第十八章　形式化证明

在设计开发一个操作系统的过程中，我们需要综合考虑并发、安全性、分布式、容错性、性能这些复杂的设计因素，为了降低设计实现操作系统的复杂性、提高操作系统鲁棒性，研究人员提出了模块化、抽象等方法。然而，随着操作系统变得越来越庞大和复杂，我们难以避免地往操作系统中不断引入漏洞，对于那些需要高可靠性的操作系统，漏洞可能会带来严重后果。

提到消除软件漏洞，我们首先想到的是软件测试，然而 1972 年图灵奖获得者 Edsger W. Dijkstra 曾说：“程序测试可用于显示错误的存在，但从不显示错误的缺席！” [2]。软件测试通过枚举可能的输入，往往只能测试有限路径下软件功能是否正常，而难以覆盖到所有可能的路径。

那么有没有一种方法能够从源头保证系统的正确性？答案便是**形式化证明**（Formal Verification），形式化证明本质上是一种数学方法，通过严格的数学证明保证覆盖到所有可能的情况。证明对我们来说并不是一个陌生的概念，我们从小就在与各种数学证明打交道，通过推导，由条件得出目标的成立。对于软件系统而言，我们在证明什么？

```c
unsigned char x;

void inc()
{
  x = x + 1;
}
```

代码片段 18.1: 一个简单的 C 语言函数

以一个简单的 C 函数`inc()`为例，证什么的答案可能多种多样。我们可以证明这个函数的功能，如`inc()`函数将全局变量`x`的值加一（事实上这个描述并不准确，没有考虑溢出的边界情况）；我们可以证明函数执行过程中，程序状态遵循的不变式，如`x`的值小于等于 256；我们可以证明各种系统级性质，

如活性、崩溃一致性；我们还可以证明非功能相关的性质，如运行时间、内存占用等。总结下来，对一个软件系统，证明什么取决于我们对它的期待，期待越多，我们能证明的就越多。

有了对软件的期待，证什么的问题仅回答了一半，我们还需要写出软件符合期待的数学表示，以此作为证明的目标。比如，为了证明程序符合不变式，我们可以将证明目标表述为：（1）不变式在初始时成立，并且（2）程序每执行一步，不变式依然成立。对于不同的期待，软件符合期待的表述也会不同，体现为不同的证明目标。

我们通常将软件称为**实现**（Implementation），对软件的期待称为**规约**（Specification），软件符合期待称为**正确性条件**（Correctness Condition），那么形式化证明则是指通过严格的数学**证明**（Proof）来保证实现符合规约，即证明正确性条件的成立。

近些年来，形式化证明技术蓬勃发展，许多系统的验证展示了形式化证明的可行性，其中标志性的项目包括 CompCert（被形式化证明的 C 编译器 [21]）、seL4（被形式化证明的微内核 [20]）等，他们不仅仅是简单的原型，而是实际的能够用于我们日常生活的软件，从而切实的提高计算机系统的可靠性。然而，在操作系统的证明上仍然有许多问题亟待解决，比如：

- 如何定义操作系统的规约?
- 如何有效地证明正确性条件的成立?
- 如何将形式化证明扩展到更大规模（工业级）的系统上?

本章内容的组织： 第18.1节介绍形式化方法的分类以及形式化证明基本的流程和概念，第18.2节介绍形式化规约的基本知识，第18.3和18.4节分别介绍两种常用的正确性条件霍尔三元组与精化关系。第18.5节介绍常见的证明手段与证明工具。第18.6节从证明的视角重新审视操作系统，介绍操作系统形式化证明的挑战与机遇。第18.7节对 seL4 的形式化证明进行案例分析。

18.1 什么是形式化证明？

本节主要知识点

- ❑ 形式化证明能保证绝对的正确吗？
- ❑ 形式化证明的流程是什么？

18.1.1 形式化方法概述

相比于形式化证明，**形式化方法**是一个更广义的概念，泛指基于数学的开发和证明技术。从笔者的经验来看，根据形式化程度的不同，形式化方法又可以按照应用场景分为**程序分析技术**、**模型检验**（Model Checking）和**定理证明**（Theorem Proving）。本章不对程序分析技术深入讨论。

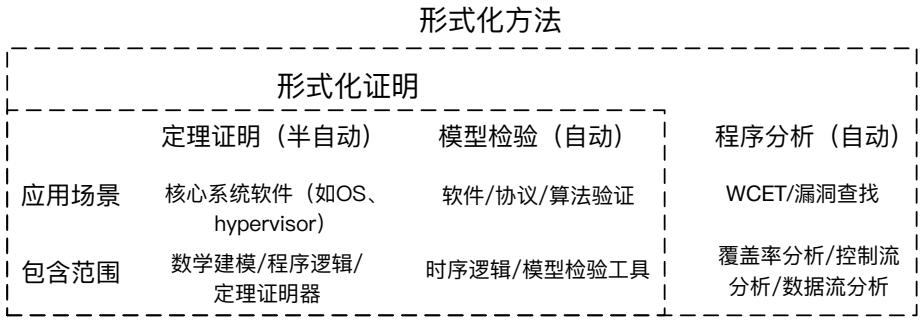

图 18.1: 形式化方法分类

程序分析技术

程序分析技术分为静态程序分析技术，如控制流分析、数据流分析、别名分析等，以及动态程序分析技术，它能够自动分析程序执行行为是否符合正确性等性质，有效保障软件质量，通常用于软件开发过程中的漏洞查找、最差执行时间分析（WCET，Worst Case Execution Time）等。然而对于实际大型程序，程序分析技术难以分析所有程序行为（如并发执行行为）的正确性，要想全面覆盖程序行为，则要用到形式化证明。

模型检验

形式化证明中的模型检验与定理证明都会证明系统实现完全符合规约，不同的是，模型检验以穷尽搜索为验证基础，通常用于软件、协议以及算法的验证，更依赖于时序逻辑，需要根据系统的状态转移构建一个有限状态图，并检验这个有限状态图是否是符合时序逻辑规约的模型，检验由机器自动完成。然而模型检验在穷尽搜索时会遇到状态空间爆炸的问题，无法在有限时间内返回，难以解决复杂的程序证明问题。

定理证明

定理证明则是以程序逻辑为基础，通过逻辑推理的方式验证程序的正确性，常用于证明复杂核心系统软件，如操作系统和虚拟机监视器等。定理证明的过程通常为交互式的形式，开发者在证明平台上输入证明策略，输出会呈现出证明目标的状态。对于复杂的问题，定理证明会带来较大的人工证明负担，但也并不是所有的定理都需要手动证明，开发者可以开发自动证明策略去解决一些具有固定模式的定理，也可以使用求解器（如微软开发的 Z3 [11]）去辅助证明。

在第18.2、18.3和18.4节中我们介绍的形式化证明的内容将偏向定理证明，但模型检验和定理证明有相通的基本概念，我们将在第18.5节涉及模型检验的内容。

18.1.2 形式化证明流程概述

通常的形式化证明流程发生在一个验证系统里面，将实现和规约在正确性条件下通过证明构建一致性。下面对这套流程的关键部分依次说明。

实现

我们熟悉的软件实现，是由某种编程语言书写的软件，如 C 程序或者 Java 程序，形式化证明中的实现也指代软件设计，如伪代码或模型，不需要可以实际执行。为了对实现进行证明，我们需要给出其精确的数学模型，即语义。简而言之，语言的语义定义了每一种语言构造执行的粒度以及对于状态的修改，状态包括各个变量的值、堆里面每个地址存储的数据等。如 C 语言中的x=x+1在操作语义上可以分为三步：第一步先计算出右值，即x+1的值；第二步，取出

