arXiv:2302.02380v1 [cs.SE] 5 Feb 2023

用于程序分析的 2LS

Daniel Kroening、Viktor Malík、Peter Schrammel 和 Tomáš Vojnar

1 引言

2LS（“工具”）是一款基于 CPROVER 框架构建的 C 程序验证工具。它可以验证用户指定断言、内存安全属性（例如缓冲区溢出）、数值溢出、除以零、内存泄漏以及终止属性。

分析过程通过将验证任务转换为基于位向量、数组和浮点算术理论的二阶逻辑公式来完成。该公式借助增量 SAT 求解，通过结合展开和基于模板的不变式综合等算法进行求解。

2LS 的优势包括速度极快的增量有界模型检测算法，以及一个灵活的框架，可用于试验新的分析和抽象思路，以进行不变式推断。其缺点是缺少对某些程序特性（例如多线程）的支持。表 1 概述了 2LS 的特性。本章其余部分的结构如下：

- 第 2 节概述了 2LS 验证方法背后的动机和理论。

Daniel Kroening 牛津大学，英国；Diffblue Ltd，英国；电子邮箱：kroening@cs.ox.ac.uk Viktor Malík 信息技术学院，布尔诺理工大学，捷克共和国；IT4Innovations 卓越中心，捷克共和国；电子邮箱：imalik@fit.vutbr.cz Peter Schrammel 萨塞克斯大学，布莱顿，英国；Diffblue Ltd，英国；电子邮箱：p.schrammel@sussex.ac.uk Tomáš Vojnar 信息技术学院，布尔诺理工大学，捷克共和国；IT4Innovations 卓越中心，捷克共和国；电子邮箱：vojnar@fit.vutbr.cz

表 1：2LS 的特性。

语言	C语言、GOTO
属性	断言、内存安全、溢出、除零、终止性
环境	Linux
所使用的技术	符号执行、有界模型检测 k-归纳 抽象解释 基于模板的谓词合成 排名函数、回归集 SAT 和 SMT 求解
其他功能	多个输入文件，以及将整个项目编译并链接为 通过 goto-cc 转换为 GOTO
当前优势	内存安全、浮点、循环
当前不足	递归、多线程

- 第 3 节介绍了有关 2LS 的实用信息，以及如何使用它来完成特定类型的验证任务，例如证明函数性质，以及检查内存安全性、终止性和溢出。
- 第 4–6 节介绍 2LS 背后的理论概念。
- 第 7 节详细介绍 2LS 软件项目及其架构。
- 第 8 节报告了软件验证竞赛中的实验比较数据，2LS 自 2016 年起一直参与该竞赛。

2 验证方法

软件验证文献中包含各种技术，可用于证明或证伪有关程序的性质，每种技术都有其优势和不足。这对非专业用户而言颇具挑战。通常，选择使用哪些工具以及将精力投入何处，取决于待验证的性质是否成立——而这恰恰是他们想要弄清楚的。因此，要构建稳健且易用的软件验证系统，就有必要结合多种技术。

一种选择是并行运行一系列相互独立的工具（例如，作为一个工具组合），或者按某种顺序依次运行。然而，这会限制算法之间可交换的信息。

另一种选择是采用单体式算法，例如 CEGAR [9], IM- PACT [27], 或 IC3/PDR [3, 18]，它们融合了较简单系统的一些理念 。由于各组件之间存在复杂而微妙的交互，要将这些算法实现好并不容易。此外，它们还需要高级求解器功能，例如插值生成，而这些功能并非所有理论（位向量、数组、浮点数等）都广泛支持。如下文所述，2LS 采用了不同的问题处理方法。

2.1 *k*归纳与 *k*-不变式（*k*I*k*I）

2LS 致力于将验证器构建为由简单组件组成、且组件间交互易于理解的整体。这些组件包括：

有界模型检测 在时间和资源充足的情况下，BMC 能为所有错误的安全性质给出反例，这对于理解故障通常具有重要价值。然而，BMC 只能证明少数真属性。

k-归纳 借鉴 Hoare 逻辑中循环不变式的思想，*k*-归纳可以证明真安全性质，在某些情况下还能为错误的安全性质提供反例。但它需要归纳不变式，而这可能代价高昂（需要用户投入时间、专业知识并进行维护）。

抽象解释 使用过近似可以轻松计算不变式，从而证明许多真命题。然而，错误的性质与虽真却无法证明的性质可能难以区分。工具对更完备分析的支持可能有限。

2LS 的 *k*I*k*I 算法 [4] 汇集了多种著名技术，并以新颖的方式将它们结合起来，使其相互增强、彼此补足。*k*-归纳技术 [36] 使用语法受限或简单的不变式（例如由抽象解释生成的不变式）来证明安全性。有界模型检测 [2] 使我们能够检查 *k*-归纳失败，判断它们是否是真实反例；如果不是，则逐步构建一组关于系统行为的假设。基于模板的抽象解释用于不变式推断 [32, 31, 16]，而展开会逐步生成更强的不变式，使这些技术能够相互增强。

*k*I*k*I 通过一个简洁优雅的算法，将这些技术结合起来，用于集成不变量推断和反例生成。k*I*k*I* 的主循环基于对转移关系进行增量展开。在第 *k* 次迭代中，转移关系会展开至深度 *k*，并推断出一个 *k*-归纳不变式。随后，利用该不变式强化程序的安全性属性，以证明程序的安全性。如果无法证明安全性，算法就会检查当前的展开是否足以生成反例。如果不足，则增加展开次数 *k*，并开始下一次迭代。第5节将更详细地介绍 k*I*k*I* 算法。

在内部，2LS 会将以二阶逻辑表达的程序分析问题（例如不变式或秩函数推断）归约为针对模板的综合问题。因此，它将（二阶逻辑求解中的）存在片段归约为一阶逻辑中的量词消去。边界模型检测所使用的循环展开以及 *k*-归纳被视为对原始程序抽象的细化。由此，2LS 基于增量循环展开、基于模板的谓词合成，最终还基于增量 SAT 求解，为 *k*I*k*I 提供了高效实现。第 4、5 和 6 节将更详细地介绍这些概念以及 2LS 的其他重要基础概念。

```
1  void copy_authority(2 char *uri, int uri_length, int authority_start , char *
authority) {3  __CPROVER_assume(0  <uri_length);4  __CPROVER_assume(0  <
authority_start && authority_start  <uri_length);5  int cp  =  authority_start;6  while (cp !=
uri_length - 1) {7  if (uri[cp]  ==  '/')8  break;9  断言(cp  <uri_length);10  authority[c
p - authority_start]  =  uri[cp];11  ++cp;12  }13  }
```

图 1：uri.c：循环中的数值计算。

3 使用场景

在介绍 2LS 背后的理论概念之前，我们先通过教程概述 2LS 的验证能力。2LS 提供了丰富的功能，使用户能够解决各种验证问题。我们将使用示例程序，这些程序可在 https://github.com/diffblue/2ls/tree/master/regression/book-examples 找到。

3.1 证明函数性质

在 C 程序中，可以借助 assert.h 提供的断言来指定函数性质。为了声明前置条件，CPROVER 框架提供了 __CPROVER_assume(cond) 函数。[1]

作为第一个示例，我们来看图 1 中的函数。该函数将以字符字符串 uri 给出的统一资源标识符的权限部分提取到缓冲区 authority 中。我们希望确保不会复制超出统一资源标识符长度的内容，这一点由第 9 行的断言表达。

2LS 提供了多种用于验证此属性的算法。例如：

- 2ls uri.c --function copy_authority --havoc 2LS 使用不变式 *true* 执行 1-归纳检查。这不足以证明该属性。因此，它返回 未知。
- 2ls uri.c --function copy_authority --intervals 2LS 使用区间不变量执行 1-归纳检查。但这仍然不足。我们得到的结果是 未知。

[1] 为了确保代码能够编译，可以将此函数放在 #ifdef _CPROVER_ ... #endif 之间；运行 2LS 时会定义该宏。

- 2ls uri.c --function copy_authority --zones 使用更强的区域抽象域后，我们能够证明该性质。2LS 输出 通过。
- 2ls uri.c --function copy_authority --octagons 更强的八边形抽象域也能让我们证明该性质。2LS 返回 OK。
- 2ls uri.c --function copy_authority --havoc --k-induction 2LS 执行 k-归纳。尽管我们没有推导出额外的不变式（havoc 选项），这已足以证明该性质。2LS 报告 OK。

上面的示例涉及数值数据。现在让我们来看图 2 中的程序。该程序会分配一个长度任意的单链表，其中的元素保存着某个整数值。[2] 随后，它计算链表元素所保存值的最小值，并检查该最小值是否计算正确；也就是说，它必须小于或等于链表中的所有值。

```
1 外部 int __VERIFIER_nondet_int(); 2 3 #include <
stdlib.h>4 #include <limits.h>5 6 #define APPEND(l,i) {i->
next=l;l=i;} 7 8 typedef struct node { 9 struct node *next;
10 int val; 11 } Node; 12 13 int main() { 14 Node *l =
NULL; 15 int min = INT_MAX; 16 17 while
(__VERIFIER_nondet_int()) { 18 Node *p =
malloc(sizeof(*p)); 19 p->val = __VERIFIER_nondet_int();
20 APPEND(l, p) 21 22 if (min >p->val) 23 min =p->val;
24 } 25 for (Node *i = l; i != NULL; i =i->next) 26 断言(i->
val >=min); 27 }
```

图 2：min.c：存储在动态分配的数据结构中的数据。

[2] __VERIFIER_nondet_int 函数已声明但没有定义。2LS 假定未定义的函数会返回一个非确定性值。

2LS 能够证明这一点，因为它可以同时对无界数据结构的形状及其内容进行联合推理。运行

2ls min.c --heap --values-refine

对于第 263 行的属性，将返回 OK 。这意味着，任何大小的任何数组都不会违反该断言。

3.2 查找内存安全漏洞

另一类可使用 2LS 验证的属性是内存安全属性。这些属性包括数组访问不越界、指针解引用安全性、释放安全性，以及不存在内存泄漏等。2LS 通过使用自定义断言对程序进行插桩来验证这些属性。运行 2LS 并指定 --help 选项，即可查看可用插桩的完整列表。

我们通过两个示例演示内存安全属性的验证。首先，来看图3中的程序，它将二维数组 matrix 转换为一维数组 array。我们将矩阵的维度设为随机值，但在第 7 行添加了一个前置条件，以确保矩阵元素的数量不超过目标数组的容量。

```
1 #define SIZE 1000
2
3 int main()
4 {
5   int **matrix;
6   unsigned m = __VERIFIER_nondet_int();
7   unsigned n = __VERIFIER_nondet_int();
8   __CPROVER_assume(m <= SIZE && n <= SIZE && m*n <= SIZE);
9
10  int 数组[SIZE];
11  for (int row = 0; row <m; ++row) {
12    for (int col = 0; col < n; ++col) {
13      int index = row *m + col;
14      数组[index] = matrix[row][c
ol];
15    }
16  }
17  return 0;
18 }
```

图 3：matrix_to_vector.c：数组界安全性。

使用以下命令运行验证：2ls --bounds-check --k-induction matrix_to_vector.c

[3] --heap -values-refine 选项采用形状域和数值域的组合。

将返回 失败 ，并提供有关失败的特定生成断言的更多信息：

[main.2] 数组 'array' 在数组中的上界[索引]: 失败

此信息表明，索引可能大于数组的上界（SIZE）。此外，如果 2LS 发现性质违规，它就能提供一个具体反例。我们可以通过使用额外的 --trace 选项重新运行 2LS 来获取该反例。图4展示了所获得执行轨迹中的重要部分。

... 文件 main.c 第 5 行 函数 main m=1000u 文件 main.c 第 6 行 函数 main n=1u ... 文件 main.c 第 10 行 函数 main row=1文件 main.c 第 11 行 函数 main col=0文件 main.c 第 12 行 函数 main index=1000

图4：图3中程序发生数组边界安全违规的反例。

通过反例，我们发现第12行有误，正确写法应为：12 int index = row * n + col;

更正错误后再次运行验证，结果为OK，这意味着程序中不再存在数组越界访问。

作为内存安全验证的第二个示例，我们来考察图5所示的、包含动态内存分配的程序。该程序分配了两个由指针 a 和 b 指向并相互链接的内存对象（b 指向的对象的值等于 a，而 a 指向一个随机整数）。随后，这两个已分配的对象都会被释放。然而，释放顺序有误，因为 b 会先被释放，之后尝试释放其指向的对象时，很可能会由于对悬空指针进行无效解引用而失败。运行以下命令即可使用 2LS 验证程序中不存在指针解引用错误和内存释放错误：

2ls --pointer-check --heap free_safety.c

如预期，在所示程序上运行此命令会返回FAILURE ，并附带有关错误的更多信息：

[main.11]对 *b 解引用失败：已释放的动态对象：失败

```
1 int main() {2 int *a =
malloc(sizeof(int));3 *a =
__VERIFIER_nondet_int();45 int **b =
malloc(sizeof(int *));6 *b =a;78
free(b);9 free(*b);10 }
```

图 5：free_safety.c：释放安全性验证。

3.3 终止与非终止

2LS 可以证明一个程序是否会终止，或在某些输入条件下是否可能不会终止。

```
1 void 冒泡排序(int *数组, int size)2 {3
__CPROVER_assume(size >= 0);4 int c, d,
swap;56 for (c = 0; c < size - 1; c++)7 {8 for
(d = 0; d < size - c - 1; d++)9 {10 if (数组[
d] > 数组[d + 1])11 {12 swap =
数组[d];13 数组[d] = 数组[d + 1];14 数组[d
+ 1] = swap;15 }16 }17 }18 }
```

