Advances in Applied Mathematics 应用数学进展, 2025, 14(11), 55-61
Published Online November 2025 in Hans. https://www.hanspub.org/journal/aam
https://doi.org/10.12677/aam.2025.1411461

Hans汉斯

基于Frama-C的希尔排序形式化验证

陈　琦，姜雨函，江建国*

辽宁师范大学数学学院，辽宁　大连

收稿日期：2025年10月4日；录用日期：2025年10月28日；发布日期：2025年11月4日

摘　要

希尔排序是一种基于间隔递减的经典排序算法，其正确性对软件可靠性至关重要。本文采用形式化方法，借助Frama-C平台及其ACSL规范语言，对希尔排序程序进行源代码级别的正确性验证。验证过程中的主要难点在于构造嵌套循环的循环不变式，以描述数组的有序性和元素排列性质。通过设计合适的循环不变式，并借助Alt-Ergo、CVC4、Z3等定理证明器进行验证，最终证明了希尔排序程序的功能正确性与内存安全性。

关键词

程序验证，形式化方法，循环不变式，希尔排序

Formal Verification of Shell Sort Based on Frama-C

Qi Chen, Yuhan Jiang, Jianguo Jiang*

School of Mathematics, Liaoning Normal University, Dalian Liaoning

Received: October 4, 2025; accepted: October 28, 2025; published: November 4, 2025

Abstract

Shell sort is a classic sorting algorithm based on decreasing intervals. Its correctness is crucial for software reliability. This paper uses a formal method and based on the Frama-C platform and its ACSL specification language, conducts source code-level correctness verification of the Shell sort program. The main difficulty in the verification process lies in constructing the loop invariants of nested loops to describe the order of the array and the arrangement properties of elements. By designing appropriate loop invariants and using theorem provers such as Alt-Ergo, CVC4, and Z3 for verification, the functional correctness and memory safety of the Shell sort program are ultimately proved.

*通讯作者。

文章引用: 陈琦, 姜雨函, 江建国. 基于 Frama-C 的希尔排序形式化验证[J]. 应用数学进展, 2025, 14(11): 55-61.
DOI: 10.12677/aam.2025.1411461

Keywords

Program Verification, Formal Methods, Loop Invariant, Shell Sort

1. 引言

排序是计算机科学中最基础、最常用的算法之一，许多实际应用的核心功能都依赖于高效的排序。希尔排序作为一种经典的排序算法，在数据处理过程中起到了不可或缺的作用，一旦出错将在软件开发过程中产生巨大损失，为此，保证希尔排序程序在数据处理过程中的正确性是十分重要的。

相关工作

目前，验证程序正确性的方法主要分为两种，分别是测试和形式化方法，测试通过运行程序来检验程序是否满足规定的要求，在一定程度上能保证程序的正确性，但是不能保证程序没有错误，而形式化方法通过数学方法严格证明一个程序的正确性，确保程序没有错误[1]。2019 年，de Gouw，Stijn 对 TimSort 算法进行形式化验证[2]，使用定理证明器验证 Java 和 Python 中，发现了一个可导致数组越界的致命错误，该错误已存在多年而未通过测试被发现。这一案例强有力地证明了形式化验证在发现深层逻辑错误方面不可替代的价值。

Danijela Petrović 主要探讨了如何使用形式化方法和程序精化技术，在 Isabelle/HOL 中对选择排序和堆排序进行验证[3]；Isabela Drămnesc 等人在 2024 年使用 Coq 证明了排序算法[4]，而目前没有文献对希尔排序进行形式化验证。因此，本文将采用形式化方法去验证希尔排序程序的正确性。

在验证方法上，已有研究多依赖于交互式定理证明器(如 Isabelle/HOL、Coq)，需要大量的人工引导和数学推理。本文则采用 Frama-C 平台及其 WP 插件进行自动化演绎验证，通过生成验证条件并交由 SMT 定理证明器(如 Alt-Ergo、CVC4、Z3)自动证明，在保证严谨性的同时，显著提升了验证的自动化程度，更贴近于对工业级 C 程序进行验证的实践需求。