左值的地址；第三步，将右值赋给左值的地址。并不是所有的语言都有精确的语义，要想对这些程序进行证明，我们需要对编程语言进行建模。

规约

规约泛指软件应该遵循的约束，表达了期待软件所具有的行为或属性，我们日常接触的规约通常是用自然语言描述的，比如 C 程序中的注释。我们将具有精确数学模型的规约称为 **形式化规约**（Formal Specification）[1]。规约不一定需要显式写出，有些证明方法只检查某些轻量级的规约（如越界检查），不要求开发者写出规约。通常规约被认为比实现更可信，因为规约往往抽象掉了复杂的软件实现，更容易理解以及发现漏洞。在第18.2节中我们会具体介绍形式化规约相关内容。

正确性条件

实现和规约相一致是我们证明的目标，而正确性条件则精确地定义了这个“一致”。简单来说，正确性条件定义了实现不会产生规约约束以外的行为，证明了正确性条件成立，对于实现正确性的依赖便转移到了规约的正确性上，而规约的正确性相对更容易保证。正确性条件是可信基的一部分，因此我们需要定义出令人信服的正确性条件，其需要直观的表达出软件为什么正确。在第18.3节和第18.4节中我们会具体介绍两种正确性条件的定义方法。

小思考

正确性条件定义了实现不会产生规约约束以外的行为，能否说正确性条件定义了实现与规约的行为相同？哪种说法更好？

证明

我们需要通过证明来说明正确性条件的成立，如果证明能通过，意味着实现和规约确实在正确性条件下一致。但证明并不意味着实现完全正确，即可能规约里面也包含错误的行为，或者在规约未能描述到的方面，实现包含错误。证明不通过，则表明两者的不一致，需要根据证明无法通过的地方去检查实现或者规约，进行相应修改以使得证明能够继续下去。

[1]为了行文简洁，后文中的“规约”也会指代“形式化规约”，可根据上下文确定具体含义

我们熟悉的证明手段是基于纸笔证明，然而这种证明方法容易产生纰漏，即证明本身出错。为了保证证明过程的可靠性，需要证明过程能够被机器检查，即写下的证明背后又有一个证明检查器，来检查证明是否符合逻辑，这个证明检查器基于专家开发的一套完善的逻辑系统，这套逻辑系统在数学上被确立了可靠性，从而从源头上保证了证明的可靠性。有的证明方法可以自动化证明，如模型检验工具，不需要开发者手动证明，在第18.5节我们会介绍证明手段的分类以及常用的证明工具。

验证系统

支持将上述所有要素结合起来，完成证明过程的系统我们称为验证系统。一个验证系统需要对实现的进行数学建模，需要告诉我们如何定义规约，并提供正确性条件，在定理证明中，验证系统也需要实现在某个特定证明平台上，如 Coq 交互式定理证明器 [1]，并通常包含某种程序逻辑，用于简化使用者的证明目标，使用者只需要按照验证系统的要求完成验证，程序逻辑可以推导出正确性条件的成立。不同的验证系统具有不同的功能，适用的问题域也不相同。好的验证系统通常包含对于问题域的证明自动化的支持，简化使用者的证明负担，我们将在后面介绍更多验证系统相关的内容。

18.2 规约概述

本节主要知识点

- ❑ 如何衡量一个形式化规约的好坏？
- ❑ 如何写出形式化规约？
- ❑ 如何保证形式化规约的正确？

18.2.1 为什么需要规约？

现代硬件和软件非常复杂，抽象是应对这种复杂性的重要原则之一。一个复杂软件可以分解为一系列模块，每一个模块经过封装后仅提供一系列接口和外界进行交互。这些接口的功能和使用方法通过各种形式的规约来说明，如说明函数功能的注释、安全检查的断言、接口的说明文档以及通过逻辑表达式来编写的形式化规约。

规约将系统开发者和用户连接起来。简洁清晰的规约能让用户快速熟悉系统功能，当用户产生新的需求时，也能通过规约快速反馈给开发者。

规约对提高系统的鲁棒性十分有益。由于规约需要准确描述系统功能，编写规约的同时会促使开发者对于系统的内部原理有更加深刻的理解，考虑到系统在不同环境下所有可能的行为，有可能发现系统在设计实现方面的漏洞或者不完善的地方，从而对系统进行改进，更重要的是，证明实现与规约的一致性能给实现提供更高的保障。

18.2.2 好的规约应有的特性

尽管规约有着很多益处，但编写出好的规约，尤其是能够用于形式化证明的形式化规约，却是一件很有挑战性的事情。我们需要更加了解好的规约应该具备的特性来更好的编写规约。DeepSpec 研究组[2]总结出了一类最有用的规约，并称之为**深度规约**，深度规约应当具有**丰富性**、**形式化**、**实时性**和**双向性**的特性 [4]。

- 丰富性：能详细描述复杂的软件行为。
- 形式化：以数学符号编写，并带有清晰的语义。
- 实时性：规约通过证明实时与实际代码（不仅仅是模型）保持一致。
- 双向性：模块的规约能参与到模块实现和模块调用者双向的证明中。

表 18.1: 不同规约在丰富性、形式化、实时性和双向性方面的对比 [4]

规约类型	丰富性	形式化	实时性	双向性
完整的非形式化描述（C 说明文档）	✓	✗	✗	✗
“传统”的规约语言（如 Alloy，Z）	✗	✓	✗	✗
类型系统等轻量级形式化规约	✗	✓	✓	✗
深度规约	✓	✓	✓	✓

表18.1从丰富性、形式化、实时性和双向性方面对比了常见的几种规约。

翻阅说明文档是最常用的获取信息的方式，其自然语言的描述更容易理解，并能提供详细的信息；但说明文档具有以下缺陷，首先，自然语言是非形

[2]DeepSpec 研究组是美国麻省理工学院，宾夕法尼亚大学，耶鲁大学和普林斯顿大学联合成立的，致力于构建完全形式化验证的硬件和软件系统。

式化的，容易产生模糊的表达甚至是歧义，其次，说明文档独立于实现，不具备实时性，在软件迭代过程中，有些文档内容可能已经过时但却没有及时修正，文档内容不保证与实现相符，并且不同地方的文档描述可能自相矛盾。

有些工具提供了形式化定义规约的方法，如形式化语言 Z [31]、Alloy [16] 等，它们具有精确的语义，定义的规约不会产生歧义；然而这些规约语言只描述了系统设计，而没有与实际的软件联系起来，即使能保证系统设计正确，却没办法保证软件不包含漏洞。

类型系统是一种最广泛运用的形式化规约，类型检查能及早发现程序中的错误，动态类型检查还能发现数组越界访问等问题；然而类型系统缺乏丰富性，无法描述函数功能和性质。

除此之外，这些规约都不具备双向性，无法被用于模块调用者的证明中。更具体的说，假如模块实现的行为已经被证明了与规约一致，那么调用者可以直接按照调用规约的行为进行推理证明，而不会有遗漏。由于规约通常相对于实现有所简化（数据结构和算法的简化，而非行为的简化），上层调用者直接使用下层实现的规约进行证明会更加方便，以此类推，我们可以不断搭建新的上层，每一个层次的证明都不需要再考虑复杂的下层实现，由此我们可以搭建一个完全形式化证明的软件栈。

总结下来，这些规约都不可否认能起到极大的帮助，但它们在这些特性方面的缺乏使其无法发挥最大的效用。

18.2.3 形式化规约长什么样？

满足上述四个特性的规约提出了以下一些要求，首先，规约语言需要形式化定义，其次，规约语言不能仅描述模型，还要能描述实现，这样才能将实现和规约连接起来，最后，要有一个环境能证明实现和规约一致。Coq 证明助手 [1] 正是这样一个平台，它提供了一个能描述实现和规约的形式化语言以及交互式的证明环境，很多形式化证明的项目都是基于 Coq 完成的，其中包括 CompCert [21]，一个形式化证明的 C 编译器。

我们将以 CompCert 为例介绍形式化规约写成什么样。CompCert 的目的是保证编译过程的正确性，即编译出来的汇编代码相比于 C 代码不会引入新的行为，因此 CompCert 需要形式化建模 C 语言以及汇编语言。