图 6：bubble_sort.c：证明终止性。

例如，图 6 展示了冒泡排序算法的一种实现。我们可以通过运行以下命令检查它是否会终止：

2ls bubble_sort.c --termination --function bubble_sort

我们会得到类似以下的输出：

... 函数冒泡排序的摘要

```
... 终止性论证: ($guard#25 && ... ==> (-1 * (d#phi25 - d#30) + 0
* ...) > 0) && ($guard#23 && ... ==> (-1 * (c#phi23 - c#32) + 0 * ...)
> 0) ... ** 终止性: ...[冒泡排序]: 是 验证成功
```

2LS 以秩函数的形式计算终止性论证。上方输出中的终止性论证经过了大幅简化，以便在此页面上显示。2LS 实际输出的终止性论证包含更复杂的守卫约束和类型转换，从而能够在不发生溢出的情况下正确进行位向量算术。检查该终止性论证可知，-c 是外层循环的秩函数，-d 是内层循环的秩函数，这证明了 bubble_sort 函数的终止性。

```
1 long sasum(long *sx, int n, int incx)2 {3 int nincx =n* incx;4 long
stemp = 0l;5 int i;6 for (i = 0; incx < 0 ? i >= nincx : i <= nincx; i
+= incx)7 {8 stemp +=sx [i-1];9 }10 return stemp;11 }
```

图7：sasum.c（参见 http://www.netlib.org/clapack/cblas/sasum.c）。

2LS 还可以检查非终止性。它通过查找套索形执行来实现这一点。图7展示了一个来自线性代数软件包的数组求和算法。运行

```
2ls sasum.c --function sasum --nontermination --trace
```

会得到图8所示的输出。检查反例轨迹后，我们可以看到变量 i 在第 6 行反复取值 0，这意味着循环不会终止，因为循环条件永远不会变为假。我们甚至可以看到这种情况发生时输入所需满足的一个条件示例：n=2，incx=0。

在实践中，可以并行运行 --termination 和 --nontermination，并等待其中一个给出确凿结果。

```
展开（k=1）... 已证明程序执行不会终止（展开 1 次后）... 反例：文件
sasum.c 第 1 行 n=2文件 sasum.c 第 1 行 incx=0... 文件 sasum.c 第
6 行，函数 sasum，i=0... 文件 sasum.c 第 6 行，函数 sasum，i=0验
证失败
```

图 8：非终止示例的输出。

3.4 溢出

另一项可以使用 2LS 验证的性质是免受整数溢出（和下溢）的安全性。与内存安全验证类似，这是通过在被分析程序中插入自定义断言来实现的。

作为示例，我们使用图 6 中的程序。在该程序中，第 3 行的前置条件 __CPROVER_assume(size >= 0) 对于避免计算 size - 1 时发生有符号算术下溢是必要的；在 2LS 支持的 C 标准中，这属于未定义行为。我们可以运行 2LS 来检查是否存在溢出，从而验证这一点：

```
2ls bubble_sort.c --signed-overflow-check --function 冒泡排序
```

如果没有前置条件，我们会得到：

```
...[冒泡排序.1] 有符号数的算术溢出 - size - 1：失败 ... 验证失败
```

这证实了前置条件的必要性。

4 程序内部表示

LS 构建于 CPROVER 基础设施之上 [8], 2，并使用 *GOTO* 程序 作为中间表示。在这种语言中，任何非线性控制流，

例如 if 或 switch 语句、循环或跳转，都会被转换为等效的守卫式 *goto* 语句。这些语句是包含（可选）条件的分支指令。CBMC 会为解析树中找到的每个 C 函数生成一个 GOTO 程序。此外，它还会添加一个新的 main 函数，该函数先调用一个用于初始化全局变量的函数，然后再调用原始程序入口函数。

与 CBMC 类似，2LS 执行轻量级静态分析，通过对所有候选函数进行分情况讨论来解析函数指针，从而生成静态调用图。此外，还会插入用于防范无效指针操作或内存泄漏的断言。另外，2LS 使用局部常量传播和表达式简化来提高效率。

在执行上述转换后，2LS 对 GOTO 程序的每个函数执行静态分析，以推导数据流方程。其结果是静态单赋值（*SSA*）形式，其中循环在回边处被截断，回到循环头。截断的作用是在循环头清空循环中被修改的变量。因此，该 SSA 是 GOTO 程序的过近似。随后，2LS 通过计算不变式来细化这一过近似。

由于该 SSA 表示对于 2LS 的验证方法至关重要，我们将在第4.2节对其进行详细描述。在此之前，我们将介绍用于在 SSA 形式中表示动态分配对象的内存模型（第 4.1 节）。最后，我们将在第 4.3 节介绍循环展开方法，这是该算法的一个重要步骤。

4.1 内存模型

现在我们描述一种用于表示所有程序内存的内存模型 [22]。我们的模型基于对象，并区分静态分配的对象 （即栈上的变量和全局变量）与动态分配的对象 （即堆上的对象）。

4.1.1 静态内存对象

我们的方法针对所有函数均已内联的非递归程序。因此，无需考虑栈，而静态内存对象的集合 *Var* 对应于所有程序变量的集合。每个变量都由其名称唯一标识。

为方便起见，我们定义 *Var* 的若干子集，分别对应于选定类型的变量集合：

- *NVar* 是所有数值类型变量（整数或浮点）的集合。
- *PVar* 是所有指针类型变量的集合。
- *SVar* 是所有结构体类型变量的集合。每种结构体类型都定义了一组命名字段，每个字段都有自己的类型。我们用*Fld* 表示被分析程序中使用的所有字段的集合。与变量类似，我们

令$NFld$,$PFld \subseteq Fld$ 分别表示所有数值类型字段和指针类型字段的集合。为了表示对结构体类型变量中各个字段的访问，我们使用 C语言中常见的“点”表示法。

我们假设$NVar$、$PVar$和$SVar$ 两两不相交。

4.1.2 动态内存对象

为了表示动态内存对象（即使用 malloc 或其某个变体分配的对象），我们使用抽象动态对象。一个抽象动态对象表示由同一次 malloc 调用分配的一组具体动态对象。我们将程序位置 i 处的 malloc 调用称为分配点 i。

通常，单个抽象动态对象不足以表示由同一次 malloc 调用分配的所有具体对象。这是因为被分析程序可能会同时使用在同一分配点分配的多个具体对象。例如，如果比较这些对象，我们的内存模型就必须能够区分它们。这可以通过按需具体化（许多内存模型都采用这种方式），也可以通过在分析开始时预先物化足够数量的对象来实现。由于我们的方法使用单个公式表示被分析程序，并在分析过程中利用公式的小幅增量变化，因此我们采用后一种方法。

因此，我们使用抽象对象集合 $AO_i = \{ao_i^k \mid 1 \leq k \leq n_i\}$ 来表示在分配点 i 分配的所有具体对象。每个分配点所需的对象数量 n_i 采用本节稍后介绍的方法确定。随后，被分析程序的所有动态对象集合定义为 $AO = \cup_i AO_i$。结合所有静态对象集合，我们将 $Obj = Var \cup AO$ 定义为程序抽象中所有内存对象的集合。我们要求 $Var \cap AO = \emptyset$，且 $AO_i \cap AO_j = \emptyset$ 对于 $i \neq j$。

与静态对象类似，我们分别用 NAO、PAO 和 SAO 表示数值类型、指针类型和结构体类型的动态对象集合。结构体类型动态对象的字段（即集合 $SAO \times Fld$ 的元素）表示所有被表示的具体对象中相应字段的抽象。利用上述集合，我们可以将程序中所有数值对象 组成的集合 Num 定义为：

$$Num = NVar \cup NAO \cup ((SVar \cup SAO) \times NFld). \tag{1}$$

类似地，我们将所有指针的集合Ptr 定义为：

$$Ptr = PVar \cup PAO \cup ((SVar \cup SAO) \times PFld). \tag{2}$$

指针可以被赋值为对象的地址。由于我们不支持指针运算，因此只考虑符号地址 和一个特殊地址空指针 。我们使用运算符 & 获取静态对象和动态对象的地址。对于抽象动态对象，其符号地址是所有被表示的具体对象的符号地址的一种抽象。我们将程序中所有地址的集合定义为$Addr$

as:

$$Addr = \{\&o \mid o \in Obj\} \cup \{\mathsf{null}\}. \tag{3}$$

动态对象预物化

上文提到，对于每个分配点 i，我们用有限个 n_i 抽象动态对象表示在 i 分配的所有对象（其数量可能是无限的）。为了使这种抽象可靠，只需确保这个数量等于在被分析程序的任意位置可能同时被指向的、在 i分配的不同具体对象的数量。

为了计算这个数值，我们首先执行标准的静态可能别名分析。此分析会确定，对于每个程序位置 j，集合 P_i^j 中包含程序里使用的所有指针表达式，这些表达式可能指向在i处分配的某个对象。这里，指针表达式可以是以下类型之一：

- 指针变量。
- 指针的解引用（指向指针）。这些对应于指针类型动态对象。

- 结构体类型变量的指针字段。
- 对指向结构体的指针进行解引用后，再访问指针类型字段。这些字段对应于动态对象的指针类型字段。这里，我们使用基于箭头的 C表示法（例如 $p\to n$，表示对 p进行解引用后，再访问所指对象的字段 n）。

为简单起见，我们假设所有形式为 $**p$ 或 $p\to f_1\to f_2$ 的链式解引用都会拆分为多个解引用，并使用一个加入 $PVar$ 的中间指针变量。

接下来，我们在所有指针表达式的集合上计算必然别名关系 $\sim_j$。对于每一对指针表达式 p 和 q，以及每个程序位置 j，当且仅当 p 和 q 在 j 必然指向同一个具体对象（即它们必然别名）时，$p \sim_j q$。

最后，我们按 $\sim_j$ 将计算得到的每个 P_i^j 划分为等价类，而数量 n_i 取任意 j 对应的此类等价类的最大数目。

4.2 静态单赋值形式

2LS 中的程序验证基于使用求解器生成程序抽象。为简化表示程序语义的公式的生成过程，2LS 使用静态单赋值形式 (SSA)来表示程序。SSA 是一种标准的程序表示形式，用于有界模型检测或符号执行工具。我们采用 SSA 的常见概念——在程序位置 i处，如果将 x赋值为i，则为每个变量 x引入一个新的副本（版本）x_i；每当读取 x时，使用 x的最新版本；如果来自合并的程序分支的 x 版本不同，则在程序汇合点 i 处引入一个φ 变量 x_i^{phi}。

对于无环程序，SSA 是一个能够准确表示运行代码后置条件的公式。

我们通过引入三个新概念来扩展传统 SSA：（1）对循环进行过近似，使 SSA 变为无环 [4]，（2）对控制流进行特殊编码 [4]，；以及（3）对符合第 4.1 [22] 节所介绍内存模型的内存操作进行特殊表示。本节其余部分将介绍这些扩展。

4.2.1 循环的过度近似

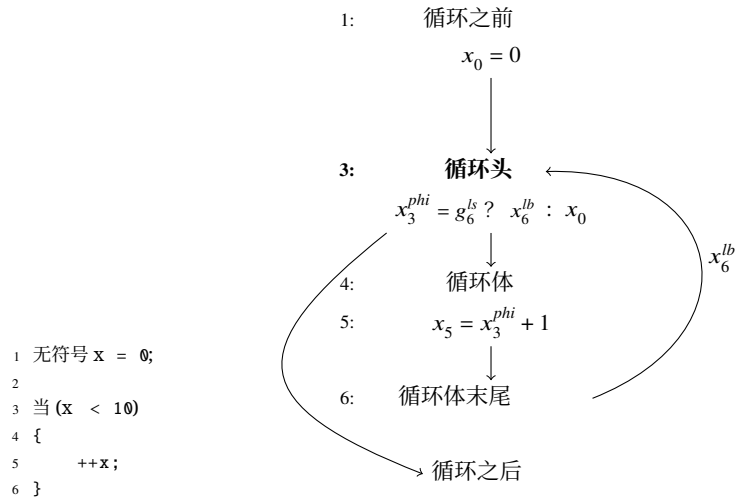
为了能够使用求解器对程序抽象进行推理，我们通过对循环效果进行过近似来扩展 SSA。如上所述，变量 x 的值在循环头处由一个 phi变量 x_i^{phi} 表示，它将循环前的 x 的值与循环体末尾的值合并（这里假设循环中的所有路径都会在循环末尾汇合，循环前的路径也同样如此）。然而，我们不使用循环末尾的 x 版本，而是将其替换为自由循环回溯变量 x^{lb}。这样一来，SSA 仍保持无环；并且由于 x^{lb} 的值最初不受约束，循环的效果便得到了过近似。为提高精度，之后可以使用循环不变量约束 x^{lb} 的值，该不变量将在分析过程中推导出来。循环不变量是在循环体末尾（任意一次迭代之后）成立的性质，因此可以假定它对循环回溯变量也成立。

为了更好地说明这一点，我们举例介绍这种 SSA 扩展。图 9 展示了 C 语言中的一个简单循环及其对应的 SSA。phi变量中不再使用 x_5，而是引入一个新变量 x_6^{lb}。此外，phi节点中的合并由一个自由布尔变量 g_6^{ls} 驱动（即所谓的循环选择守卫），用于对 x_6^{lb} 和 x_0 之间的非确定性选择进行建模。

4.2.2 编码控制流

在 2LS 中，程序由一个单体公式表示。因此，公式需要对控制流信息进行编码。这通过使用所谓的守卫变量实现，这些变量跟踪每个程序位置的可达性信息。具体而言，对于每个程序位置 i，我们引入一个布尔变量g_i，其值编码 i 是否可达。例如，在图 9 中，对循环体的可达性进行编码的守卫 g_5取值为：

$$g_5 = x_3^{phi} < 10. \tag{4}$$




```
1 无符号 x = 0;
2
3 当 (x < 10)
4 {
5       ++x;
6 }
```