希尔排序的核心思想是使数组中任意间隔为 gap 的元素都是有序的，本文明确采用希尔排序的原版间隔序列(即初始间隔为 n/2，并逐次减半)，最终完成排序。这一选择使得我们的循环不变式能够紧密贴合算法的实际执行路径，从而能够精确地描述每一轮间隔下数组的状态演变。使用形式化方法去验证希尔排序程序的正确性难点主要是如何处理嵌套循环，因为对于基本的程序结构，比如赋值语句、if 语句和顺序语句，Hoare 逻辑给出了生成最弱前置条件的方法，但对于循环语句，并没有通用的规则可以为其生成最弱前置条件。Hoare 提出使用一个断言构造规则来验证循环程序，将其描述为“在任意一次执行循环体前后均为真的断言”，该类断言被定义为循环不变式，即在循环体的每次执行前后均为真的谓词[5]。因此，构造循环不变式是理解、证明希尔排序程序正确性的基础。

本文选取 Frama-c 平台去验证希尔排序程序的正确性，因为最弱前置条件是 Frama-c 验证平台中 WP 插件的工作原理，WP 插件以最弱前提演算命名，它是一个最弱的前置条件演算，用于为带有 ACSL 注释的 C 程序生成验证条件[6]。Frama-c 是一个源代码分析平台，旨在对工业规模的 C 程序进行验证。它为用户提供了一系列插件，用于执行静态分析、演绎验证和测试。通过学习 ACSL 规范语言[7]，描述数组的排列属性以及有序性，调用 Alt-Ergo、CVC4、Z3 等定理证明器证明构造的循环不变式成立，最终证

明希尔排序程序的正确性。

2. 理论基础

2.1. 最弱前置条件(Weakest Preconditions)

最弱前置条件演算的原理本质上非常简单，给定程序的代码注释，比如说，语句 P 之后的 φ 断言，ϕ 的最弱前置条件定义为“最简单”的属性 ϕ，该属性必须在 P 之前有效，使得 φ 在 P 执行之后仍然有效。

定义 1 形如 $(|\phi|)P(|\varphi|)$ 的规范称为 Hoare 三元组，公式 ϕ 称为 P 的前置条件，φ 称为 P 的后置条件。

定义 2 如果对满足 ϕ 的所有状态，只要 P 实际终止，执行 P 后的结果状态就满足后置条件 φ，我们就说三元组满足部分正确性，此时，关系 $\vDash_{\text{par}} (|\phi|)P(|\varphi|)$ 成立，$\vDash_{\text{par}}$ 为部分正确性的满足关系。

定义 3 最弱前置条件是指保证一条语句执行正常结束并满足语句的后置条件的最弱的前提条件。最弱前置条件是一个谓词公式，通常用 $wp(S,Q)$ 表示，我们可以通过证明 $P \Rightarrow wp(S,Q)$ 来证明 $\{P\}S\{Q\}$。

例 1 考虑整数局部变量 x 上的简单赋值 $x = x+1$：

在这种简单的情况下，可以通过将 φ 中的 x 替换为 $x+1$ 来获得 φ 在 x 上的分配的最弱前置条件，为 $\{x+1>0\}\ x = x+1; \{x>0\}$。

定义 4 部分 while 语句

$$\frac{(|\varphi \wedge B|)C(|\varphi|)}{(|\phi|)\ \text{while}\ B\{C\}(|\varphi \wedge \neg B|)}$$

φ 为循环不变式，表达了在任何执行操作下都保持不变的关系，如果开始时 φ 为真，且 while 语句终止，那么结束时 φ 仍为真，而且因为 while 语句终止，故 B 为假。

2.2. ACSL 语言

ACSL 是 ANSI/ISO C Specification Language 的首字母缩写，是一种在 Frama-C 框架中实现的行为接口规范语言(Behavioral Interface Specification Language, BISL)，旨在指定 C 源代码的行为属性，主要以注释的形式存在于代码之中[7]。