代码片段18.2是简化后的 CompCert 的一段 C 语言规约，这一段规约是用 Coq 的规约语言 Gallina 编写的，Gallina 是一个函数式编程语言，函数式有两重含义，一是每一个程序所有的效果都通过输出呈现，这样每个程序都能简单看作一个由输入到输出的函数，与之相对的是，如 C 语言除了返回值以

外，还会在程序中修改内存和变量状态；二是函数本身也是值，可以作为参数、返回值或被包括进数据结构。除此之外 Gallina 还支持方便的模式匹配、多态、依赖类型等特性，这些使得其作为规约语言具有强大的表达力。

```
(*definition of int is omitted*)

Inductive expr : Type :=
  | Econst_int : int -> expr.

Inductive val : Type :=
  | Vint : int -> val.

Definition eval_expr (e : expr) (v : val) : Prop :=
  match e with
  |  Econst_int i => v = Vint i
  end.
```

代码片段 18.2: 使用 Coq 描述 C 语言表达式估值的规约

这段代码形式化定义了 C 语言的表达式`expr`、表达式的值`val`以及表达式的求值函数`eval_expr`。Coq 中可以用`Inductive`关键词来定义类型，后面紧跟类型名字，如`expr`和`var`，它们的类型是`Type`，Coq 自带的基本类型之一，`A : B`的含义为`A`的类型是`B`，`:=`后面是这个类型的构造方法，不同的构造方法通过`|`分隔。

具体来说，C 语言表达式可能是整型`Econst_int`、变量、表达式二元运算等，其中`Econst_int`接受一个整数`int`，来构造出一个表达式类型，表达式类型的其它构造方式在这里省略了。表达式的值同样可能是整数`Vint`、浮点、指针等，`Vint`接受一个整数，来构造出一个值的类型。

`Definition`关键词用于定义函数，其后紧跟函数名称`eval_expr`和一系列参数，最后一个`:`后是函数返回值的类型，`Prop`代表 Coq 中命题相关的类型，其值为`True`或者`False`，代表命题的真假。`eval_expr`的实现中使用`match`对表达式进行类型匹配，假如匹配到整形表达式`Econst_int i`，那么其对应的值必须是整形值`Vint i`，并且两者值`i`相同。

给出的形式化规约示例就像一段软件程序，它也是用特定的语言编写，符合其语法规则，并且这段规约也需要修改，测试和维护。这段规约具有丰富性，我们可以继续写出所有的表达式和值的构造方法以及他们在求值函数中的对应方式，从而详细描述表达式估值的规约；这段规约是形式化的，相比于自然语言，Coq 编写的程序具有清晰的语义；这段规约具有实时性，在 Coq 中会实时的检查实现和规约一致性的证明，一旦规约或者实现产生修改，相应的证

明也无法通过，除非两者重新一致；这段规约也具有双向性，一方面通过证明连接编译器的实现，另一方面面向编译器的使用者。

18.2.4 形式化规约来自哪里？

一段精确的形式化规约，也需要开发者反复打磨。通常我们可以有多种手段来写出正确的形式化规约：

- **自然语言文档**：虽然自然语言不够精确，但标准文档、用户手册、程序员注释等自然语言文档包含着大量信息，这些信息捕获了不同开发阶段的人员的理解，并能成为形式化规约的基础。
- **软件实现**：形式化规约最终还是源于对于软件的理解，通过加深对软件实现的理解，开发者更容易写出准确的规约。
- **模型检验**：规约描述出了软件的运行模型，我们可以通过模型检验的手段来检验模型是否符合系统性质，从而增强规约的可靠性。
- **形式化证明**：最终在证明实现和规约一致性的环节，通过两者的相互印证，我们能够最大程度地发现规约及实现中的漏洞。

18.2.5 形式化规约出错了怎么办？

程序员编写规约的时候可能因为笔误或者考虑不周出错。所以在进行形式化验证的时候如果发现证明失败，不一定是软件实现有错误，也有可能是形式化规约写错了。这是因为编写好的形式化规约是一件很有挑战性的事情，如果编写者对于系统的理解不够深刻，很有可能写出错误的规约或者不准确的规约。

形式化规约的双向性可以帮助发现规约中的问题，在与实现和调用者的双向验证中保证规约的正确性。

例如代码片段18.3中，函数`get_item`调用函数`get_index`得到一个随机生成的数组下标，之后返回这个下标对应的数组`buf`中的元素。假如一开始将函数`get_index`的规约错误的定义为返回值是一个大于等于 0 的数字，在证明`get_index`的实现符合规约时没有问题，但是在证明函数`get_item`的过程中会失败，因为`get_item`要求`get_index`不仅要大于等于 0，而且要小于数组`buf`的长度，这时候就会发现函数`get_index`的规约有问题，包含的信息不够准确，所以要修改规约为返回值是一个大于等于 0 且小于 4 的整数，这时候函数`get_item`和函数`get_index`才能验证通过。

```c
//函数 A 随机生成一个数组下标
int get_index(void)
{
  return rand()%4;
}

int buf[] = {1, 2, 3, 4};

//函数 B 随机返回数组中的一个元素
int get_item(void)
{
  int index = get_index();
  int res = buf[index];
  return res;
}
```

代码片段 18.3: 错误形式化规约实例

18.2.6 形式化规约小结

通过本节的介绍，我们知道了形式化规约如何区分于一般的规约，提供更高的保障。同时我们也须认识到，规约不仅是软件的重实现，还能包括许多实现中隐式蕴涵的属性，写出充分刻画期望属性的形式化规约是项目中非常有挑战的部分。最常见的规约包含了**功能正确性**，即软件实现能正确完成期待的功能，不会出现除零、整数溢出、越界访问等问题；对于像操作系统的大型软件而言，上层应用往往期待更多的性质，如隔离性、活性、崩溃一致性等，以及系统状态的各种不变式。很多性质我们知道该如何进行刻画，描述在规约里，但也有很多性质我们还不知道如何刻画。有的性质虽然能够通过规约描述出来，但不同的描述方式证明的难易程度也有所不同，因此需要寻找最适合证明的规约。在后面的章节中，我们将通过更多例子介绍如何定义形式化规约。

18.3 霍尔三元组

本节主要知识点

- ❑ 霍尔三元组的规约是什么？
- ❑ 霍尔三元组如何定义正确性？
- ❑ 如何使用霍尔逻辑进行证明？

对于不同的规约，实现符合规约的正确性条件的定义也会不同，在这里我们介绍证明功能正确性时最常用到的两种正确性条件的定义方式：本节的**霍尔三元组**（Hoare Triple）和下一节的**精化关系**（Refinement）。

18.3.1 霍尔三元组的规约

描述一段程序的功能时，最常见的办法是描述它接收什么输入，执行完返回什么输出以及完成的功能，这种由前到后的描述方式即是霍尔三元组的想法。霍尔三元组的形式如下：

$$\{P\}\ C\ \{Q\}$$

其中 P 和 Q 都是**断言**（Assertion），用来刻画程序的**状态**（State），我们通常称 P 为前置条件，描述程序执行前的状态，而将 Q 称为后置条件，描述程序执行结束后的状态，断言通常采用一阶逻辑表示。为了行文简便，在本章剩余的部分中，我们都假设程序状态仅包含变量到值的映射，其中变量用字符串表示，如 x，y 等，值为整形，如 1，2 等，那么断言的例子包括 $x = 1 \wedge y = 2$ 以及 $\exists n, x = n$。C 表示**指令**（Command），本节考虑一个简化的指令集，包含赋值语句（$V = E$）、顺序语句（$C_1;\ C_2$）、选择语句（$if\ B\ then\ C_1\ else\ C_2$）以及循环语句（$while(B)\ C$）。

P 和 Q 是霍尔三元组的规约，表达了执行 C 这一段程序时，期待的状态变化。例如对霍尔三元组 $\{x = n\}\ x = x + 1;\ x = x + 1\ \{x = n + 2\}$，前置条件 $\{x = n\}$ 以及后置条件 $\{x = n + 2\}$ 表达了我们对 $x = x + 1;\ x = x + 1$ 的期待，即这段程序在串行环境下会将 x 的值加 2。