（a）C语言中的一个循环。

（b）循环的 SSA 形式。

图 9：2LS 中使用的静态单赋值形式下循环的转换。

4.2.3 内存操作的表示

动态内存分配

如第 4.1 节所述，在程序位置 i 处通过一次 malloc 调用分配的所有具体对象，都由一组抽象动态对象 AO_i抽象表示。在静态单赋值形式中，我们用从 AO_i 中的对象进行非确定性选择来表示此类调用。因此，程序赋值 `p = malloc(...)` 会被转换为公式

$$p_i = g_{i,1}^{os} \, ? \, \&ao_i^1 : (g_{i,2}^{os} \, ? \, \&ao_i^2 : (\ldots (g_{i,n_i-1}^{os} \, ? \, \&ao_i^{n_i-1} : \&ao_i^{n_i}))) \tag{5}$$

其中 $g_i^{os}{}_{,\ j}, 1 \leq j < n_i$ 是自由布尔变量，称为对象选择守卫。

读取解引用指针

现在我们介绍如何对出现在赋值右侧或条件中的指针解引用进行编码（即在 R 表达式中）。在生成 SSA 之前，我们会执行静态可能指向分析，对每个程序位置 i 和每个指针 $p \in Ptr$，近似计算来自 Obj 的所有对象的集合，其中p 在 i处可能指向这些对象。随后，将 p 在 i 处的解引用表示为在这些被指向对象中的一种选择。

此外，为了简化表示并提高精度（原因将在下文解释），我们还引入所谓的解引用变量。设 $*p$ 是出现在程序位置 i 处的一个 R 表达式，并假设 p 可能指向一组对象 $O \subseteq Obj$。我们用一个新的变量 $drf(p)_i$ 替换 $*p$，并按如下方式定义其值：

$$\bigwedge_{o \in O} p_j = \&o \implies drf(p)_i = o_k \wedge \left(\bigwedge_{o \in O} p_j \neq \&o \right) \implies drf(p)_i = o_\perp \tag{6}$$

其中，p_j 和 o_k 分别是在程序位置 i 处变量 p 和 o 的相关版本，而 $o_\perp$ 是一个特殊的“未知对象”，表示对未知或无效的（空指针）指针进行解引用的结果。

通俗地说，(6)表示这样一个事实：如果在 i 处 p 指向 o，那么 $drf(p)_i$ 就等于 i 处 o 的值；否则，它就等于未知对象的值。

通过解引用指针进行写入

与读取操作类似，我们为使用指针解引用进行内存写入的操作引入一种 SSA 编码。同样，我们利用上文介绍的可能指向分析，并引入特殊的解引用变量。设在程序位置 i 处有一条赋值 $*p = v$，并假设在进入 i 时，p 可能指向一组对象 $O \subseteq$ 对象。那么，这条赋值可替换为一个等式：

$$drf(p)_i = v_l. \tag{7}$$

其中，v_l 是 v 在 i 处的有效版本。随后使用解引用变量 $drf(p)_i$ 更新被引用对象的值。具体使用以下公式：

$$\bigwedge_{o \in O} o_i = (p_j = \&o) \,?\, drf(p)_i : o_k \tag{8}$$

其中，p_j、o_k 分别是变量 p 和 o 在程序位置 i 处的相关版本。

换句话说，该公式表示：当 p 指向 o 时，对象 o 被赋予 v 的值；否则，它保持原值。如上所述，使用解引用变量可以提高表示的精确度。在以下情形中，这一点尤为明显：我们通过某个指针向一个抽象对象写入数据，随后又通过同一个指针读取数据，并且期间指针的值和被指向对象的值都没有改变。在这种情况下，我们可以重复使用同一个解引用变量，以确保读回刚刚写入的值。否则，由于我们是从一个表示多个具体对象的抽象对象中读取数据，因此未必能读到相同的值。

4.3 结构变换

*k*I*k*I 算法使用循环展开来细化程序的控制流结构。展开循环有几种可能的策略。我们在 Cbmc [35]中试验了一种展开策略：按照控制流逐个展开循环，并在每次展开后进行性质检查。采用这种策略时，每个循环都会逐步展开，直到所有路径都退出该循环，或者达到最大深度 k。如果在展开 j 时到达的每个状态都不满足循环条件，我们就可以判定该循环在展开 $j < k$ 时已完全展开。一个循环展开完成后，Cbmc 会继续处理下一个循环。重复这一过程，直到所有循环都展开完毕或发现漏洞。递归函数调用也采用类似的处理方式。

考虑图 10(a) 中的控制流图（CFG）。图 10(b) 展示了该 CFG 的展开策略。该程序有三个循环，循环头分别为 1、2 和 6（循环 2 嵌套在循环 1 内）。生成增量 BMC 公式 Φ(*k*)的符号执行（参见第 5.2 节）会遍历 CFG，并在每次遇到 CFG 中返回循环头的边（即所谓的回边）时停止。图 10(b) 展示了部分展开的 CFG 的三个快照，它们对应于增量 BMC 公式 Φ(*k*) 在 $k = 1, 2, m$时各个实例所考虑的程序部分。我们用 Φ(1) 表示截至首次遇到返回内层循环（循环头为 2）的回边时所得到的公式。公式 Φ(2) 在 Φ(1) 的基础上，将内层循环再展开一次。假设 m 是内层循环的最大展开次数，那么 Φ(*m*) 表示将公式扩展到以下情形：在外层循环（循环头为 1）的第一次迭代中，内层循环已展开至该最大次数。随后，公式 Φ($m + 1$) 将通过在外层循环的第二次迭代中首次展开内层循环（直到程序位置 4），对 Φ(*m*) 进行扩展。此过程会持续进行，直到遇到断言失败或到达程序末尾（8）。

我们意识到，这种方法的效率不高。因此，我们在 2LS [34, 4] 中采用了另一种方法。2LS 会将所有 循环展开 k 次，并逐步为所有 循环添加第 $k + 1$ 次展开，而不是只展开遇到的第一个循环，直到它完全展开。

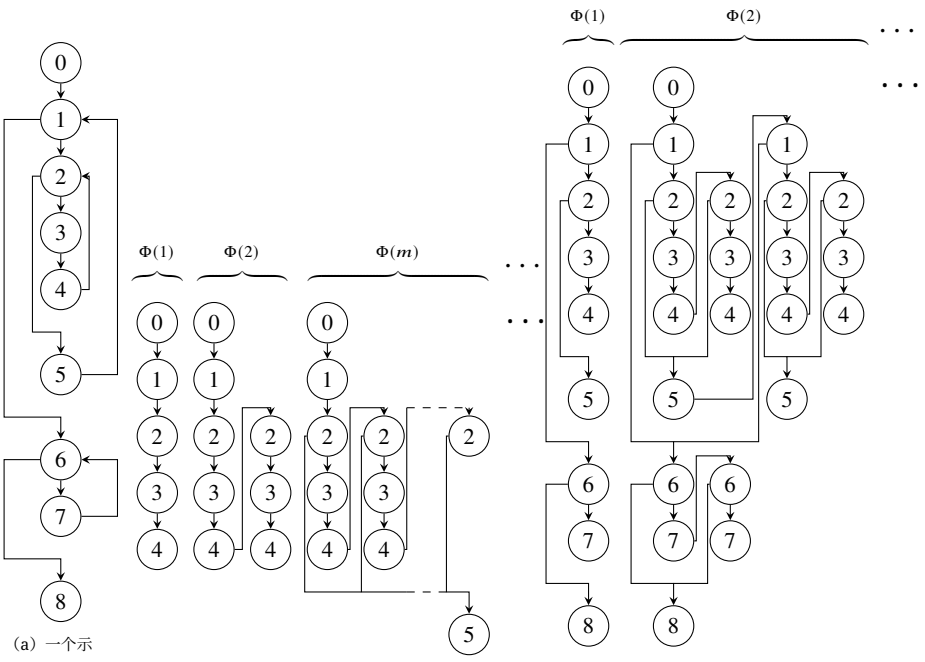
我们在图 10(c) 中展示了这种展开策略。图中显示了图 10(a) 中 CFG 的前两个部分展开，分别对应于 Φ(1) 和 Φ(2)。公式 Φ(1) 包含对循环 1、2 和 6 各进行一次展开（展开到回边之前，但不包括回边）。公式 Φ(2) 则为每个循环再增加一次展开。请注意，现在内层循环（循环头为 2）有两组、每组两次循环展开；外层循环（循环头为 1）的每次展开都对应其中一组。

从结构上看，这种展开策略与我们在非增量式 Cbmc 中使用的策略相同，即以 k 的固定值调用它时所用的策略。

2LS 会对静态单赋值形式进行循环展开，并为其补充一些关于循环结构和层次结构的元信息。我们执行的展开是增量式的，也就是说，当将 k 增加到 $k + 1$ 时，我们会逐步扩展静态单赋值形式。棘手之处在于合并循环出口（图 10(c) 中的节点 5、6 和 8）。我们必须考虑这样一种情况：“值合并自尚未添加的展开

”，这使得公式的构造不具单调性。实际上，我们必须有效地移除描述每个此类合并的析取式（它在静态单赋值形式中代表一个合取项），并用更大的析取式替换它，以反映通过额外展开为相关合并新增的输入分支。然而，SAT 求解器不支持移除公式中的部分内容。幸运的是，正如我们将在第 5.4 节中解释的那样，可以借助假设下求解解决这个问题。这是提升性能的关键要素，因为它使我们能够使用单个SAT 求解器实例运行增量有界模型检测，并最终运行整个 $kIkI$ 算法。

这种方法的一个缺点是，在符号执行期间通常可以进行许多用于简化公式的优化，例如常量传播，但这些优化只能在局部进行，甚至可能完全无法进行。



（a）一个示例，展示一个包含三个循环的程序。

(b) Cbmc 的增量展开策略 Cbmc 的。

(c) 增量展开策略 2LS。

图 10：增量展开策略。

5 k*I*k*I* 算法

正如我们在第 2 节中概述的，2LS 的核心是 k*I*k*I* 算法，我们将在本节中详细介绍该算法。该算法的总体工作流程如图 11 所示。最初， $k = 1$ 和 $\mathscr{T}$是一组可用作不变式的谓词，其中包含 $\top \in \mathscr{T}$（关于具体实现方式，详见第5节 5.1）。

在进行初始测试以检查是否有任何初始状态属于错误状态后，k*I*k*I* 会计算一个 k-归纳不变式，该不变式覆盖初始状态，并包含此前状态中不存在错误的假设。随后检查该不变式，判断它是否足以证明安全性。如果可能存在可达错误状态，则需要进行第二次检查，以确定错误是否能在 k 步内到达（真正的反例），或者它是否只是由不够强的不变式造成的潜在假象。后一种情况下，会递增 k，以便找到更强的（k-）不变式，然后算法继续循环。

图 11 还展示了增量有界模型检测、 k-归纳和经典过近似抽象解释的步骤，分别由红色点线、蓝色虚线和绿色虚线／点线框及箭头表示。k*I*k*I* 可以通过设置 $\mathscr{T} = \{\top\}$ 来模拟 k-归纳，并可通过对第一次 SAT 检查进行过近似来模拟增量有界模型检测。经典过近似抽象解释可以通过为抽象域 $\mathscr{A}$设置 $\mathscr{T} = \mathscr{A}$ 来模拟；如果第一次 SAT 检查找到一个模型，则以“未知”作为结果终止。

5.1 基于模板的谓词推断

kIkI 的一个关键阶段是生成 $KInv$，即一个 k-归纳不变式。或许最显而易见的方法是使用现成的抽象解释器。这种方法可行，但无法发挥 kIkI 的真正威力。在每次迭代中， kIkI 会将循环再展开一步（这可以改进抽象解释器给出的不变式），并添加假设，确保此前的循环展开不会产生错误。

直接使用求解器时，我们需要处理二阶逻辑的（存在量词）片段。由于目前还没有这类求解器，我们将问题转化为可通过反复调用一阶求解器来解决的问题。我们将目标限定为寻找形式为 $\mathrm{T}(\mathbf{x}, \boldsymbol{\delta})$ 的 k-归纳的 不变式 $KInv$，其中 T 是一个固定表达式，即所谓的模板，它定义在程序变量 $\mathbf{x}$ 和模板参数 $\boldsymbol{\delta}$ 之上。

$$\begin{aligned}\exists \boldsymbol{\delta}.\ \forall \mathbf{x}_0 \ldots \mathbf{x}_k.\ (&Init(\mathbf{x}_0) \wedge T[k-1] \Rightarrow \mathcal{T}[k](\boldsymbol{\delta})) \wedge \\ &(\mathcal{T}[k](\boldsymbol{\delta}) \wedge T[k] \Rightarrow \mathcal{T}(\mathbf{x}_k, \boldsymbol{\delta}))\end{aligned} \tag{9}$$

其中 $T[k]$ 是转移关系的第 k 次展开，而 $\mathrm{T}\ [k]$ 是除最后一个状态 $\mathbf{x}_k$之外，展开中所有状态的模板：