1. 核心结构：

必须以/*@或//@开头，并像 C 语言中一样结束。

2. 逻辑表达式：

ACSL 使用逻辑表达式来描述属性，它们类似于 C 语言的布尔表达式，但有一些扩展和关键区别：

1) 全称量词(forall)：\forall type v；condition；expression；

2) 存在量词(exists)：\exists type v；condition；expression；

3) 结果(result)：\result

在函数契约的 ensures 子句中，指代函数的返回值。

4) 数组逻辑：

可以使用\valid 等谓词，但直接使用下标(如 a[i])也是常见的。

3. 函数契约

函数契约是 ACSL 的核心，它规定了函数的职责和调用者必须满足的条件。它必须放在在函数声明之前。

1) requires 子句(前置条件)：requires P；

定义了函数被调用时必须满足的条件。

2) ensures 子句(后置条件)：ensures Q；

定义了函数在正常返回时必须确保成立的条件。

3) assigns 子句(分配子句)：assigns S；

指定了函数可能修改的所有内存位置。

4. 语句注解

用于在代码块内部标注中间属性，帮助工具进行推理。

1) 断言：//@ assert P；

声明在程序的这一点上，逻辑表达式 P 必须为真。

2) 循环注解：

① 循环不变式：//@ loop invariant I；

这是验证循环正确性的关键。它是一个在每次循环迭代开始时都必须为真的谓词。它通常包含循环索引的范围和循环已经完成的工作。

② 循环分配：//@ loop assigns...；指定循环体内修改的变量。

③ 循环变体：//@ loop variant V；提供一个每次迭代都会严格递减(并下界为 0)的表达式，用于证明循环会终止。

3. 验证过程

希尔排序的核心思想是使数组中任意间隔为 gap 的元素都是有序的，通过逐渐减小这个间隔 gap，最终完成排序。我们在验证希尔排序程序正确性过程中，从最基本的内存安全性和功能正确性两个方向入手，功能正确性在本文中特指一段程序执行前后程序状态变化需要满足的性质，即证明希尔排序最终能产生一个有序的数组，以及确保数组在有效访问范围内，避免发生溢出等错误。

3.1. 内存安全性

我们必须保证程序的数组访问是在有效范围内，在 requires 子句中使用\valid(a + (0..n-1))。由于希尔排序会按较大的间隔进行比较和交换，必须确保所有索引，如 i，j 在任何时候都不会越界。通过循环不变式定义 i，j，gap 的范围，或者可以在程序后添加断言确保当前索引都在数组边界内。

3.2. 功能正确性

这是验证的核心，即证明希尔排序最终确实能产生一个有序的数组。本文将从元素排列属性和升序验证两个方向验证希尔排序的功能正确性。必须保证数组元素是升序排列的，以及排序后的数组是初始数组的一个排列，即算法没有创建、删除或篡改元素，只是改变了它们的位置。

一、元素排列性质

元素的排列性质保证了数组元素相同，顺序可能不同，定义谓词

```
permut{K,L}(int *a, integer n) =
    \forall integer k; 0 <= k < n ==>
      \exists integer p; 0 <= p < n && \at(a[k], K) == \at(a[p], L) &&
    \forall integer p; 0 <= p < n ==>
      \exists integer k; 0 <= k < n && \at(a[p], L) == \at(a[k], K);
```

表示排序前后数组是彼此排列的关系，即 K 处的数组中的每个元素，都能在 L 处的数组中找到。在

ensures 子句中写入 permut{Pre,Post}(a,n)；确保结果是输入的一个排列。如果我们想知道在循环过程中数组是否为排列关系，可以在每层循环过程中添加循环不变式：

loop invariant permut: permut{Pre,Here}(a, n);

在验证元素排列性质时还可以定义以下定理辅助排列性质验证：

```
lemma permut_refl{L}:
    \forall int *t, integer n; permut{L,L}(t, n);
lemma permut_sym{K,L}:
    \forall int *t, integer n;
    permut{K,L}(t, 0, n) ==> permut{L,K}(t, n);
lemma permut_trans{K,L,M}:
    \forall int *t, integer n;
    permut{K,L}(t, n) ==> permut{L,M}(t, n) ==> permut{K,M}(t, n);
```