18.3.2 霍尔三元组的含义

那么执行 C 产生的效果和规约的描述是否一致？严格地证明这一点就需要给出霍尔三元组的含义，即连接实现和规约的正确性条件。霍尔三元组的含义是：当程序从某一状态 S_1 开始执行，并且 S_1 满足前置条件 P，如果程序 C 执行结束后，假设结束时的程序状态为 S_2，那么末状态 S_2 将满足后置条件 Q。霍尔三元组按照程序的执行语义直观地给出了实现和规约一致的含义。

有两点需要说明，一是这种正确性定义仅仅适用于串行条件下，如对第18.3.1小节的例子，其表达了当 x 的初始值是 n，执行 $x = x + 1;\ x = x + 1$ 之后，x 的值会变成 $n + 2$，在一个并发的执行环境中这个霍尔三元组却可能不成立，其他线程可能也在执行 $x = x + 1$ 操作，使得后置状态不满足 $x = n + 2$。

究其原因在于这个正确性条件的定义中没有考虑可能的并发环境的干扰。对于并发环境下的正确性定义我们会在第18.4节涉及到。

二是这个正确性描述仅仅描述了**部分正确性**（Partial Correctness），其主动假设了 C 会执行结束时，后置状态应当满足的条件。对于 C 不会终止的情况（如 $while(true);$），这个正确性条件会无意义的成立，因为 C 不会终止的情况与假设的前提相矛盾。这个正确性条件的部分性体现在其仅仅刻画了程序的功能正确性，而没有刻画终止性。只有当正确性条件也刻画了终止性的时候我们才称之为**完全正确性**（Total Correctness）。由此可见，定义正确性条件的时候需要特别的小心，可能我们定义的正确性条件并不（够）正确。

18.3.3 霍尔三元组的证明：推理规则🌶

为了证明任意给定霍尔三元组的成立，我们可以利用一套推理规则对程序进行推理证明，例如有如下的推理规则：

$$\frac{}{\{Q[E/x]\}\ x = E\ \{Q\}}\ (ASSN) \qquad \frac{\{P\}\ C_1\ \{R\} \quad \{R\}\ C_2\ \{Q\}}{\{P\}\ C_1;\ C_2\ \{Q\}}\ (SEQ)$$

$$\frac{P \Rightarrow P_1 \quad \{P_1\}\ C\ \{Q_1\} \quad Q_1 \Rightarrow Q}{\{P\}\ C\ \{Q\}}\ (CONSEQ)$$

推理规则的含义是横线上方的命题能够推导出横线下方的命题。比如在顺序规则（SEQ）中，当 $\{P\}\ C_1\ \{R\}$ 和 $\{R\}\ C_2\ \{Q\}$ 能够证明成立时，那么 $\{P\}\ C_1;\ C_2\ \{Q\}$ 成立，其作用是将顺序指令的证明拆解为 C_1 和 C_2 分别的证明。同理在因果规则（$CONSEQ$）中，当 $P \Rightarrow P_1$、$\{P_1\}\ C\ \{Q_1\}$ 和 $Q_1 \Rightarrow Q$ 成立时，$\{P\}\ C\ \{Q\}$ 能够成立，这条规则说明了假如一个霍尔三元组成立，强化其前置条件，弱化其后置条件得到的霍尔三元组依然成立。

赋值规则（$ASSN$）稍微不那么直观，在赋值规则中，E 是一个表达式，x 是一个变量，其表示的含义为如果初始程序状态满足 $Q[E/x]$，那么完成赋值的后置状态满足 Q。其中 $Q[E/x]$ 表示将命题 Q 中 x 的值用表达式 E 的值代替，这条推理规则采用“倒推”的方式给出，即要想赋值的后置 Q 成立，那么假设赋值在前置中已完成了，这样产生的前置也应该成立。

小思考

或许你会觉得这种倒推式的推理规则有些不符合直觉，赋值语句其实也有顺推式的推理规则，该写成什么样呢?

证明霍尔三元组可以使用对应的推理规则，将证明目标从横线下方的命题形式转移为横线上方的命题形式，接着对现有的证明目标继续应用推理规则直到所有的命题得到证明。

18.3.4 一个霍尔三元组证明的例子🌶

我们可以形式化证明第18.3.1小节的例子。

- 目标 1：$\{x = n\}\ x = x + 1;\ x = x + 1\ \{x = n + 2\}$，应用顺序规则得到子目标 2 和 3
- 目标 2：$\{x = n\}\ x = x + 1\ \{?R\}$

由于顺序规则的前提里没有规定 R 的形式，所以会产生目标 2 中的 $?R$，代表中间状态满足的断言不确定，需要我们在证明中实例化出来。根据我们对程序的理解我们可以将 R 确定为 $x = n + 1$。

- 目标 2：$\{x = n\}\ x = x + 1\ \{x = n + 1\}$，应用因果规则得到子目标 2.1、2.2 和 2.3
- 目标 2.1：$x = n \Rightarrow ?P_1$
- 目标 2.2：$\{?P_1\}\ x = x + 1\ \{?Q_1\}$
- 目标 2.3：$?Q_1 \Rightarrow x = n + 1$

其中 P_1 和 Q_1 需要实例化，由于赋值规则对前置条件的形式做了限定，对后置条件没有限制，所以我们需要先实例化后置条件 Q_1 为 $x = n + 1$，再对目标 2.2 应用赋值规则后，就可以得到 P_1。

- 目标 2.3：$x = n + 1 \Rightarrow x = n + 1$，成立
- 目标 2.2：$\{?P_1\}\ x = x + 1\ \{x = n + 1\}$，应用赋值规则得证，并实例化 P_1 为 $(x = n + 1)[(x + 1)/x]$
- 目标 2.1：$x = n \Rightarrow (x = n + 1)[(x + 1)/x]$，将 $(x = n+1)$ 中的 x 用 $x+1$ 替换，可化简为 $x = n \Rightarrow x + 1 = n + 1$，进一步化简为 $x = n \Rightarrow x = n$，得证

同理我们可以说明目标 3：$\{x = n + 1\}\ x = x + 1\ \{x = n + 2\}$ 的成立。

18.3.5 推理规则的可靠性

在前一小节的例子中，我们只需要按照推理规则进行证明，但推理规则也是很容易写错的，从而导致有些可以证明的程序现在无法证明或是不能证明的程序变得可以证明，因此我们需要额外证明符合推理规则的霍尔三元组，其也符合霍尔三元组的正确性定义，这个证明被称为**可靠性证明**（Soundness Proof），第18.3.3小节介绍的推理规则都是经过了可靠性证明的，推理规则加上可靠性证明形成的整个逻辑系统被称为**霍尔逻辑**（Hoare Logic）。

小思考

为什么需要额外提出一套推理规则进行证明，而不是直接证明霍尔三元组的正确性定义？

18.3.6 霍尔三元组小结

本节介绍了霍尔三元组的规约及其定义程序正确性的方式，并以一个例子呈现了使用霍尔逻辑证明的过程。本节的叙述采用了半形式化的方式，对于状态、指令、断言、语义及正确性条件这些形式化概念的介绍尽量采用了自然语言，对于推理规则和例子证明部分采用了偏形式化的方式，为了使读者不陷于形式化细节的同时又能体会到证明过程的严格。感兴趣的读者可以参考Software Foundation [29, 28] 对霍尔逻辑的形式化实现。

霍尔逻辑是程序形式化证明的基础理论，在后续的形式化证明研究中，人们在霍尔逻辑的基础上继续对其进行扩展和完善，使其具有更强的表达力，例如**分离逻辑** [30]（Separation Logic）扩展了霍尔逻辑使其支持指针程序以及局部推理，**并发分离逻辑** [8]（Concurrent Separation Logic）进一步扩展了分离逻辑使其支持并发程序的证明。

