求解器。载于：C.R. Ramakrishnan, J. Rehof（编）系统构造与分析工具和算法，第14届国际会议，TACAS 2008，作为欧洲软件理论与实践联合会议ETAPS 2008的一部分举行，匈牙利布达佩斯，2008年3月29日至4月6日。论文集，计算机科学讲义，第4963卷，第337–340页。施普林格（2008）。DOI 10.1007/ 9783 24 19 3 540 78800 3 2410100797835407880 0- - - \ 。URL https://doi.org/./----。Nemer, F., Cassé, H., Sainrat, P., Bahsoun, J.P., Michiel, M.D.：PapaBench：免费的实时基准测试。载于：2006年第四届国际最坏情况执行时间分析研讨会，2006年7月，德国德累斯顿，*OASICS*，第（ ）卷。URL http://drops.dagstuhl.de/opus/volltexte/2006/67820. Nieuwenhuis, R., Oliveras, A., Tinelli, C.：抽象DPLL与抽象理论模DPLL。载于：LPAR，计算机科学讲义，第3452卷，第36–50页。施普林格（2004）21. Nieuwenhuis, R., Oliveras, A., Tinelli, C.：求解SAT和理论模SAT：从抽象Davis–Putnam–Logemann–Loveland过程到DPLL()。《ACM期刊》**53**(6)，937–977（2006）22. Rosen, B.K., Wegman, M.N., Zadeck, F.K.：全局值编号与冗余计算。载于：POPL，第12–27页（1988）23. 开放组织：数据大小中立性与64位支持。电气电子工程师学会（1998）