分别表示 permut 的自反性，对称性和传递性。

二、升序验证

1、数组递增

为了保证希尔排序最终呈现排列后是一个有序的数组，定义以下谓词：

```
predicate increasing{L}(int * a, integer n) =
    \forall integer k; 0 <= k < n-1 ==> a[k] <= a[k+1];
```

表示数组 a 是递增的(已排序)。

而希尔排序在循环过程中随着间隔 gap = n/2 逐层递减，内层循环表示数组 a 在当前间隔 gap 下是部分有序的，于是我们定义谓词：

```
predicate gapsort{L}(int * a, integer n, integer gap) =
      gap > 0 && gap <= n &&
    \forall integer k; 0 <= k < n-gap ==> a[k] <= a[k+gap];
```

直至 gap 逐渐递减到 1 时，数组 a 最终是递增的，也就是验证后置条件：

ensures increasing: increasing{Post}(a,n);

而从部分有序到最终有序，我们可以写一个定理辅助验证我们的想法是否正确：

```
lemma gapsort_increasing{L}:
    \forall int * a, integer n; gapsort{L}(a, n, 1) ==> increasing{L}(a, n);
```

表示数组 a 在间隔 gap = 1 时最终有序。

2、寻找循环不变式

希尔排序外层循环表示以间隔 gap = n/2 逐层递减，中间循环和内层循环为插入排序的过程，要保证嵌套循环的每层循环不变式都成立，可以先从最内层循环入手，从最内层循环到最外层循环逐层验证。

希尔排序最内层循环代码如下：

```
for (j = i - gap; j >= 0 && a[j] > a[j+gap]; j -= gap) {
                    swap(&a[j], &a[j+gap]);}
```

最内层循环负责交换 a[j]和 a[j+gap]的顺序，遍历数组 a[0..n-1]，随着中间层循环 i 的递增，数组 a 在当前 gap 下子序列是有序的，并不只是单纯地 a[j]<a[j+gap]，

在最内层循环结束后添加断言//@assert j >= 0 ==> a[j] <= a[j+gap]，更加直观地看出循环结束后 a[j]是否能小于 a[j+gap]。

为了表示数组 a 在间隔 gap 下子序列有序，定义谓词：

```
predicate subseq_sorted{L}(int * a, integer n, integer gap, integer start) =
gap > 0 && 0 <= start < gap && start < n && \forall integer k;
0 <= k && start + k*gap < n-gap ==> a[start+k*gap] <= a[start+k*gap+gap];
```

随着 gap 逐渐减小，数组 a 的子序列都逐渐有序，最终演变成在当前 gap 下部分有序，定义引理：

```
lemma subseq_sorted_gapsort{L}:
        \forall int *a, integer n,gap;
          n >= 0 && gap > 0 && \forall integer s; 0 <= s < gap ==>
          subseq_sorted{L}(a, n, gap,s) ==> gapsort{L}(a,n,gap);
```

表示对数组 a 的所有起始点 s(0 <= s < gap)的子序列都有序，能推出数组 a 部分有序，这便是每层循环的循环不变式，用 loop invariant \forall integer s; 0 <= s < gap ==> subseq_sorted{Here}(a, n, gap, s)表示。由于 Frama-c 平台对这么复杂的定理不能进行证明，下面给出子序列有序和整体 gap 有序之间的蕴涵关系的详细的数学证明：

前提条件：假设对于所有起始位置 s(0 <= s < gap)，相应的子序列 a[s], a[s+gap], a[s+2*gap], ...都是有序的，即满足 subseq_sorted{L}(a, n, gap, s)。