18.4 精化关系

本节主要知识点

- ❑ 霍尔三元组有哪些局限？
- ❑ 精化关系的规约是怎样的？
- ❑ 精化关系如何定义正确性？

精化关系（Refinement）是一种建立在抽象程序与具体程序之间的关系，其从程序行为的角度定义了两个程序的一致性，也广泛地被用于正确性条件。

18.4.1 霍尔三元组的局限

霍尔三元组从程序执行逻辑的角度定义了规约，然而霍尔三元组表达能力有限，不适用于某些程序的规约。

```
inc() {
   int done = 0, tmp;
   while(!done){
      tmp = cnt;
      done = cas(&cnt, tmp, tmp+1)
   }
}
```

(a) 错误的规约：
{cnt = N} inc() {cnt = N+1}

(b) 较弱的规约：
{∃N. cnt = N} inc() {∃N. cnt = N}

图 18.2: 霍尔三元组的局限

比如对于图18.2所示的并发程序，`inc()`函数通过原子指令`cas`对共享变量`cnt`完成加一操作。图18.2(a) 中所示的规约是错误的，前置和后置条件都要求`cnt`等于某个常值，这在并发环境下不成立，因为除了考虑当前程序的执行，还要考虑环境中其它程序的执行，它们可能会修改`cnt`的值。我们可以将规约写为图18.2(b) 那样，此时`N`不再是常值，而是一个被存在量词约束的变量，这样即使环境在不断修改`cnt`的值，断言总能够根据当前`cnt`的值实例化变量`N`；然而（b）中所示的规约太弱了，不能表达出`inc()`函数的功能，假设有一个将`cnt`值加2的函数，同样可以使用（b）中的规约。

另外，对于某些不终止的程序，如`while(true);`，此时的程序后置我们可以写为 $False$，意味着后置可以是任意状态，因为程序无法执行到后置，这种无限循环常见于某些服务器端的处理代码，以此来不断相应用户请求，使用霍尔三元组同样无法描述这类程序的规约。

18.4.2 行为序列集合

除了从执行逻辑的角度定义程序的规约，我们还可以从程序行为的角度定义规约，一段程序的功能可以通过使用这段程序的人所能看到的行为来衡量，给定某个初始状态，将一段程序（包括并发程序）在所有可能执行，包括部分执行的情况下产生的行为序列都记录下来，会得到程序的**行为序列集合**。

关于行为序列集合的定义，有两点需要说明，一是其并没有明确定义程序行为，对于不同的观察者，所能看到的程序行为也不同，对内部观察者，可以看到程序每一次访存或是程序执行的操作，对外部观察者，看到的是 **外部可观测行为**（Externally Observable Behavior），如程序输出以及程序的异常中止等。从定义程序规约的角度考虑，本节将程序行为选取为外部可观测行为，因为它屏蔽了更多实现的细节。

二是这里考虑的行为序列是程序前缀执行产生的行为序列，即程序在执行 0 步、1 步或 n 步产生的行为序列，由此构成的行为序列集合是**前缀闭合**（Prefix Closed）的集合。

小思考

行为序列其实也可以定义为完整执行产生的，这种定义与前缀执行定义的适用场景有何不同?

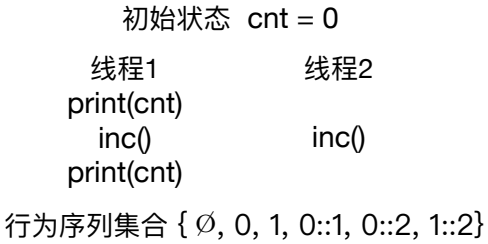

图 18.3: 行为序列集合示例

如图18.3中有两个线程并发执行，调用的`inc()`函数采用图18.2中的实现，在`cnt`初始为`0`的情况下，我们观测程序产生的行为序列，即`print`能够输出的值，为了表达方便，将行为序列 l 记为 $\varnothing$（空）、a（仅有一个元素）或者 $a :: l$（含多个元素）。根据并发执行序列的不同，这样一个程序执行前缀操作后共能产生 6 种行为。具体而言，分为`inc()`的效果在首个`print(cnt)`前完成（输出 $1 :: 2$），在两个`print(cnt)`中间完成（输出 $0 :: 2$）和在后一个`print(cnt)`后完成（输出 $0 :: 1$）。另外考虑没有输出（$\varnothing$）和只执行首个`print(cnt)`（输出 0 或输出 1）的行为。虽然这六种行为并不那么有意义，我们依然可以从中了解到这段程序拥有的功能。

18.4.3 精化关系的规约

行为序列集合能够用于衡量程序的功能，却不适合直接用于规约，因为行为序列集合不够直观的反应程序功能并且也难以枚举完整。但行为序列集合却提供了建立两个程序一致性的思路，以此来用作正确性条件。在介绍精化关系的含义之前，我们先介绍精化关系的规约，其需要包含三个部分：一是抽象状态的描述，二是定义在抽象状态上的抽象操作，三是抽象状态和具体状态的**映射关系**（Refinement Mapping）。下面依次说明。

术语： 我们除了称实现和规约分别为具体和抽象，还有一些其它的说法，如目标和源，底层和高层，这些都是等价的说法。

抽象状态的引入是对具体状态的一个简化或者精炼，是使用者关心的状态，例如对于一个高效键值查找的实现（如用二叉树或者哈希表实现），其抽象状态只需要被认为是一个键到值的映射，这就足够代表这一段程序的功能。

抽象操作是定义在抽象状态上的操作，是暴露给用户的接口，抽象操作也是由程序语言实现，其可以是和具体程序相同的语言，或者是自定义的更适合操作抽象状态的语言。抽象操作仅仅用作规约，不会用于具体执行。抽象操作的接口需要与具体操作的接口相一致，即参数和返回值的类型在两个程序语言中要对应。

映射关系定义了怎样的具体状态和抽象状态是等价的。例如对于图18.2中的`inc()`函数，我们可以实现`INC()`程序作为其精化规约，其抽象状态为`CNT`，`INC()`的实现为`<CNT=CNT+1>`，在`INC()`的实现中我们使用`<C>`表达C是原子执行的，因此`INC()`是一个原子的将`CNT`加一的函数，其将非原子的`inc()`函数进行了抽象。虽然`INC()`函数作用于`CNT`这个变量，但只要我们将映射关系定义为`cnt`的值对应于`CNT`的值，就能够将`inc()`与`INC()`联系起来。

18.4.4 精化关系的含义与证明

从任意等价的初始状态开始执行，实现的行为序列集合不多于规约的行为序列集合，我们将实现和规约的这种关系称为精化关系，即实现**精化**（Refine）规约，或者说实现是规约的一个精化。精化关系的含义意味着实现被规约所描述，即实现符合规约，因而精化关系可以用于形式化证明中的正确性条件。

为什么精化关系定义为具体行为不多于抽象行为？这并不意味着具体程序的功能有遗漏，而是更多时候规约会定义的更宽松一些，给予实现更多的自由性，例如在 POSIX 标准中，定义了文件系统的规约，其中对于`rename`里的边

界情况说到，假如rename将一个文件改名成一个已存在非空目录，这个操作需要返回错误符EISDIR，EEXIST或者ENOTEMPTY中的一个。即规约允许这三种行为，而具体实现由于确定性，只会产生其中一种行为，换句话说，规约定义了正确行为的集合，精化关系保证实现的行为落在这个集合里面。

许多证明问题需要建立两个程序间行为上的关系，都可以使用精化关系作为正确性条件，比如精化关系可以用于验证编译程序的正确性或者是验证程序优化的正确性，除了能刻画串行程序，也能用于并发对象（如文件系统、操作系统等）的正确性。

直接根据精化关系的定义，收集实现和规约的行为序列集合进行证明比较复杂，因为收集行为序列集合需要考虑程序所有可能的执行情况。另一种证明思路是，行为序列是由程序的执行产生的，对于实现能够产生的任一行为序列，规约都要通过某种执行方式产生对应的行为序列，因此，我们可以将行为序列的证明转化为程序执行的证明，即实现每执行一步，证明规约都可以对应的执行一步，这里的对应有状态对应和行为对应两层含义。这种证明的思想即是**模拟**（Simulation），模拟是证明精化关系最常用的手段。