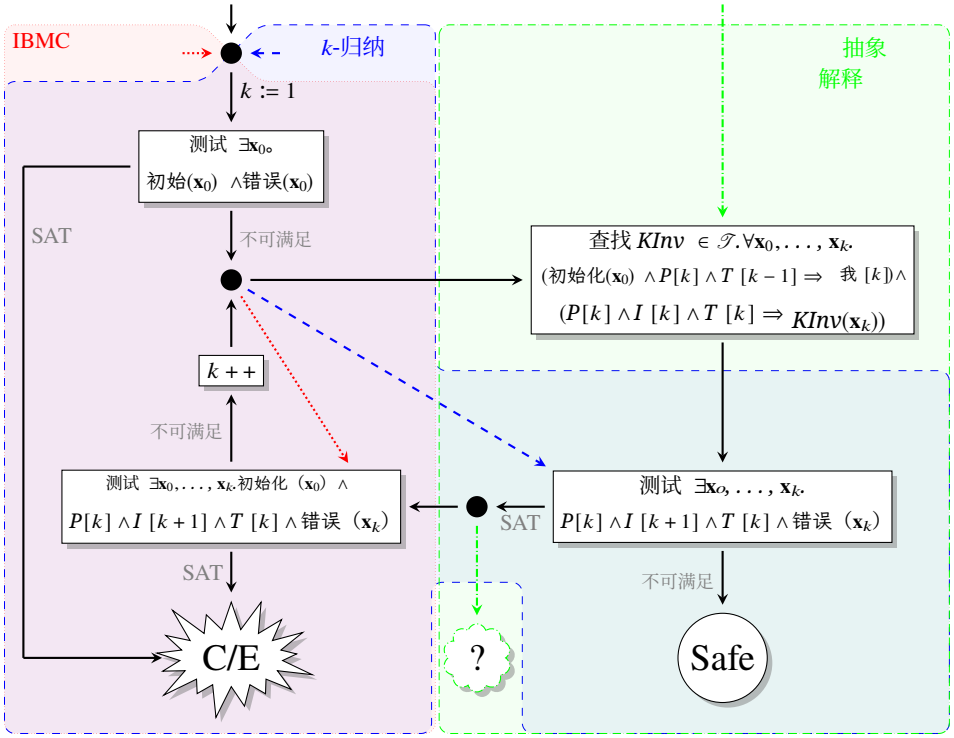


图 11： *k*I*k*I 算法。

$$T[k] = \bigwedge_{i\in[0,k-1]} Trans(\mathbf{x}_i, \mathbf{x}_{i+1}) \tag{10}$$

$$\mathcal{T}[k](\boldsymbol{\delta}) = \bigwedge_{i\in[0,k-1]} \mathcal{T}(\mathbf{x}_i, \boldsymbol{\delta}). \tag{11}$$

我们通过针对常数 **d** 的不同取值（作为参数 $\boldsymbol{\delta}$ 的值），迭代求解否定公式，特别是式 (9) 的第二个合取项，从而解决 ∃∀问题：

$$\exists \mathbf{x}_0 \ldots \mathbf{x}_k . \neg\big(\mathcal{T}[k](\mathbf{d}) \wedge T[k] \Rightarrow \mathcal{T}(\mathbf{x}_k, \mathbf{d})\big). \tag{12}$$

由此得到的公式可用无量词逻辑表示，并由 SMT 求解器高效求解。以此为构建模块，即可解决前面提到的∃∀ 问题。

从抽象解释的角度来看，**d** 是一个抽象值，也就是说，它表示（具体化为）满足公式 T (**s**,**d**) 的所有程序状态**s**的集合——这里，状态是**x**中变量值组成的向量。表示抽象域下确界 ⊥和上确界 ⊤的抽象值分别表示空集和整个状态空间：T (**s**, ⊥) ≡ *false* 以及 T (**s**, ⊤) ≡ *true* [4]。

形式化地说，具体化函数 γ 为： $\gamma(\mathbf{d}) = \{\mathbf{s} \mid \mathrm{T}(\mathbf{s},\mathbf{d}) \equiv true\}$。 在抽象函数中，为了获得表示给定具体程序状态$\mathbf{s}$的最精确抽象值，我们令 $\alpha(\mathbf{s}) = \{\min(\mathbf{d}) \mid \mathrm{T}(\mathbf{s}, \mathbf{d}) \equiv true\}$。如果抽象域构成完备格，则可以保证存在这样的最小值$\mathbf{d}$。

不变式推断算法以 $\mathbf{d} = \bot$ 的初始值为起点，并使用 SMT 求解器迭代求解 (12)。如果公式不可满足，则已找到不变式；否则，求解器会返回一个可满足性模型 $\mathbf{d}'$。该模型表示当前模板实例化不能作为不变式的反例。随后，通过使用特定于域的连接运算符 [4]，将模板参数 $\mathbf{d}$ 的当前值与获得的可满足性模型相结合，从而更新该参数。

例如，假设我们有一个程序，其中包含一个循环，该循环将变量 x 的值从 0 计数到 10，并且我们有一个模板 $x \le d$。假设参数 d 的当前值为 3，而我们得到一个新模型 $d' = 4$。然后，我们通过计算 $d \sqcup d' = \max(d, d')$ 将参数更新为 4，因为 max 是用于跟踪数值上界的域的连接运算符。

在 2LS 中，我们使用单个模板来计算被分析程序的所有不变式。因此，模板通常由多个部分组成，每个部分描述一组程序变量的不变式。对此，我们认为模板 $\mathrm{T}(\mathbf{x}, \boldsymbol{\delta})$ 由所谓的 模板行 $\mathrm{T}_r(\mathbf{x}_r, \delta_r)$ 组成，每一行 r 描述变量子集 $\mathbf{x}_r$ 的不变式 $\mathbf{x}$，并具有自己的行参数 δ_r。总体不变式由各模板行与相应行参数的计算值组合而成。组合的类型（例如，可以是简单的合取）由各个域定义。

5.1.1 带守卫模板

由于我们使用静态单赋值形式，而非控制流图，因此不能直接使用模板。我们改用 带守卫模板。如上所述，模板由多个模板行组成，每一行描述一组程序变量的不变式。在带守卫模板中，每一行 r的形式为 $G_r(\mathbf{x}_r) \Rightarrow \widehat{\mathrm{T}}_r(\mathbf{x}_r, \delta_r)$，其中 r^{th}行 $\widehat{\mathrm{T}}_r$属于基础模板域（例如模板多面体）。 G_r 是与定义变量 $\mathbf{x}_r$ 相关联的 SSA 守卫的合取，这些变量出现在 T_r 中。由于我们旨在推断循环不变量， $G_r(\mathbf{x}_r)$ 表示与循环头处出现的变量 $\mathbf{x}_r$ 相关联的守卫。因此，不同循环的模板行具有不同的守卫条件。

我们用图 9 中的示例程序来说明上述内容，采用一个带守卫的区间模板。该模板的形式为：

$$\mathcal{T}(x_6^{lb}, (\delta_1, \delta_2)) = \begin{array}{l} g_3 \wedge g_6^{ls} \Rightarrow x_6^{lb} \le \delta_1 \wedge \\ g_3 \wedge g_6^{ls} \Rightarrow -x_6^{lb} \le \delta_2. \end{array} \tag{13}$$

这里，g_3 和 g_6^{ls} 用于限定 x_6^{lb} 的定义——g_3 表示循环头可达这一事实，而 g_6^{ls} 表示 x_6^{lb} 被选作 x_3^{phi} 的值。

5.1.2 求解 ∃∀ 问题

如上所述，要推断不变式，必须求解一个 ∃∀问题，以确定模板参数 $\boldsymbol{\delta}$ 的值。

模型枚举

使用 SMT 求解器求解该问题(12)的一种广为人知的方法是：针对不同的抽象值$\mathbf{d}$，反复检查公式的可满足性$\mathbf{d} = \bot$（从下确界开始）：

$$\mathcal{T}[k](\mathbf{d}) \wedge T[k] \wedge \neg\mathcal{T}(\mathbf{x}_k, \mathbf{d}). \tag{14}$$

如果公式不可满足，那么我们就找到了一个不变式；否则，就将求解器返回的模型与先前的抽象值$\mathbf{d}$ 合并，并使用合并后得到的新$\mathbf{d}$ 值重复这一过程。

这种方法相当于在抽象格上执行经典的 Kleene 迭代，直到收敛。由于我们的抽象域是有限的，因此可以保证收敛。然而，虽然这种方法对于某些抽象域（特别是可能状态数量较少的抽象域）或许足够，但对于其他抽象域来说，在实践中却无法使用。例如，处理整数变量时，格的高度极大；即使对于一个循环中递增无约束 64 位变量的程序，朴素算法也无法在人的一生中终止。因此，对于某些抽象域，我们将采用优化方法（例如，对于某些数值域，参见第6.1节）。

一般来说，每个域都可能有一种求解该 ∃∀问题的特定方法。我们将这些方法称为域策略迭代。

通用域策略迭代

尽管每个域都可以实现自己的策略迭代算法，但我们观察到所有算法在某种程度上都相似。这与抽象域模板通常由多个模板行组成这一事实有关，正如本节前文所述。

为此，我们开发了一种通用策略迭代算法 [26]，该算法以模板形式的抽象域为参数。该算法如图 12 所示。

对于给定的抽象域，算法会反复求解（14）。如果公式可满足，那么对于每个模板行 r，算法都会获取满足性模型，其中包含 r 所描述变量的取值。获得的模型（即相关变量的取值）用于实例化模板行公式（第 8 行），并将模板参数的对应值与该行参数的先前值合并。

1: **d** $\leftarrow\perp$2: **当** T $[k]$**(d)** 不是不变式时 **执行**3: 求解器 $\leftarrow$T $[k]$**(d)** 4: 求解器 $\leftarrow \neg$T **(x**$_k$**, d)** 5: **如果** solver.solve() = SAT **则**6: **对于** T$_r$**(x**$_r$**,** d_r**)** $\in$T **(x, d) 执行** 7: $m_r \leftarrow$solver.model(**x**$_r$) 8: 找到 d_m 使得 T$_r(m_r, d_m)$ 成立 9: $d_r \leftarrow d_r \sqcup d_m$ 10: **结束循环**11: **结束条件判断**12: **结束循环**

图 12：用于求解 $\exists\forall$ 问题的通用策略迭代算法。

由于 2LS 使用增量求解，我们假设转移关系（由静态单赋值形式表示）已属于求解器子句集的一部分。此外，每次迭代后都会移除第 3 行和第 4 行加入求解器子句集的公式。这样可以确保该方法的效率，因为每次只需重新求解新增的公式。

然而，正如我们在上一节中所提到的，为了确保算法能够扩展，可能需要进行优化。在这种情况下，第 7 至 9 行将替换为用于确定模板行参数取值的优化方法。我们将在第 6.1 节中给出此类优化技术的示例。

5.2 增量有界模型检测

有界模型检测（BMC） [2] 通过选取一个展开界限 k 并求解方程来进行反驳

$$\exists \mathbf{x}_0, \ldots, \mathbf{x}_k . Init(\mathbf{x}_0) \wedge T[k] \wedge \neg P[k+1] \tag{15}$$

其中，$T[k]$ 是式 (10) 定义的展开后的转移关系，而$P[k]$ 是一个谓词，表示 k 状态不存在错误：

$$P[k] = \bigwedge_{i \in [0,k-1]} \neg Err(\mathbf{x}_i). \tag{16}$$

式 (15) 的模型对应于长度为 $n \leqslant k$ 的某个具体反例。展开界限给出了可达状态集合的下近似，因此可能无法找到需要经过大量转移步骤的反例。在实践中，BMC 的效果很好，因为该公式是存在量化的，因而属于 SAT 和 SMT 求解器能够高效处理的片段。

增量有界模型检测是第5节中介绍的 kIkI 算法的核心组件之一。它对应于图 11 中的红色部分。增量有界模型检-

测（IBMC）（例如 [13]）通过随着界的增加反复执行 BMC 检查（通常通过增量使用求解器进行优化），避免需要固定界。如果界从 0 开始（即检查 $\exists x_0$。初始状态 $(\mathbf{x}_0)$ ∧ 错误 $(\mathbf{x}_0)$ ），并且每一步都增加 1 （这是常见用例），那么可以假设先前状态中不存在错误，从而得到一个更简单的测试：

$$\exists \mathbf{x}_0, \ldots, \mathbf{x}_k . Init(\mathbf{x}_0) \wedge T[k] \wedge P[k] \wedge Err(\mathbf{x}_k). \tag{17}$$

在第 5.4 节中，我们将讨论如何使用增量SAT求解，高效地实现增量有界模型检测。

5.3 增量 *k*-归纳

增量 *k*-归纳 [36] 是图 11 中 *k*I*k*I 算法的蓝色部分。它可以视为 IBMC 的扩展，既能证明系统安全性，也能生成反例。它使用 *k*-归纳不变量，这些谓词具有以下性质：

$$\forall \mathbf{x}_0 \ldots \mathbf{x}_k . I[k] \wedge T[k] \Rightarrow KInv(\mathbf{x}_k) \tag{18}$$

其中

$$I[k] = \bigwedge_{i \in [0,k-1]} KInv(\mathbf{x}_i).$$

k-归纳不变量具有以下有用性质：

- 任何归纳不变式都是 1-归纳不变式，反之亦然。
- 任何 *k*-归纳不变式都是 $(k+1)$-归纳不变式。
- 当且仅当存在满足以下条件的 *k*-归纳不变式 *KInv* 时，有限系统才是安全的：

$$\begin{aligned} \forall \mathbf{x}_0 \ldots \mathbf{x}_k . \; & (Init(\mathbf{x}_0) \wedge T[k] \Rightarrow I[k]) \wedge \\ & (I[k] \wedge T[k] \Rightarrow KInv(\mathbf{x}_k)) \wedge \\ & (KInv(\mathbf{x}_k) \Rightarrow \neg Err(\mathbf{x}_k)) . \end{aligned} \tag{19}$$

证明存在 *k*-归纳不变式，就足以证明存在归纳不变式，但这并不意味着 *k*-归纳不变式本身就是归纳不变式。通常，相应的归纳不变式要复杂得多。因此， *k*-归纳可以看作是在不变式生成与检查之间的一种权衡，因为它通过使用更复杂的性质检查，尽可能地利用了更简单的不变式。

然而，寻找候选的 *k*-归纳不变式仍然很困难，因此实现通常会使用 ¬ *Err*(**x**) 作为候选项。与 IBMC 类似，可以使用线性递增的 *k*，通过假设先前状态中没有错误来简化表达式：

$$\begin{aligned} \exists \mathbf{x}_0, \ldots, \mathbf{x}_k . \; & (Init(\mathbf{x}_0) \wedge T[k] \wedge P[k] \wedge Err(\mathbf{x}_k)) \vee \\ & (T[k] \wedge P[k] \wedge Err(\mathbf{x}_k)). \end{aligned} \tag{20}$$