结论：需要证明整个数组满足 gapsort{L}(a, n, gap)，即对于任意 k(0 <= k < n-gap)，都有 a[k] <= a[k+gap]。

证明过程：对于任意满足 0 <= k < n-gap 的索引 k，令 s = k mod gap，即 s 是 k 除以 gap 的余数。

由于 0 <= s < gap，根据前提条件，以 s 为起点的子序列是有序的。

存在整数 m 使得 k = s + m * gap，因为 s 是 k 模 gap 的余数。

由于 k < n-gap，所以 s + m * gap < n - gap，即 s + m * gap 是子序列中有效的索引。

根据 subseq_sorted 的定义，对于该子序列中的任意连续元素，有 a[s + m * gap] <= a[s + (m+1) * gap]。

即 a[k] <= a[k + gap]，这正是 gapsort 所要求的性质。

3、验证

在上述代码中调用了 swap 交换函数，所以还需要书写 acsl 注释验证 swap 程序的正确性，即 a[j]和 a[j+gap]是否交换了位置，代码如下：

```
/*@
   requires valid:      \valid(p);
   requires valid:      \valid(q);
   assigns                 *p;
   assigns                 *q;
   ensures   exchange: *p == \old(*q);
   ensures   exchange: *q == \old(*p);
*/
void swap(int * p, int * q){
   int tmp = *p;
   *p = *q;
   *q = tmp;}
```

内层循环的核心是通过相邻元素的交换操作逐步构建子序列的有序性。具体而言，对于当前间隔 gap 和起始位置 s (其中 0 <= s < gap)，内层循环维护以下关键性质：

1) 局部有序性：在每次内层循环迭代中，当发现 a[j] > a[j+gap]时，通过交换操作确保 a[j] <= a[j+gap]成立。

2) 传递性：交换操作不仅影响当前位置，还会通过后续的 j -= gap 操作向前传递，使得整个以 s 为起点的子序列 a[s], a[s+gap], a[s+2*gap], ...逐步达到有序状态。

3) 不变式保持：内层循环开始时，假设子序列 a[s], a[s+gap], ..., a[i-gap]已经有序；循环体通过交换操作将 a[i]插入到正确位置，使得子序列扩展到 a[s], a[s+gap], ..., a[i]仍然保持有序。

本文使用 Frama-c 平台中的 wp 插件对上述书写的代码进行验证，调用 CVC4，Alt-ergo，z3 这三个定理证明器辅助内置插件 Qed 进行验证，在终端中输入代码 frama-c -wp -wp-prover cvc4,z3,alt-ergo shell-sort.c swap.c，所有验证条件均被成功证明，表明希尔排序程序完全符合其形式化规范，具有内存安全性和功能正确性。

4. 结论

本文基于形式化验证方法，借助 Frama-C 平台及其 ACSL 规范语言，对希尔排序程序进行了严格的功能正确性与内存安全性验证。通过构造适用于嵌套循环的循环不变式，成功描述了数组在递减间隔下的部分有序性和元素排列性质，并使用 Alt-Ergo、CVC4、Z3 等定理证明器完成了验证条件的自动证明。实验结果表明，所设计的循环不变式能够有效支撑希尔排序程序的正确性验证，验证过程充分体现了形式化方法在保证程序可靠性方面的优势。

参考文献

[1] 翟娟, 汤震浩, 李彬, 等. 常用循环摘要的自动生成方法及其应用[J]. 软件学报, 2017, 28(5): 1051-1069.

[2] de Gouw, S. (2019) Verifying OpenJDK's Sort Method for Generic Collections. *Journal of Automated Reasoning*, **62**, 93-162.

[3] Petrovic, D. (2014) Verification of Selection and Heap Sort Using Locales. Archive of Formal Proofs.

[4] Drămnesc, I., Jebelean, T. and Stratulat, S. (2024) Certification of Sorting Algorithms Using Theorema and Coq. In: *Lecture Notes in Computer Science*, Springer, 38-56. https://doi.org/10.1007/978-3-031-69042-6_3

[5] Hoare, C.A.R. (1969) An Axiomatic Basis for Computer Programming. *Program Verification*: *Fundamental Issues in Computer Science*, **14**, 83-96.

[6] Correnson, L., Dargaye, Z. and Pacalet, A. (2015) Frama-C's WP Plug in Manual. https://www.frama-c.com/download/frama-c-wp-manual.pdf

[7] Baudin, P., Cuoq, P., Filliâtre, J.C., *et al*. (2020) ANSI/ISO C Specification Language. Version 1.16.