18.4.5 精化关系小结

精化关系被广泛用于系统的正确性条件，因为其概念简单，且方便扩展。前文介绍的精化关系能够有效地刻画功能正确性，因为我们观察的行为能够包括程序的输出以及程序是否出错。这样一来，当规约的行为都满足功能正确性时，证明精化关系后，功能正确性能够传递给具体实现。但除了功能正确性，我们还关心其他的性质，如终止性、安全性、崩溃一致性等。这时我们就需要扩展精化关系来支持这些性质的刻画，有研究工作基于精化关系的定义进行扩展来支持终止性 [22]、安全性 [10] 和崩溃一致性 [9]

18.5 证明概述

本节主要知识点

- ❑ 常见证明手段有哪些？
- ❑ 常用的证明工具有哪些？
- ❑ 不同的证明工具各自有哪些优点和缺点？

明确正确性条件之后，便进入了证明的阶段，我们熟悉的证明手段是纸笔证明，但纸笔证明依然需要通过人工确认其可靠性，就像完成的证明题需要老师的批改，这使得证明结果依然受到人为疏漏的影响。形式化证明的特点之一是**机器可检查的证明**（Machine Checkable Proof），即写下的证明可以交给证明检查程序检查，保证证明过程的可靠性。另外还有一些工具可以自动证明定理，减少了编写证明过程的负担。在本节中我们会根据证明自动化程度对证明手段进行分类，并分别介绍各类证明手段及其常用的工具。

18.5.1 证明手段的分类

目前的证明手段按照自动化程度由高到低可以分为三类：**一键证明**（Push-button Verification）、**半自动证明**（Auto-active Verification）和**交互式证明**（Interactive Verification）。

一键证明自动化程度最高，只要求使用者定义规约，证明过程全部自动化，非常方便。但是，目前的一键证明技术有一些问题仍待解决。首先，它无法处理无界循环，无界循环是指最大循环次数没有上界的循环。其次，由于这些技术会遍历程序可能的状态，如果程序中的分支太多，或者程序中存在并发，那么会发生状态爆炸的问题。另外，这些技术通常依靠 SMT 求解器实现自动化，而求解器的求解能力是有限的，如果求解器发生求解超时或者无法求解的情况，那么这些技术就无能为力。常见的一键证明技术包括**符号执行**（Symbolic Execution）和模型检验等。

半自动证明不仅需要使用者定义规约，有时还需要使用者在代码中插入标记或者定义辅助定理。虽然半自动证明的自动化程度不如一键证明，但是因为证明过程可以得到使用者的帮助，所以证明能力比一键证明要强大。例如，一键证明不能处理的无界循环，半自动证明技术可以处理。常见的半自动证明技术包括**最弱前置条件**（Weakest Precondition）等。

交互式证明的自动化程序最低，需要使用者定义规约并编写大量证明代码，但是另一方面，交互式证明的证明能力是最强大的。常用的交互式证明工具包括 Coq 和 Isabelle/HOL 等。

18.5.2 一键证明

本节主要介绍一键证明的相关技术和工具。一键证明技术通常依靠 SMT 求解器实现自动化，所以本节首先介绍什么是 SMT 求解器，之后会介绍常见的一键证明技术模型检验和符号执行。

SMT 求解器

许多的形式化规约可以使用一阶逻辑表达式定义，SMT 求解器是用于自动解决一阶逻辑表达式可满足性问题的工具。一阶逻辑表达式的可满足性问题是指判断给定的一阶逻辑表达式能否在某些情况下成立。

```
context cxt;
expr a = cxt.int_const("a");
expr b = cxt.int_const("b");
solver s(cxt);

//判断是否存在整数 a 和 b 使得 a>=0 和 b<a 同时成立
s.add(a >= 0);
s.add(b < a);

//输出判断结果 sat
cout << s.check();
```

代码片段 18.4: SMT 求解器程序示例

例如在代码片段18.4中，a 和 b 是两个整数，程序调用 SMT 求解器判断一阶逻辑表达式 $a \geq 0 \wedge b < a$ 是否是可满足的，也就是说是否存在整数 a 和 b 使 $a \geq 0 \wedge b < a$ 成立，`s.check()`会返回这个表达式是否是可满足的，由于 $a \geq 0 \wedge b < a$ 明显是可满足的，所以这里的输出结果就是 sat。

由于 SMT 求解器的输入是逻辑表达式而不是实际的程序，所以它一般和其它技术配合使用。例如后面讲到的符号执行先对真实程序进行处理，然后生成逻辑表达式交给 SMT 求解，实现自动化证明。目前的自动化证明工具基本都是依靠 SMT 求解器实现自动化。

一阶逻辑表达式的可满足性是一个非常困难的问题，所以对于有些复杂的一阶逻辑表达式，SMT 求解器会耗费很长时间进行求解，或者根本无法判定表达式是否可满足。目前常用的 SMT 求解器包括 CVC4 和 Z3 等。

模型检验

由于较高的自动化程度，模型检验被广泛的用于工业界中软件和硬件的证明，其由三个主要部分组成：模型、规约和算法。

模型检验并不直接证明对象的实现代码，而是对对象建模得到的模型进行证明，因此这里的模型即是证明中的实现。模型通常被表示为 **状态转移图**（State-transition Graph）的形式，例如图18.4是一个简单的状态转移图的例子，

圆圈代表了红绿灯系统所处的可能的状态，而箭头代表了状态之间如何转移。**时序逻辑**（Temporal Logic）是一个描述时间相关性质的逻辑，很适合用于描述状态转移序列，被用作书写模型检验的规约。对图18.4的例子，使用时序逻辑可以表述以下性质，如灯总会变成绿色，红灯下一个颜色是绿色等。证明环节由一个模型检验的算法通过穷尽搜索自动证明模型是否符合时序逻辑的规约，如果不符合则会给出反例。

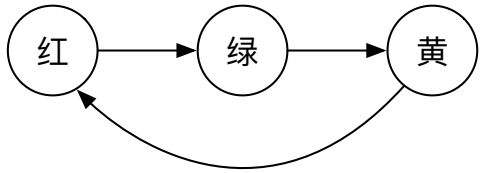

图 18.4: 红绿灯的状态转移图

模型检验的优势在于完全自动化的证明过程，并能够处理并发的证明。但模型检验会遇到**状态空间爆炸**（State Explosion）的问题，即在穷尽搜索的时候由于状态过多而无法在有限时间内返回，尽管有许多工作在尝试应对这个难题并取得了进步，但很有可能状态空间爆炸的问题仍然会是模型检验的局限。

符号执行

符号执行可以用于生成测试用例，由于它能遍历每一条执行路径，所以也被用于形式化证明。符号执行的基本想法是使用符号而不是具体的值来表示程序状态，也就是说程序中变量的值都是符号，这样就能考虑在任意输入的情况下程序所有可能的行为。

```
1 void func(int a, int b) {
2   if(a > 0)
3     a = -b;
4   else
5     b = 1;
6   assert(a + b < 1);
7 }
```

代码片段 18.5: 符号执行程序示例

下面以代码片段18.5为例具体讲解符号执行的流程，符号执行会判断代码中的`assert(a + b < 1)`是否会报错。符号执行过程中会维护 3 个状态，包括当前将要执行的语句、执行这个语句前每个变量的值和执行到当前语句要满

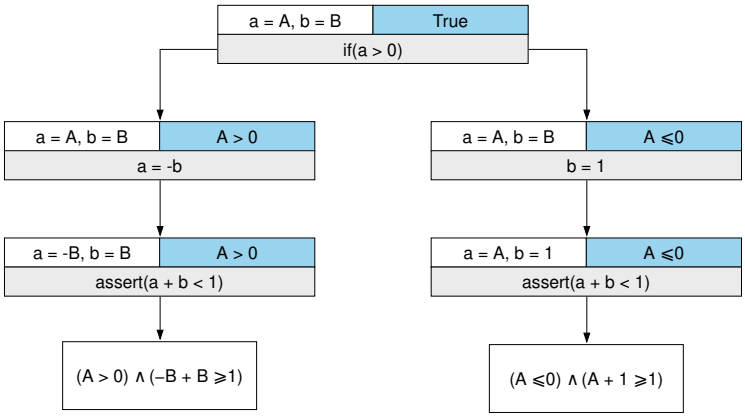

图 18.5: 符号执行流程 [5]