析取项第一部分的一个模型是具体反例（k-归纳涵盖了 IBMC）；如果整个公式没有模型，那么 $\neg Err(\mathbf{x})$ 就是一个 k-归纳不变式，且系统是安全的。

5.4 增量 SAT 求解

2LS 需要增量式后端求解器。增量 SAT 求解的最初构想可追溯到 20 世纪 90 年代 [19, 37, 20]。问题在于，如何在求解一系列相似 SAT 问题时，重用求解先前实例所付出的工作。 [38, 39] 的作者确定了可重用学习子句的条件，但这需要耗费大量资源进行簿记，部分抵消了增量性的优势。显然，如果对问题的合取范式表示进行修改，使其单调增长，那么增量 SAT 求解就很容易实现。这意味着，如果我们要用合取范式公式 $\Phi(k)$ 求解一系列约束逐渐增多的 SAT 问题，其中 $k \geq 0$，那么 $\Phi(k)$ 必须在 k 中单调增长，即 $\Phi(k+1) = \Phi(k) \wedge \varphi(k)$，以得到合取范式公式 $\varphi(k)$。从 $\Phi(k)$ 中删除子句则更为棘手，因为求解过程中学习到的某些子句不再由新实例蕴含，因此也需要删除。这需要额外的求解器功能，例如在假设条件下 求解 [13],，这是增量 SAT 求解中最常用的方法：假设是对变量的临时赋值，仅在 SAT 求解器的某一次调用中有效。我们将看到，增量 BMC 需要一系列非单调公式。

另一种方法是使用 SMT 求解器。SMT 求解器提供了一种接口，可以像操作栈一样压入和弹出子句。压入会添加子句，弹出则会从公式中移除子句。这使用户能够直观地修改公式，但效率取决于底层压入和弹出操作的实现。例如，在 [17], 中观察到，一些 SMT 求解器（如 Z3）并未针对增量使用进行优化，因此其增量求解表现不如非增量求解。

因此，由于 SMT 求解器对增量求解的支持仍落后于 SAT 求解器，我们采用 SAT 求解。CPROVER 框架 [8] 本身实现了强大的位向量判定过程，并使用 MiniSAT2 [12] 或 Glucose[4] 等 SAT 求解器作为后端求解器。对于 SAT 求解器，基于假设求解是主流方法。

增量有界模型检测中的公式构造

按照 [13]（针对有限状态机的表述）中的构造，增量有界模型检测可以表示为一系列 SAT 问题 $\Phi(k)$，我们需要求解这些问题：

[4] http://www.labri.fr/perso/lsimon/glucose/#glucose-4.0

$$\Phi(0) := \text{初始状态}(\mathbf{x}_0) \wedge (\alpha_0 \Rightarrow \text{错误状态}(\mathbf{x}_0)),\ \text{并假设}\ \alpha_0,\ \Phi(k+1) := \Phi(k) \wedge \neg\,\text{错误状态}\,(\mathbf{x}_k) \wedge \text{转移关系}\,(\mathbf{x}_k, \mathbf{x}_{k+1}) \wedge \neg\alpha_k \wedge (\alpha_{k+1} \Rightarrow \text{错误状态}\,(\mathbf{x}_{k+1})),\ \text{并假设}\ \alpha_{k+1} \tag{21}$$

其中 α_k 是布尔变量。现在来看看为什么需要它们。考虑一个程序，其中变量 x 从 0 计数到 10。循环内有一个断言，预期 $x \leq 1$ 应成立。我们从初始公式开始（省略布尔变量 α_k）：

$$\Phi(0) := (x_0{=}0) \wedge (x_0{<}10) \wedge (x_1{=}x_0 + 1) \wedge \neg(x_1{\leq}1).$$

该公式不可满足。现在我们（天真地）将下一次循环迭代加入公式：

$$\Phi(1) := (x_0{=}0) \wedge (x_0{<}10) \wedge (x_1{=}x_0{+}1) \wedge \neg(x_1{\leq}1) \wedge (x_1{<}10) \wedge (x_2{=}x_1{+}1) \wedge \neg(x_2{\leq}1).$$

这仍然不可满足，尽管我们预期它可满足。显然，原因在于我们向一个已经不可满足的公式中添加了更多合取项。因此，为了保证可靠，我们必须移除上一轮迭代中的 $x_1 > 1$，这使得公式构造变成非单调的。

不过，如上所述，我们不能直接从求解器中移除公式的部分内容，但可以借助假设下求解来解决这个问题。在迭代 k中，假设 α_k 为真，而在迭代 $k' > k$ 中则假设它为假。这样一来，在迭代 k' 中，子公式（ $\alpha_k \Rightarrow$ 错误 $(\mathbf{x}_k)$）就会自动成立。因此，它不会影响 $\Phi(k')$的可满足性，这就模拟了删除该子公式的效果。[5]

增量有界模型检测的优势

非增量式有界模型检测必须在每次展开 k 时额外执行符号执行工作。正如 [35], 中的研究所示，有人可能会认为，消除这部分开销是增量有界模型检测速度更快的主要原因。然而，与生成并求解 SAT 公式相比，增量方法和非增量方法在符号执行上的开销相近：在 [35], 27 的嵌入式软件基准测试集中，增量方法耗费的时间中有 % 用于求解 SAT 公式（582 秒，共 2,151 秒），而非增量方法耗费的时间中有 28% 用于此项工作（3,317 秒，共 11,811 秒）。我们在图13的柱状图中展示了这一现象。该图绘出了增量式和非增量式有界模型检测的总运行时间，其中浅灰色表示生成和求解 SAT 公式所用的时间，深灰色表示开销。不出所料，如图13中的第三根柱所示，为非增量方法中最大的 k 求解实例所需的时间

[5] 经过大量迭代 k 后，这些显然满足的子公式可能会作为“垃圾数据”在公式中不断累积，从而减慢其求解速度。在适当的时机重新启动求解器，是解决这一问题的常见方法 [6]。

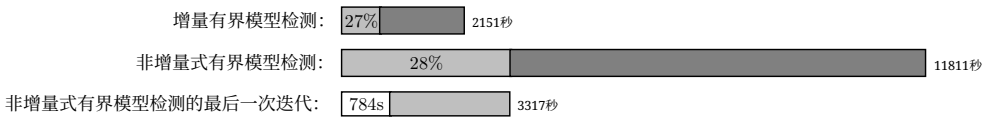


图13：增量有界模型检测的优势：求解时间与总体运行时间的对比（解释见正文）。

增量方法（白色部分）耗时相当可观（约 24%），与求解第 1 至 k 次迭代的 SAT 公式所用总时间（白色+灰色部分；3,317 秒中的 784 秒）相比更是如此。

增量有界模型检测带来加速的一种解释，可能在于两种方法所发出查询的规模。每次求解器调用的平均子句数从非增量方法的 1,367k 个子句减半至增量方法的 709k 个子句。类似地，与非增量方法相比，增量方法最后一次调用中的变量平均数不到其三分之一，分别为 217k 和 746k。这意味着，SAT 求解器能够在预处理中更高效地简化增量方法中的公式。

文献 [35] 在 Cbmc 中尝试将增量切片算法与增量展开相结合；然而，这并未在 2LS 中实现。文献 [30]在 2LS 中开发了一种 SMT 求解算法（可在 2LS 的一个原型分支中使用），该算法使用模板多面体而非布尔文字，以便基于抽象冲突驱动学习的概念执行理论层传播 [11]。

6 已实现的分析

2LS 支持对各种程序特性进行分析，例如断言的可达性、终止性或内存安全。对于其中大多数分析，2LS 会引入用于不变式推断的抽象域 [10]，而不变式推断是 kIk 算法的步骤之一。

这些域包括数值域，例如模板多面体（见第 6.1 节）、等式与不等式、用于排名函数的域（见第 6.2 节）和用于回归集的域（见第 6.3 节），以及用于描述数据结构形状的域（见第 6.4 节）。

本节将介绍最重要的受支持抽象域，以及支持验证各种程序性质的其他概念。

6.1 模板多面体抽象域

模板多面体 [32]是一类用于数值变量的模板，其形式为 $\mathbf{T} = (\mathbf{Ax} \leq \boldsymbol{\delta})$，其中 $\mathbf{A}$ 是一个系数固定的矩阵。这类模板的子类包括 区间，它要求对每个变量 x_i施加约束 $\begin{pmatrix} 1 \\ -1 \end{pmatrix} x_i \leq \begin{pmatrix} \delta_{i1} \\ \delta_{i2} \end{pmatrix}$；区域 （差值）以及八边形 [28]。模板的 r^{th} 行 是由矩阵 $\mathbf{A}$ 的 r^{th} 行所生成的约束。

在我们的模板表达式中，变量 $\mathbf{x}$ 是位向量，表示有符号或无符号整数。这些变量可以混合用于模板约束中。应用类型提升规则，将表达式类型的位宽扩展，以避免模板表达式中的算术下溢和溢出。 $\top$ 对应提升后类型中的相应最大值，而 $\bot$ 必须编码为一个特殊符号。

对 ∃∀问题的优化求解

为了高效地解决参数综合问题，我们需要一种收敛加速方法，使计算开销不受状态数和循环迭代次数的影响。为此，我们采用一种受最大策略迭代方法所用编码启发的技术 [15, 14, 29]。这些方法将模板多面体上的不变式推断问题表述为一个析取线性优化问题，并通过在模板多面体格上向上迭代来逐步求解：利用 SMT 求解，选出一个解能够扩展当前不变式候选的合取子系统（“策略”）。随后使用线性规划求解器求解该子系统；一旦找到归纳不变式，过程便终止。[6]

只有在域为凸集且参数值相对于具体化有序并满足单调性时，才能使用上述方法。例如，模板多面体 $\mathbf{Ax} \leq \mathbf{d}$ （其中 $\mathbf{d}$ 是参数）符合这一条件，但 $\mathbf{A}$ 是参数的模板多面体则不符合。如果转移关系中的操作满足某些性质（例如条件谓词的单调性），那么得到的结果就是最小不动

[6] 直观地说，其工作原理如下：例如，假设数值变量上界的域（不过，这种方法也适用于满足下文所述条件的任意域）。那么，程序位置处的不变式上界，就是所有入边转移对该程序位置所贡献的值中的最大值。将其写成一个关于线性不等式组的优化问题时，入边转移构成析取。考虑多个程序位置后，我们便得到这些析取的合取。因此，我们得到一个合取范式的线性系统。通过找出每个子句中那些满足初始状态的析取项（必须存在这样的析取项，否则系统不可行），即可求解该系统。然后，我们可以选取这些析取项并将它们合取起来（作为“策略”），从而得到一个（合取的）线性系统（称为合取子系统），并使用线性规划求解器求解。所得到的上界仅对选中的析取项有效。如果它尚未构成不变式，那么必定存在可以替换某些析取项、从而改进该上界的析取项。重复这一过程，保证最终会得到最小不变式。

点，也就是与上述朴素模型枚举所返回的相同结果，但平均速度要快得多。

我们将此方法调整为适用于具有位向量变量和守卫模板的情形。由于我们处理的是有限域（位向量），因此可以使用二分搜索 作为优化方法，而不是使用线性规划求解器。

算法如下：我们首先检查当前抽象值$\mathbf{d}$ （从$\mathbf{d} = \bot$开始）是否为归纳的（见（14））。如果是，我们就找到了一个不变式；否则，仍有一些模板行 R 的值尚未归纳。此外，归纳性检查得到的归纳反例会告诉我们哪些界需要改进。我们构造系统

$$\bigwedge_{i\in[0,k-1]} \left\{ \begin{array}{l} \bigwedge_{r\notin R}\ G_r(\mathbf{x}_i) \Rightarrow (e_r(\mathbf{x}_i) \leq d_r) \\ \wedge\ \bigwedge_{r\in R}\ G_r(\mathbf{x}_i) \Rightarrow (e_r(\mathbf{x}_i) \leq \delta_r) \end{array} \right\} \wedge T[k] \wedge \bigwedge_{r\in R} (G_r(\mathbf{x}_k) \wedge \delta_r \leq e_r(\mathbf{x}_k)) \tag{22}$$

其中e_r 是模板中对应于 r^{th} 行的不等式左侧。然后，我们开始在此系统上对 $\sum_{r\in R}\delta_r$ 的最小值进行二分搜索。 $\sum_{r\in R}\delta_r$ 的初始界如下：

- 下界l 为 $\sum_{r\in R} d'_r$，其中 d'_r 是上述归纳性检查（14）的模型中e_r （$\mathbf{x}_k$）的值；
- 上界 u 为 $\sum_{r\in R} max_value(r)$，其中 max_value 返回$e_r(\mathbf{x}_k)$可能具有的最大值（取决于变量类型）。

二分搜索通过在假设 $\sum_{r\in R}\delta_r \geq m$ where $m = median(l, u)$ 的条件下，反复检查 (22) 是否可满足来进行。如果可满足，则设置$l := m$; 否则设置 $u := m$ ，并重复此过程，直到$l = u$。最后一次可满足查询中 δ_r 的值将赋给 d_r ，以获得新的抽象值。随后，通过测试$\mathbf{d}$ 是否为归纳的 (14)，重复该过程。请注意，该算法使用与策略迭代相似的界优化编码，但其迭代次数可能高于策略迭代。我们有意作出这一选择，以保持生成的 SMT公式规模较小，代价是迭代次数可能增加。同一行 r 可以从 R 中移除，并在之后的迭代中再次改进。不过，由于每次迭代都会取得进展，该算法会在有限次迭代后终止。