足的条件 [5]。图18.5中是符号执行的流程，灰色的方框表示当前将要执行的语句，白色的方框表示每个变量的值，蓝色的方框表示执行到当前语句要满足的条件，最下面的两个方框是生成的逻辑表达式。程序执行前变量 a 和 b 的值用符号 A 和 B 表示。图18.5的根节点对应程序的第一条语句`if(a > 0)`，a 的值是符号 A，b 的值是符号 B，因为这条语句是无条件执行的，所以执行到当前语句要满足的条件是 True。由于下面程序的执行有两种可能的路径，所以根节点由此产生两个子节点，左边对应`a > 0`成立的程序执行情况，右边对应`a > 0`不成立的程序执行情况。现在来看左边这条路径，下一步要执行的语句是`a = -b`，执行到当前语句的条件变成了 $A > 0$。接下来执行的是`assert(a + b < 1)`，现在 a 的值变成了-B，b 的值还是 B，执行到当前语句要满足的条件是 $A > 0$。接下来符号执行会生成一阶逻辑公式 $(A > 0) \wedge (-B + B \geqslant 1)$，如果这个公式在某些情况下成立，那么说明 assert 会报错，而这个公式任何情况下都不成立，说明 assert 任何情况下都不会报错。接下来看右边这棵子树，按照之前讲的符号执行流程，最终生成的逻辑表达式是 $(A \leqslant 0) \wedge (A + 1 \geqslant 1)$，这个公式在 A=0 时成立，说明如果输入的 a 是 0，那么 assert 会报错，这样就找到了程序的 bug。由于符号执行会遍历程序所有可能的执行情况，所以也可以用符号执行判断霍尔三元组是否成立。

Serval 是基于符号执行实现的一键证明工具，能够自动化证明串行程序的功能正确性和安全性，由华盛顿大学的研究人员用 Rosette 语言开发，获得了 SOSP 2019 的最佳论文奖 [25]。SMT 求解器本身能力有限，在证明大型系统时常常遇到证明超时的问题，如何将一键证明工具用于证明大型系统是

一个极富挑战且很有价值的问题。程序的具体实现可以使用各种高级语言，而后编译成汇编或者 LLVM IR 等中间代码，Serval 证明的是这些编译后的代码。这样一来，一方面编译器不在可信基里面，它如果编译出错误的代码也可以发现；另一方面，如果程序实现同时包括高级语言和汇编，那么这样就可以统一处理。为了解决 SMT 求解器的性能问题，Serval 基于之前的工作 SymPro，能够帮助开发者发现导致 SMT 求解器求解缓慢的瓶颈代码，修正瓶颈代码之后可以有效加快证明的速度。研究人员用 Serval 证明了 Keystone 里面的安全监视器和 Linux 内核中的 BPF 编译器，发现了 18 个新的 bug，都得到了开发者的确认和修复 [25]。这些都证明了 Serval 在证明大型真实系统方面的实用性。在 github 上有一个 Serval 的使用教程，附有详细的例子和讲解[3]。

18.5.3 半自动证明

本节首先介绍常用的半自动证明技术最弱前置条件，之后介绍半自动证明工具 Dafny，最后介绍使用 Dafny 证明密码学库的案例 Vale。

最弱前置条件

Dafny 等半自动证明工具以霍尔逻辑为理论基础，通过最弱前置条件实现半自动证明。下面主要讲解如何利用最弱前置条件实现半自动证明。

对于一个霍尔三元组 $\{P\}\ C\ \{Q\}$，W 是最弱前置条件是说对任意的 R，$\{R\}\ C\ \{Q\}$ 成立等价于 R 能推导出 W，即 $R \Rightarrow W$。这里的最弱前置条件可以用 $wp(C, Q)$ 来表示。例如 $wp(x++, x > 2)$ 就是 $x > 1$。

根据最弱前置条件的定义，要证明霍尔三元组 $\{P\}\ C\ \{Q\}$ 成立，只需要证明前置条件 P 能够推导出最弱前置条件 $wp(C, Q)$ 就可以了，即证明 $P \Rightarrow wp(C, Q)$ 成立。这个公式是否成立可以利用 SMT 求解器自动证明。例如，要证明 $\{x = 2\}\ x++\ \{x > 2\}$ 成立，只要证明 $x = 2 \Rightarrow x > 1$ 就可以了。

自动化的最后一个问题就是如何计算最弱前置条件。如果程序中不含有无界循环，那么有公式可以自动根据程序和后置条件算出最弱前置条件。如果程序中有无界循环，那么需要使用者定义循环不变式才能算出最弱前置条件。循环不变式是一个命题，循环执行前成立，每次循环体执行后都成立，循环结束后也成立。所以半自动证明是可以处理无界循环的。

[3]https://github.com/uw-unsat/serval-tutorial-sosp19

Dafny

Dafny 是微软开发的用于证明串行程序的半自动证明工具，自动化程度非常高，同时 Dafny 也是一门编程语言，它的底层原理是霍尔逻辑，利用 SMT 求解器自动证明定理。

```
method func(a : int, b : int) returns (c : int)
  requires a > 0
  ensures c < 1
{
  var tmp1 := a;
  var tmp2 := b;
  if a > 0
    { tmp1 := -b;}
  else
    { tmp2 := 1;}
  return tmp1 + tmp2;
}
```

代码片段 18.6: Dafny 程序示例

给定一段程序，用户通过编写前置条件和后置条件来定义形式化规约，之后 Dafny 会自动证明程序是否满足形式化规约。例如代码片段18.6中，给定一个函数`func`，参数是两个`int`类型的变量，返回值是一个`int`类型的变量。这个函数的功能和之前一键证明中的代码片段18.5相同。`requires`是 Dafny 中的关键字，用于定义函数的前置条件，这里的前置条件`requires a > 0`表示函数的参数`a`大于 0，`ensures`是 Dafny 中的关键字，用于定义函数的后置条件，这里的后置条件`ensures c < 1`表示函数的返回值`c`小于 1。之后 Dafny 会证明这个程序是否满足前置条件和后置条件定义的形式化规约，这段程序是满足的，所以 Dafny 最后会输出证明通过。

Dafny 是一个半自动的工具，不能全自动地证明任意程序。如果程序比较简单，证明过程可以实现全部自动化。但是对于很复杂的程序，程序员需要定义一些定理或者在程序中加一些标记辅助 Dafny 进行证明，如果程序中包含无界循环，程序员还要自己定义循环不变式，否则 Dafny 无法证明通过。

Vale

Vale 是微软研究院开发的用于自动化证明密码学代码的工具，基于半自动化证明工具 Dafny 开发，获得了 USENIX Security Symposium 2017 的杰出论文奖 [7]。密码学库在网络等方面应用非常广泛，但是其中隐藏着难以发

现的漏洞。Vale 不仅能够形式化证明密码学代码的实现是正确的，而且保证了其中不存在基于时间的侧信道漏洞和基于 cache 的侧信道漏洞。

密码学代码为了高性能，通常是用汇编实现的，但是 Dafny 不能解析汇编代码，所以研究人员在 Dafny 基础上开发了新的语言 Vale，语法和汇编类似。要证明密码学程序，需要按照 Vale 的语法实现一遍。等到证明完成了，Vale 会生成能在 ARM 等体系结构上运行的真实汇编代码。如果是直接用 Dafny 实现一遍密码学算法，性能会比汇编实现的差很多。测试发现，使用 Vale 开发证明的几种密码学程序的性能和 OpenSSL 大体相当 [7]。