我们通过图 9 中的示例来说明二分搜索算法。我们使用区间抽象域来计算变量 x 在循环结束时取值的区间。令 $k = 1$，因此令$\mathbf{x}_0 = (x_6^{lb})$ 和 $\mathbf{x}_1 = (x_5)$。[7]对应的带守卫模板 T 具有式 (13) 中定义的形式。该模板包含两个定义如下的模板行：

$$G_0(x_6^{lb}) \Rightarrow x_6^{lb} \leq \delta_0 \tag{23}$$

$$G_1(x_6^{lb}) \Rightarrow -x_6^{lb} \leq \delta_1 \tag{24}$$

[7] 变量 $\mathbf{x}_0$ 适用于循环迭代次数为零的情况，因此此处将无约束的 x_6^{lb} 作为 x 的值。

其中 $G_0(x_6^{lb}) = G_1(x_6^{lb}) = g_3 \wedge g_6^{ls}$。我们还定义了变量$\mathbf{x}_1$ 的模板行守卫条件（对应于执行一次转移关系后程序变量的取值），如下所示：$G_0'(x_5) = G_1'(x_5) = g_5 = x_3^{phi} < 10$。

最初，$\mathbf{d} = \perp$，我们求解 (14)。由于将$\mathbf{d} = \perp$ 用作参数值会根据定义使模板的实例*false*，因此该不变式无法归纳得到，我们得到一个可满足性模型。两个模板行都不是归纳的，因此我们用 $R = \{0,1\}$ 实例化 (22)。得到的公式系统如下（其中*Trans*($\mathbf{x}_0$,$\mathbf{x}_1$是图 9(b) 中所示约束的合取）：

$$\begin{aligned} G_0(x_6^{lb}) \Rightarrow x_6^{lb} \leq \delta_0 \wedge \\ G_1(x_6^{lb}) \Rightarrow -x_6^{lb} \leq \delta_1 \wedge \\ Trans(\mathbf{x}_0, \mathbf{x}_1) \wedge \\ G_0'(x_5) \wedge \delta_0 \leq x_5 \wedge \\ G_1'(x_5) \wedge \delta_1 \leq -x_5. \end{aligned} \tag{25}$$

我们从$l = -2^{31}$ 和 $u = 2^{31} - 1$ 开始，对该系统应用二分搜索。得到 $\delta_0 = 1$ 和 $\delta_1 = -1$。现在，模板行 1（$-x_6^{lb} \leq -1$）是归纳的，但模板行0不是归纳的。我们根据 (22) 用 $R = \{0\}$ 构造一个新系统：

$$\begin{aligned} G_0(x_6^{lb}) \Rightarrow x_6^{lb} \leq \delta_0 \wedge \\ G_1(x_6^{lb}) \Rightarrow -x_6^{lb} \leq -1 \wedge \\ Trans(\mathbf{x}_0, \mathbf{x}_1) \wedge \\ G_0'(x_5) \wedge (\delta_0 \leq x_5). \end{aligned} \tag{26}$$

使用二分搜索，我们得到 $\delta_0 = 10$。由此可知，模板行 0 是归纳的，因此我们找到了一个不变式，即：

$$\begin{aligned} g_3 \wedge g_6^{ls} \Rightarrow x_6^{lb} \leq 10 \wedge \\ g_3 \wedge g_6^{ls} \Rightarrow -x_6^{lb} \leq -1. \end{aligned} \tag{27}$$

位向量宽度扩展

在大多数体系结构中，整数发生上溢或下溢时会回绕。因此，为了保证结果的可靠性，我们必须留意模板中的算术运算。

让我们通过综合一个秩函数，来分析以下函数的终止性。

```
void f(){ 有符号字符
型 x; while(1) x++; }
```

排序函数综合旨在计算模板参数l的一个值，使得$l(x-x') > 0$对转移关系 $x'=x+1$ 和计算出的不变式*true*下的所有 x, x' 均成立。

因此，假设l的当前值为 -1，需要求解的约束为

$$true \wedge x'=x+1 \wedge \neg(-1 \cdot (x-x')>0),$$

这等价于 $\neg(-1(x-(x+1))>0)$。对于数学整数，该约束可满足；但对于有符号位向量，由于溢出，该约束不可满足。对于 $x=127$，如果有符号字符型是 8 位有符号整数，则会发生如下溢出： $x+1=-128$ 。因此， $\neg(-1\cdot(127-(-128))>0)$ 变为 $\neg(1>0)$，这使得该约束不可满足，我们会错误地得出 $-x$ 是一个秩函数的结论；但在有符号位向量语义下，这一结论并不成立。然而，如果将位向量宽度扩展到 $k=9$ ，使模板中的算术运算不会溢出，那么 $\neg(-1 \cdot ((\text{有符号}9)127-(\text{有符号}9)(-128)) > 0)$ 的计算结果为 $\neg((-1\cdot 255) > 0)$，其中 有符号_k 是向 k 位有符号整数的类型转换。现在， $x=127$ 是一个模型，它表明 $-x$ 不是有效的秩函数。

为确保可靠性，我们将有符号和无符号整数操作数的位宽扩展为足以容纳运算结果、不会发生上溢或下溢的整数。例如，加法增加一位，乘法则将位数加倍后再增加一位。所需的最大位宽取决于模板的结构。由于我们的模板只包含有限次运算，因此最大位宽也是有限的。

浮点数无需扩展，因为它们溢出（相应地，下溢）时会变为无穷大（相应地，负无穷大），这不会影响可靠性。

6.2 终止分析

为了推理终止性，我们需要了解秩函数和终止前置条件的概念。

定义 1（秩函数）
过程（秩函数 *Init*，*Trans*）的一个函数 r ，满足

$$\begin{aligned}&\exists \Delta > 0, Inv : \forall \mathbf{x}, \mathbf{x}' : \\ &\quad (Init(\mathbf{x}) \implies Inv(\mathbf{x})) \\ &\wedge\ (Inv(\mathbf{x}) \wedge Trans(\mathbf{x}, \mathbf{x}') \implies Inv(\mathbf{x}') \wedge r(\mathbf{x})-r(\mathbf{x}')>\Delta \wedge r(\mathbf{x})>0)\,.\end{aligned}$$

因此， r 是一个从程序状态集合到良基域的函数，例如$\mathbb{R}^{\geq 0}$。

我们用 $RR(\mathbf{x},\mathbf{x}')$ 表示 r 上构成终止性论证的约束，即，对于单体排序函数， $r(\mathbf{x}) - r(\mathbf{x}') > \Delta \wedge r(\mathbf{x}) > 0$。对于一个过程，存在秩函数即可保证其普遍终止性。

单体排序函数是完备的，即如果程序终止，总能通过单体方式证明其终止性。然而，实践中更倾向于使用线性排序函数的组合，例如线性字典序函数。这是因为单体线性 排序函数的表达能力不足，而且现有的 SMT 求解器难以处理非线性 理论；相比之下，它们处理线性情况的效率要高得多。

定义 2（字典序排序函数）

过程的字典序排序函数 R 是一个由表达式组成的 n 元组（R_n、R_{n-1}，……，R_1），满足

$$
\begin{aligned}
&\exists \Delta > 0, Inv : \forall \mathbf{x}, \mathbf{x}' : \\
&\quad (Init(\mathbf{x}) \implies Inv(\mathbf{x})) \\
&\wedge \big(Inv(\mathbf{x}) \wedge Trans(\mathbf{x}, \mathbf{x}') \implies Inv(\mathbf{x}') \wedge \exists i \in [1, n] : \\
&\qquad R_i(\mathbf{x}) > 0 && \text{（有界）} \\
&\qquad \wedge\ R_i(\mathbf{x}) - R_i(\mathbf{x}') > \Delta && \text{（递减）} \\
&\qquad \wedge\ \forall j > i : R_j(\mathbf{x}) - R_j(\mathbf{x}') \geq 0\ \big) && \text{（非递增）。}
\end{aligned}
$$

注意，这是定义 1 的一个特殊情况。特别是，$\Delta > 0$的存在以及有界条件保证了 $>$ 是一个良基关系。

在将字典序排序函数的要求编码为约束之前，我们需要对其进行优化，以充分利用位向量语义。由于位向量是有界的，因此有界条件显然成立，可以省略。此外，位向量是离散的，因此我们可以将递减条件替换为$R_i(\mathbf{x}) - R_i(\mathbf{x}') > 0$。当且仅当（$R_n$，$R_{n-1}$，……，$R_1$）是一个由位向量构成、包含$n$个分量的字典序排序函数时，以下公式$LR^n$成立。

$$
LR^n(\mathbf{x}, \mathbf{x}') = \bigvee_{i=1}^{n} \left(R_i(\mathbf{x}) - R_i(\mathbf{x}') > 0 \ \wedge \bigwedge_{j=i+1}^{n} (R_j(\mathbf{x}) - R_j(\mathbf{x}') \geq 0) \right)
$$

假设给定过程 f的转移关系$Trans(\mathbf{x}, \mathbf{x}')$。过程 f可能由多个循环组成，每个循环都关联有守卫条件 g （以及 g'），分别表示循环头（以及循环体末尾）的可达性；参见第4.2节。也就是说，假设 f有 k 个循环，且n_i 表示循环 i的字典序分量数，那么用于证明 f终止的终止性论证形式如下：

$$
RR^{\mathbf{n}}(\mathbf{x}, \mathbf{x}') = \bigwedge_{i=1}^{k} g_i \wedge g_i' \implies LR_i^{n_i}(\mathbf{x}, \mathbf{x}')
$$

虽然针对数学整数的秩函数技术会使用例如 Farkas 引理的方法，但这不适用于位向量运算。因此，我们采用综合方法，并将其从单体秩函数扩展到字典序排序函数。

我们考虑由该模板生成的字典序排序函数类，其中 $R_i(\mathbf{x})$ 是 $l_i\mathbf{x}$ 的乘积，l_i 是模板参数组成的行向量。我们将循环 i 对应的约束记为 $\mathrm{LR}_i^{n_i}(\mathbf{x}, \mathbf{x}', L_i^{n_i})$，其中 $L_i^{n_i}$ 是向量 $(l_i^1, \ldots, l_i^{n_i})$。整个过程的秩函数约束为 $\mathrm{RR}(\mathbf{x}, \mathbf{x}', \mathbf{L^n})$，其中 $\mathbf{L^n}$ 是向量 $(L_1^{n_1}, \ldots, L_k^{n_k})$。

综合以上内容，我们得到如下将秩函数综合归约为模板上的一阶量词消去问题的过程：

$$\exists \mathbf{L^n} : \forall \mathbf{x}, \mathbf{x}' : \mathit{Inv}(\mathbf{x}) \wedge \mathit{Trans}(\mathbf{x}, \mathbf{x}') \Longrightarrow \mathcal{RR}(\mathbf{x}, \mathbf{x}', \mathbf{L^n})$$

参数 $L_i^{n_i}$ 是由位向量构成的向量，并扩展了特殊值 $\top$，以补全秩约束格 $\mathrm{LR}_i^{n_i}$。我们定义 $\mathrm{LR}_i^{n_i}(\mathbf{x}, \mathbf{x}', \top) \equiv$ 真，表示尚未为给定模板找到秩函数（“未知”）。我们用 $\bot$ 表示底元素的等价类，对于这些元素，$\mathrm{LR}_i^{n_i}(\mathbf{x}, \mathbf{x}', L_i^{n_i})$ 的值为 假，这意味着尚未计算出秩函数。例如，$\mathbf{0}$ 是一个底元素。请注意，这直观上对应于抽象解释中的不变式推断所熟知的 $\bot$ 和 $\top$ 的含义（见第5.1节）。

求解这些公式的算法在 [7] 中有详细说明。此外，[7] 介绍了条件终止的算法，即仅当其输入满足某些条件时才会终止的程序。其目标是确定这些条件。此外，[7] 还介绍了一种使用过程级摘要进行模块化终止分析的方法。2LS 对这些算法进行了实验性实现。文献 [7, 33] 研究了如何求解模块化分析产生的这些过程间公式的近似解。此外，文献 [7, 21] 还探讨了过程间场景下的欠近似反向分析。

6.3 非终止性分析

2LS 实现了两种证明非终止性 [24, 25]的技术。这两种方法都相对简单，但在许多实际情况下似乎效率相当高。

第一种方法基于查找单点循环集。所有循环都展开 k 次（其中 k 逐步递增），随后检查是否存在某个循环 L 和一种程序配置，使其在经过某些 $k' < k$次迭代后，分别展开 k' 次和 k 次时，都能到达 L 的循环头部。这种检查可以在 2LS 中轻松表示为一个公式：该公式基于循环展开 k 次的程序的 SSA 表示。此技术能够找到这样的套索形执行：经过 k' 次初始迭代后，循环每隔 $k - k'$ 次迭代就会返回到相同的程序配置。

第二种方法通过寻找在每个整数变量上生成等差数列 的循环，尝试减少循环展开的次数。更多

具体而言，它寻找这样的循环 L：每个整数变量 x 都可以关联一个常量 c_x，使得 L 每次迭代都会将 x 的值变为 $x + c_x$，同时保持非整数变量不变。我们使用两个查询来检测此类循环：第一个查询询问，是否存在某个配置 x 和一个常量向量 c（向量取遍循环中修改的所有整数变量，常量则取自与这些变量关联的位向量域），使得 L 的一次迭代结束于配置 $x + c$；第二个查询则确保不存在这样的配置 x'，使得 L 的一次迭代会终止于不同于 $x' + c$ 的配置。如果找到了这样的循环 L 和常量向量 c，则可以如下证明 L 非终止：首先，逐步排除在 L 开头可达的每个配置 x，只要存在某个 k 使得无法从 $x + k$ 执行 L。c（直观地说，这意味着无法从 x 执行 L $k + \Gamma$ 次）。其次，检查 L 开头是否还存在未被排除且可达的配置。

终止性分析和非终止分析并行运行，并采用第一个确定的答案。在非终止分析中，首先使用单元素递归集方法尝试进行若干轮展开。如果这还不够，则尝试算术级数方法。如果仍未成功，则使用前一种方法继续进行若干轮展开。

6.4 内存安全

分析内存安全需要对指针的值和程序堆的形状进行推理。为此，我们引入一个用于堆分析的抽象域——即所谓的抽象形状域 [22]。内存安全分析基于在该域中计算不变式，然后自动生成断言，以检查是否不存在内存安全错误，例如空指针指针解引用。

6.4.1 堆分析的抽象域

与 2LS 中的所有抽象域类似，抽象形状域采用模板 的形式，用于描述所需的性质。堆的形状由内存对象之间的指针链接定义，因此我们的形状域仅限于由式 (2) 定义的所有指针组成的集合 Ptr。更具体地说，由于形状域用于推导循环不变量，因此我们将其限制为所有回边 指针组成的集合 Ptr^{lb}。

设 L 为被分析程序中所有循环的集合。我们定义

$$Ptr^{lb} = Ptr \times L. \tag{28}$$

该集合中的元素表示程序表示中引入的、从循环体末尾返回的各个指针值的抽象（参见第4.2节）。我们将回边指针 $(p, l) \in Ptr^{lb}$ 表示为 p_i^{lb}，其中 i 是循环末尾的程序位置 l。

形状域对集合Ptr^{lb}与所有符号地址$Addr$之间的可能指向关系进行上近似。我们将堆模板的形式定义为公式

$$\mathcal{T}^S \equiv \bigwedge_{p_i^{lb} \in Ptr^{lb}} \mathcal{T}^S_{p_i^{lb}}(d_{p_i^{lb}}). \tag{29}$$

模板是由模板行 $\mathrm{T}^S_{p_i^{lb}}$构成的合取，其中每一行对应一个回边指针 p_i^{lb}，并描述该指针的指向关系。行的参数 $d_{p_i^{lb}} \subseteq Addr$ （即该行的抽象值）指定了集合 $Addr$ 中的所有地址构成的集合，这些地址在位置 i处可能被 p 指向。因此，模板行可以表示为回边指针与所有可能地址之间等式的析取式：

$$\mathcal{T}^S_{p_i^{lb}}(d_{p_i^{lb}}) \equiv (\bigvee_{a \in d_{p_i^{lb}}} p_i^{lb} = a) \tag{30}$$

在给定抽象域中计算不变式，使 2LS 能够刻画程序堆的形状。例如，与抽象动态对象的指针字段对应的模板行的抽象值，描述堆中的链式路径，例如链式段。

6.4.2 内存安全断言

为分析内存安全，2LS 会自动通过断言对被分析程序进行插桩，以检查典型的内存安全错误 [22, 23]。具体而言，这些错误包括指针解引用错误、释放错误和内存泄漏。下面介绍这些断言的结构。

解引用/释放空指针

要检查这类错误，2LS 会在每个出现 $*p$ 或 free(p) 的程序位置添加断言 $p \neq$ 空指针 。由于形状域对指向关系进行了过近似，因此可以可靠地证明这类错误不存在。如果发现错误，可以使用 BMC 检查该错误是否为伪错误。

解引用/释放已释放的指针

我们引入一个特殊变量fr，并将其初始化为空指针，用于跟踪可能已释放的对象。在程序位置 i 处，程序中每次对 free(p) 的调用都会被替换为一个公式

$$fr_i = g_i^{fr} ? p_j : fr_k \tag{31}$$

其中，p_j 和fr_k 分别是 p 和fr在 i处有效的相关版本，而 g_i^{fr} 是一个自由布尔变量。该公式表示通过已释放的地址对fr 进行非确定性更新。

随后使用形状域对所有可能指向已释放的 地址的集合进行过近似；本质上，这就是所有可能已释放内存对象的集合。接着，在每个出现 $*p$ 或 free(p)的程序位置添加断言 $p \neq$ 已释放的，从而证明释放安全性。

形状域的性质保证了这种方法的可靠性；然而，将其用于抽象动态对象时，往往会导致精度很低。这是因为释放由某个抽象对象表示的具体对象之一，并不意味着其余被表示的对象无法安全地解引用或释放。通过修改第4.2.3节所述的 malloc 调用表示方式，可以解决这一问题。

除了用来表示在 i处分配的所有对象的抽象动态对象集合 AO_i 外，我们还向 AO_i添加一个对象 ao_i^{co}。该对象可以非确定性地被选作 malloc 的返回结果（与其他任何 ao_i^k一样），但它保证表示一个具体对象（即只能分配一次）。这是通过添加一个条件实现的：如果在分配点 i的入口处有指针指向 ao_i^{co}，则断言 ao_i^{co}不能被分配。因此，malloc 的表示形式如下：

$$p_i = (g_{i,co}^{os} \wedge \bigwedge_{p \in Ptr} p \neq \&ao_i^{co}) \,?\, \&ao_i^{co} : (g_{i,1}^{os} \,?\, \&ao_i^1 : (\dots)) \tag{32}$$

之后，在每个分配点 i，只允许将具体对象 ao_i^{co}的地址赋给fr。释放安全性检查也只针对具体对象进行，这有助于避免上述精度不足。由于 ao_i^{co}表示在 i 处分配的任意对象，因此这种方法仍然可靠；如果能够证明它是安全的，就可以假定在 i处分配的所有对象也都是安全的。

内存泄漏安全性

与上一节类似，变量fr 用于检查是否存在内存泄漏风险。在程序结束时，我们检查是否存在对象 ao_i^{co}满足 $fr \neq \&ao_i^{co}$。如果找到这样的对象，就说明存在内存泄漏。然而，只有对不含循环的程序（或所有循环都已完全展开的程序），才有可能证明不存在内存泄漏。这是因为我们不跟踪表示在同一分配点分配的具体对象的抽象对象的顺序，而我们的分析通常会发现，在释放循环中可以跳过 ao_i^{co}，因此无法就是否存在内存泄漏得出结论。

6.5 域组合

在 2LS 中实现的程序验证的一项主要优势是，所有抽象域都必须采用统一的模板形式——无量词一阶公式。得益于这一特性，我们可以相当直接地组合不同的域，同时依靠求解器为域运算符的组合以及域抽象值之间的相互归约完成繁重的工作。在 2LS 中，我们使用两种此类组合：积模板 和幂模板，尤其是其中称为带符号路径的模板 [22]的形式。

6.5.1 积模板

积模板是 2LS 中最简单的抽象域组合形式之一。它们基于使用笛卡尔积模板，将不同类型的域并列组合起来。只需对它们的模板取合取即可实现这一点。

这种形式中一个非常有趣的组合是形状抽象域与多面体抽象域的组合。它使 2LS 能够同时分析指针类型和数值类型变量的值。这不仅有助于分析同时操作指针和数值的程序，还提供了推理程序堆中数据结构内容的可能性。得益于这种组合，我们可以验证如图 2 所示的程序。

6.5.2 使用符号路径的模板

使用简单的不变式模板（例如上述多面体模板或堆形状模板），可能不足以精确分析某些程序，尤其是处理抽象动态对象的程序。在这类程序中，通常需要为循环 l 计算的不变式能够区分在到达 l 之前哪些循环已经执行、哪些尚未执行。在处理循环中分配的抽象动态对象时，这使我们能够区分这样的情况：某个抽象动态对象并不代表任何实际分配的对象，因此描述它的不变式并不成立。

为此，为了提高精度，2LS 引入了符号循环路径这一概念。符号循环路径表示程序中执行过哪些循环。由于 2LS 使用循环选择守卫来捕捉循环中的控制流，因此符号路径就是循环选择守卫文字的合取。

形式化地，令 G^{ls} 表示被分析程序中所有循环选择守卫的集合。符号循环路径 π 定义为

$$\pi = \bigwedge_{g \in G^{ls}} l_g \tag{33}$$

其中，l_g 是 g的一个文字量，即 g 或 $\neg g$。我们用 Π表示所有符号路径的集合。我们还定义了一条仅包含负文字量的特殊路径 $\pi_\perp$。对于该路径，不计算循环不变量，因为该路径未执行任何循环。

给定一个表示抽象域（例如形状域，甚至是积域）的模板 T，我们将相应的带符号路径的模板定义为

$$\mathcal{T}^L \equiv \bigwedge_{\pi \in \Pi} \pi \implies \mathcal{T}. \tag{34}$$

该模板可以被视为一种幂模板——就幂域的意义而言——它为基域中的每个元素分配指数域中的一个元素。

从符号路径模板推断不变式 Inv^L 的算法如算法 14 所示。

$Inv^L \leftarrow$**真对于** $\pi \in \Pi$**执行**将 断言 **(**π**)** 添加到求解器$Inv_\pi \leftarrow$ 从 T 计算不变式 从求解器中移除断言**(**π**)** **如果 (**$\pi \wedge Inv\pi$**)** 可满足**则**$Inv^L \leftarrow Inv^L \wedge (\pi \implies Inv\pi)$**结束 if 结束 for**

图 14：从符号路径模板推断不变式的算法。

该算法会针对每条符号路径 π，在内部域中使用模板 T 计算一个单独的不变式 $Inv\pi$。为仅限制对 π 的不变式计算，我们断言在计算过程中 π（表示执行了哪些循环的公式）成立。计算出 $Inv\pi$ 后，也可能发现该符号路径实际上在被分析程序中不可达。因此，我们通过在被分析程序 SSA 生成的公式上下文中求解 $\pi \wedge Inv\pi$，来检查其可达性。如果 π 可达，则将 $\pi \implies Inv\pi$ 合取到最终的不变式中；否则，舍弃$Inv\pi$。

7 2LS 的实现与架构

2LS 基于 CPROVER 框架。[8] 2LS 使用 C语言++ 实现，目前约有 25 千行代码（不包括 CPROVER 本身）。除了 CPROVER 之外，2LS 没有任何外部依赖，因此可以在许多平台上构建。不过，目前仅在 Linux 上经过了充分测试。

[8] https://www.cprover.org

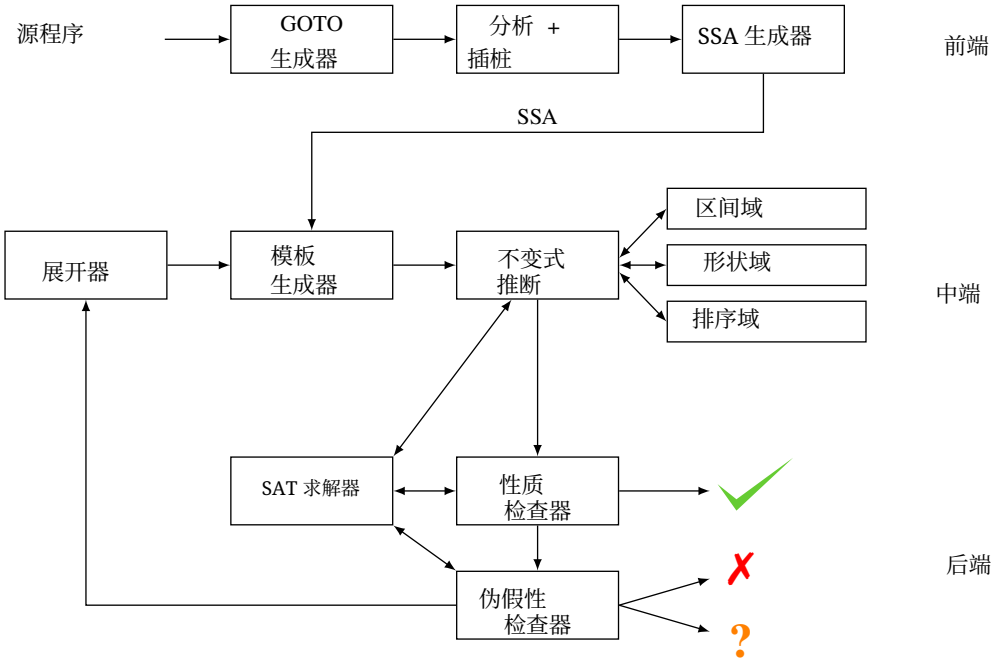


图 15：2LS 架构概览。

源代码采用 BSD 许可证，可在 https://www.github.com/diffblue/2ls 获取。2LS 最初由 Daniel Kroening 和 Peter Schrammel 于 2014年 开发，此后已有超过 15 位贡献者参与贡献。

2LS 与大多数 CPROVER 工具兼容，例如 goto-cc（可直接替代 GCC），它可以使用待分析项目现有的构建系统来构建和链接 C源代码。生成的 *goto二进制文件* 可传递给 2LS 和其他 CPROVER 工具进行分析，例如 Cbmc。

架构

与其他验证工具类似，2LS 遵循编译器的流水线架构，包括用于解析源代码和进行类型检查的前端；用于基于中间表示（通常采用静态单赋值形式，即 SSA）转换和优化代码的中端过程；以及负责分析和验证，而非代码生成的后端。图15展示了 2LS 架构概览。

8 评估

现在，我们介绍 2LS 框架在国际软件验证竞赛（SV-COMP）中的表现。自 2016 年起，2LS 一直参加 SV-COMP。第一部分将比较 2LS 多年来在所选类别中取得的分数，这些类别要求运用 2LS 所实现的分析方法。第二部分将介绍参赛工具的一些替代排名，以展示 2LS 除成功验证程序数量之外的优势（例如时间或功耗）。

8.1 SV-COMP 中的得分

在本节中，我们展示 2LS 在 SV-COMP 中获得的得分多年来（从 2016 年到 2022 年）的变化。我们重点介绍促使得分显著提升的重要里程碑，并提及该工具取得的一些成就。

由于 SV-COMP 包含种类繁多的验证任务，我们选择了若干类别来展示该工具的能力。我们重点关注那些包含此类任务的类别：其验证需要用到 2LS 所提供、并在本章中介绍过的程序分析技术。具体而言，我们展示以下类别中的得分：

循环 +浮点数 此处，我们给出 浮点数 和循环 两个子类别的得分总和，这两个子类别隶属于可达性安全类别[9]。验证这些类别中的程序需要推断数值变量上的不变式，2LS 使用模板多面体域（第6.1节）来完成这一工作。具体而言，我们始终使用区间模板。

堆 2LS 的优势之一是能够分析堆操作程序。在 SV-COMP 中，这类程序属于可达性安全-堆 子类别（用于检查用户定义断言的可达性），以及内存安全 类别（工具在此验证是否不存在内存安全错误）。本部分给出了这两个类别及子类别的总分。为了分析本部分中的程序，2LS 使用第 6.4.1 节所述的抽象形状域。

终止性 除了验证安全性质外，2LS 还可以通过同时运行终止性分析（第 6.2 节）和非终止性分析（第 6.3 节），来验证程序是否终止。在 SV-COMP 中，终止性在终止性 类别中进行验证。

总体 我们展示的最后一类结果来自总体 类别，该类别包含对 SV-COMP 中所有任务程序进行验证后得到的（归一化）得分。

[9] 2016 年的 SV-COMP 类别布局与现在大不相同，不过浮点数 和循环 类别当时已经存在。浮点数 是一个独立类别，而循环 则是整数控制流的一个子类别。这里展示的 2016 年结果反映了这些类别及子类别中的得分。

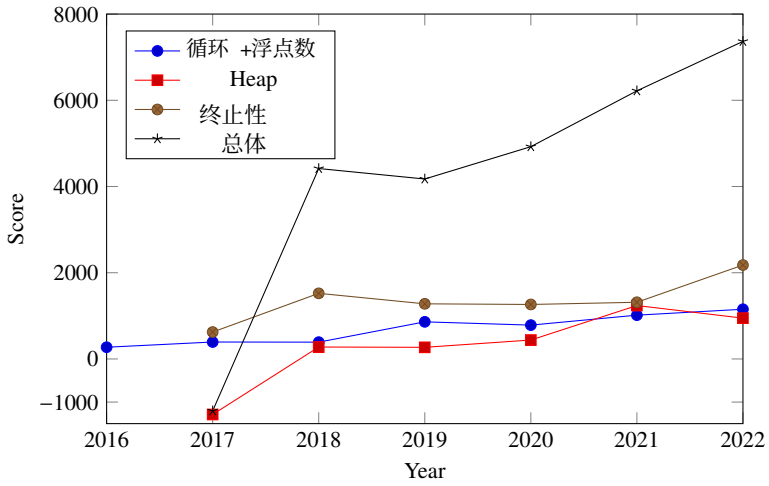


图 16：SV-COMP 2016—2022 年的精选结果

图 16 展示了 2LS 在 2016 年至 2022 年间这些类别的得分变化（2016 年，2LS 未参加大多数类别的竞赛）。最显著的进步出现在 2018 年，堆和终止性类别的得分大幅提高，也因此带动总体得分上升。这是因为 2018 年，2LS 加入了抽象形状域和非终止性分析。

接下来，我们选取 2LS 在上述类别中的一些成果，以佐证有关该框架优势的论述：

- **第**一名，浮点数和**第**二名，循环类别中，2016 年，
- **第**三名，终止性类别中，2020、2021 和 2022 年
- **第四**名，堆类别中，2019 年。

最后，2LS 的优势之一是能够同时推理无界动态数据结构的形状和内容（这得益于第 6.5 节介绍的抽象域组合）。这类程序没有单独的类别；唯一需要进行此类推理的任务程序，是我们团队在 2019 [22] 中添加的。2022 年，2LS 仍是仅有的两个能够验证这些任务中大多数任务的工具之一。

8.2 替代排名

除了能够可靠且正确地验证程序之外，还有其他一些属性可能有助于验证工具的实用性。在本节中，

我们使用一些替代指标将 2LS 与其他验证器进行比较，具体包括验证速度、能耗和正确率。

8.2.1 2LS 的验证速度

从 SV-COMP 的结果中可以观察到，2LS 相较于其他工具，能够在很短的时间内验证大多数任务。我们通过一项实验来支持这一观点：设置较短的时间限制，然后观察 2LS 在 SV-COMP'22 中与其他工具的竞争表现。表2展示了如果时间限制设为 5 秒，2LS 在若干主要类别中能够取得的名次。

表2：时间限制为 5 秒时，2LS 在 SV-COMP'20 若干类别中的名次。

可达性安全	内存安全	无溢出	终止性	总体	
5.	5.	1.	1.	2.	

表格显示，2LS 在所有提及的类别中都能取得较高名次。值得注意的是，它在*终止性* 和*无溢出*类别中排名第一，在*总体* 类别中排名第二。

8.2.2 功耗与正确率

近年来，SV-COMP [1] 竞赛报告提供了两种替代排名，以体现验证器在验证过程中的不同表现。具体如下：

- *正确验证器*排名依据所谓的*正确率*对验证器进行排序，该比率为错误结果数量与总得分之比。2LS 在 2020 年的这一排名中位列第三，每个得分点仅有 0.0016 个错误[10]。
- *绿色验证器*根据获得单个得分点所消耗的能量对验证器进行排名。2020 年和 2021 年，2LS 在这一排名中位列第二，每个得分点仅消耗 180 焦耳。

8.3 评估结论

总体而言，我们总结了从 SV-COMP 结果中得到的观察。2LS 既能够证明性质成立，也能够为性质违规找到反例。

[10] 在 SV-COMP 中，工具若能在程序中找到漏洞，可得 1 分；若能证明程序安全，则可得 2 分。最终得分根据各个类别中获得的分数经归一化后计算得出

对于它能够证明的大多数任务，它都能非常快速地完成证明，这主要归功于对单个 SAT 实例进行增量展开。

然而，2LS 无法扩展到更复杂的问题。原因有很多：模板可能过于弱；不变式推断无法扩展到复杂模板；k-归纳往往无法奏效。

此外，2LS 仍不支持验证程序的若干特性，例如数组内容、并发或递归。这使 2LS 在某些类别（内存安全、总体）中的排名难以进一步提升。

9 结论与展望

2LS 是一种用于 C 程序的验证系统，它基于有界模型检测、 *k*-归纳和基于模板的不变式推断的简单组合。该框架简单易用，便于尝试新的抽象域。就速度而言，2LS 击败了大多数其他最先进工具，但在稳健性和特性支持方面仍然落后。例如，它目前尚不支持并发。此外，其增量结构转换算法尚不支持递归。要处理诸如不存在内存泄漏之类的属性，还需要进一步改进内存模型。数组不变式推断也是一项正在进行的工作。

参考文献

1. Beyer, D.：自动软件验证的进展：Sv-comp 2020。载于：A. Biere、D. Parker（编）《系统构造与分析的工具和算法》，第347–367页。Springer（2020）2. 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：《系统构造与分析的工具和算法》，计算机科学讲义，第1579卷，第193–207页。Springer（1999）3. Bradley, A.R., Manna, Z.：通过对归纳反例进行归纳泛化来检查安全性。载于：《计算机辅助设计中的形式方法》，第173–180页。IEEE 计算机学会（2007）4. Brain, M., Joshi, S., Kroening, D., Schrammel, P.：基于*k*-不变式和 *k*-归纳的安全性验证与反驳。载于：《静态分析专题研讨会》，计算机科学讲义，第9291卷，第145–161页。Springer（2015）5. Brauer, J., King, A., Kriener, J.：作为增量SAT的存在量化。载于：《计算机辅助验证》，计算机科学讲义，第6806卷，第191–207页。Springer（2011）6. Cabodi, G., Camurati, P., Mishchenko, A., Palena, M., Pasini, P.：IC3中的SAT求解器管理策略：一种实验方法。《系统设计中的形式方法》**50**(1)，39–74（2017 0 7 10 1007 10703 017 0272 0 10 1007 10703 017 0272）。DOI . /s - - - 。网址 https://doi.org/ . /s - - - 。Chen, H.Y., David, C., Kroening, D., Schrammel, P., Wachter, B.：位精确的过程模块化终止性证明。《TOPLAS》**40** （2017）

8. Clarke, E., Kroening, D., Lerda, F.：用于检查ANSI-C程序的工具。载于：《系统构造与分析的工具和算法》，计算机科学讲义，第2988卷，第168–176页。Springer（2004）9. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.：反例引导的抽象细化。载于：《计算机辅助验证》，计算机科学讲义，第1855卷，第154–169页。Springer（2000）10. Cousot, P., Cousot, R.：抽象解释：通过构造或近似不动点，为程序静态分析建立统一的格模型。载于：《编程语言原理》，第238–252页（1977）11. D'Silva, V., Haller, L., Kroening, D.：抽象冲突驱动学习。载于：《编程语言原理》，第143–154页。ACM（2013）12. Eén, N., Sörensson, N.：可扩展的SAT求解器。载于：SAT，计算机科学讲义，第2919卷，第502–518页。Springer（2003）13. Eén, N., Sörensson, N.：通过增量SAT求解进行时序归纳。《理论计算机科学电子札记》**89:4**，543–560（2003）14. Gawlitza, T.M., Monniaux, D.：通过SMT求解改进策略。载于：ESOP，计算机科学讲义，第6602卷，第236–255页。Springer（2011）15. Gawlitza, T.M., Seidl, H.：通过策略迭代获得精确的关系不变量。载于：《计算机科学逻辑》，计算机科学讲义，第4646卷，第23–40页。Springer（2007）16. Gulwani, S., Srivastava, S., Venkatesan, R.：将程序分析视为约束求解。载于：《编程语言设计与实现》，第281–292页。ACM（2008）17. Günther, H., Weissenbacher, G.：增量式有界软件模型检查。载于：SPIN，第40–47页。ACM（2014）18. Hoder, K., Bjørner, N.：广义性质导向可达性。载于：《可满足性测试理论与应用》，计算机科学讲义，第7317卷，第157–171页。Springer（2012）19. Hooker, J.N.：求解增量可满足性问题。《JLP》**15**(1&2)，177–186（1993）20. Kim, J., Whittemore, J., Sakallah, K.A., Silva, J.P.M.：关于将增量可满足性应用于延迟故障测试。载于：DATE，第380–384页。IEEE 计算机学会（2000）21. Madhukar, K., Schrammel, P., Srivas, M.K.：组合式安全性反驳技术。载于：《验证与分析自动化技术》，计算机科学讲义，第10482卷，第164–183页。Springer（2017）22. Malík, V., Hruška, M., Schrammel, P., Vojnar, T.：对操作堆的程序进行基于模板的验证。载于：FMCAD，第103–111页（2018）23. Malík, V., Hruška, M., Schrammel, P., Vojnar, T.：2ls：堆分析与内存安全（竞赛贡献）（2019 00712 24 1903）。网址 http://arxiv.org/abs/ . . Malík, V., Martiček, Š., Schrammel, P., Srivas, M., Vojnar, T., Wahlang, J.：2LS：内存安全与非终止性（竞赛贡献）。载于：TACAS，第417–421页。Springer（2018）25. Martiček, Š.：从模板综合非终止证明。硕士论文，布尔诺理工大学，布尔诺（2017）26. Marušák, M.：程序抽象的通用基于模板综合。硕士论文，布尔诺理工大学，布尔诺（2019）27. McMillan, K.L.：使用插值的惰性抽象。载于：《计算机辅助验证》，计算机科学讲义，第4144卷，第123–136页。Springer（2006）28. Miné, A.：八边形抽象域。载于：《逆向工程工作会议》，第310–319页。IEEE 计算机学会（2001）29. Monniaux, D., Schrammel, P.：加速逻辑数值策略迭代。载于：《静态分析专题研讨会》，计算机科学讲义，第8723卷，第253–267页。Springer（2014）30. Mukherjee, R., Schrammel, P., Haller, L., Kroening, D., Melham, T.：将CDCL提升至用于程序验证的基于模板的抽象域。载于：《验证与分析自动化技术》，计算机科学讲义，第10482卷，第307–326页。Springer（2017）

31. Reps, T.W., Sagiv, S., Yorsh, G.: 最佳变换器的符号实现。载于：验证、模型检测与抽象解释，计算机科学讲义，第 2937 卷，页码 252–266。Springer（2004）32. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: 使用数学规划分析线性系统的可扩展方法。载于：验证、模型检测与抽象解释，计算机科学讲义，第 3385 卷，页码 25–41。Springer（2005）33. Schrammel, P.: 分解验证问题编码所面临的挑战。载于：HCVS@ETAPS 2016，*EPTCS*，第 219 卷，页码 29–32（2016）34. Schrammel, P., Kroening, D.: 用于程序分析的 2LS——（竞赛贡献）。载于：系统构造与分析的工具和算法，计算机科学讲义，第 9636 卷，页码 905–907。Springer（2016）35. Schrammel, P., Kroening, D., Brain, M., Martins, R., Teige, T., Bienmüller, T.: 面向嵌入式软件的增量有界模型检测。计算的形式化方面 **29**(5)，911–931（2017年）36. Sheeran, M., Singh, S., Stålmarck, G.: 使用归纳和 SAT 求解器检查安全性质。载于：FMCAD，计算机科学讲义，第 1954 卷，页码 108–125。Springer（2000）37. Silva, J.M., Sakallah, K.A.: 用于测试模式生成的鲁棒搜索算法。载于：FTCS，页码 152–161。IEEE 计算机学会（1997）38. Strichman, O.: SAT 有界模型检测问题的剪枝技术。载于：CHARME，计算机科学讲义，第 2144 卷，页码 58–70。Springer（2001）39. Whittemore, J., Kim, J., Sakallah, K.A.: SATIRE：一种新型增量可满足性引擎。载于：DAC，页码 542–545。ACM（2001）