Vale 和 Dafny 一样使用霍尔逻辑形式化证明密码学程序的功能正确性。使用者需要定义前置条件和后置条件，之后 Vale 会自动生成相应的定理交给 Dafny 进行证明。

Vale 形式化定义了安全性，能够消除时间侧信道和 cache 侧信道。机密信息是通过程序的外部行为泄露的，包括攻击者可见的程序状态、程序运行时间和 cache 访问模式。安全性定义为如果程序运行产生的外部行为和机密数据的值没有关系，那么这个程序就是安全的。

Vale 并没有直接使用这个安全性定义证明具体程序，而是实现了一个分析器，分析器的输入包括要证明的程序和攻击者可见内存位置，输出程序是否安全，之后形式化证明如果分析器判定程序是安全的，那么程序一定安全。这样一来就不需要对每个密码学程序都证明一遍了，只需要用分析器检查一遍就能保证安全性，实现一劳永逸。

后来微软研究院基于 Vale 开发证明了一个密码学库 EverCrypt，其中的部分代码在 2020 年被加入到 Linux 内核当中[4]。

18.5.4 交互式证明

在本小节，我们会介绍交互式定理证明工具 Coq、为什么选择 Coq 以及 Coq 的应用案例 CompCert。

Coq

Coq 是一个交互式的定理证明工具，由法国国家信息与自动化研究所研发，它提供了一种语言，可以用来定义算法、定义定理和编写形式化的证明等。Coq 常用于编写形式化规约并证明程序实现和形式化规约一致。

[4]https://www.cylab.cmu.edu/news/2020/04/29-parno-evercrypt.html

```
Lemma example:
  forall p q r,
    (p -> q) ->
    (q -> r) ->
    (p -> r).
Proof.
  intros.
  apply X0.
  apply X.
  assumption.
Qed.
```

代码片段 18.7: Coq 程序示例

代码片段18.7中展示了 Coq 是如何形式化定义定理并给出证明的。这个定理叫example，p->q表示命题p可以推导出命题q，这个定理说的是对于任意的 3 个命题p、q和r，如果已知p能推导出q，q能推导出r，那么可以得到p一定能够推导出r。Proof是 Coq 中的关键字，意思是证明过程的开始，之后是一系列形式化证明的代码，最后的Qed表示证明结束。

如图18.6所示，Coq 的图形界面分为左右两边，三个窗口。左边的窗口是编写代码的地方。右边分为上下两个窗口，上面的窗口显示当前的证明状态，包括已知条件和证明目标，两者之间用一个横线分开，下面的窗口显示提示信息，当代码出现错误，运行不下去时提示错误的原因，左边窗口中出错的那行代码用红色标出。Coq 是一个交互式的证明工具，左边绿色的部分是已经执行的代码，开发者每次编写一行代码后执行一下，右边的当前已知条件和证明目标就会发生变化，之后开发者根据新的条件和目标编写证明代码，循环往复，完成定理的证明。

Coq 是一个开源的形式化证明工具，拥有丰富的定理库，定义了包括整数、数据结构等相关的许多定理和操作，用户在进行相关证明的时候可以引用这些定理辅助证明，减轻证明负担。Coq 支持编写高阶逻辑表达式，在描述某些性质的时候比一阶逻辑更加简洁方便。另外，Coq 支持用户定义 Ltac 来使证明过程更加自动化一点，Ltac 根据当前的目标搜索合适的条件自动进行一些证明操作。因为这些优点，Coq 成为了主流的形式化证明工具之一。

CompCert

CompCert 是一个形式化证明过的 C 语言编译器，由法国 INRIA 的研究人员使用交互式证明工具 Coq 证明 [21]。CompCert 支持的是一种简化的 C

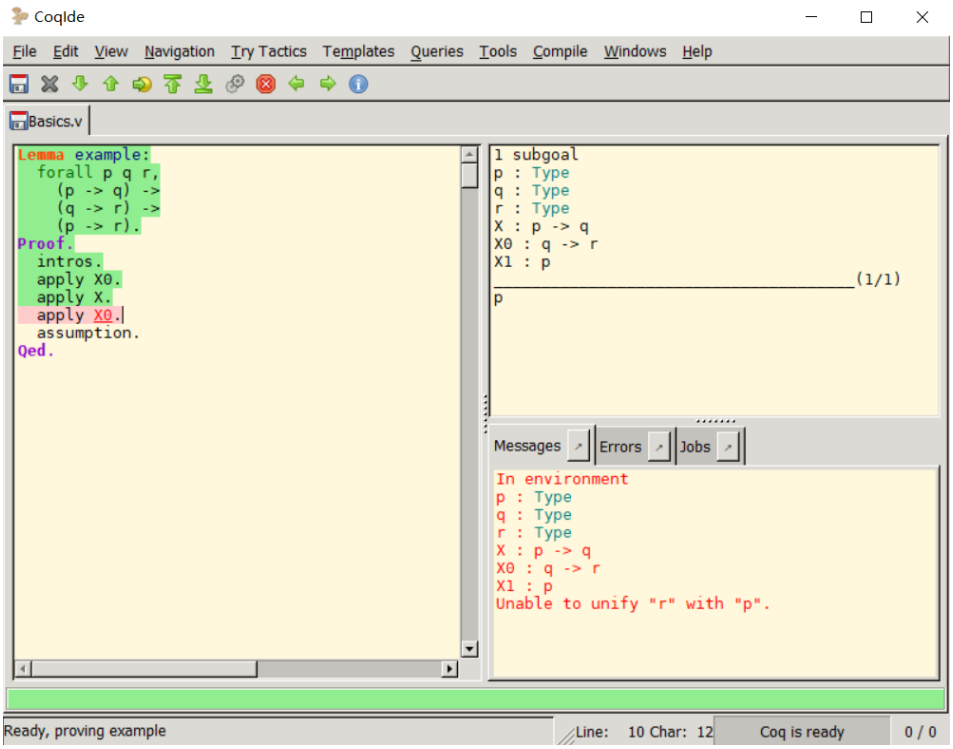

图 18.6: Coq 的图形界面

语言，叫做 Clight，遵照 ISO C 99 标准，保证编译产生的代码行为和源代码的行为一致。编译器是一个非常复杂和重要的系统软件，里面隐藏着很多的错误，如果编译器有错误，即使源代码的正确性被形式化证明了，编译产生的可执行文件也可能是错误的，所以编译器的形式化证明是非常有价值的。具体来说，CompCert 编译器保证了如果源代码 S 没有未定义行为和编译错误，那么 S 的行为和 CompCert 产生的代码 C 的行为一致。

18.5.5 证明小结

本节按照自动化程度将证明手段分为一键证明、半自动证明和交互式证明，讲解了它们使用的技术和各自的优缺点，并介绍了一些具体的证明工具。

证明一个系统的时候，需要根据系统的特点和证明需求来选择合适的证明手段，也可以把多种证明手段结合。例如，一键证明自动化程度高，但是不能处理无界循环，半自动证明能处理无界循环，但是自动化程度不如一键证明，那么可以把系统分为两部分，一部分不包含无界循环，这部分用一键证明工具

处理，另外一部分包含无界循环，这部分用半自动证明工具处理。

18.6 操作系统的形式化证明

本节主要知识点

- ❑ 为什么要形式化证明操作系统？
- ❑ 证明操作系统有哪些挑战？

18.6.1 操作系统形式化证明的必要性与挑战

作为最基础与最重要的系统软件，操作系统被广泛应用于航空、交通、能源、医疗、军事等安全攸关的基础设施以及人们的日常生活中的电子设备中，任何一个微小的操作系统内核错误都有可能导致整个计算机系统的崩溃，造成财产甚至生命的损失，操作系统的安全与可靠具有至关重要的意义。传统的方法难以提供操作系统需要的高保障性，与此同时，用形式化证明来保证操作系统的正确性成为了新的研究热点。

然而，证明操作系统具有以下一些显著的特点：

- 不同于应用软件依赖于下层软件提供的简洁的抽象，操作系统下层基于硬件实现。
- 大多数操作系统的实现基于 C 语言内联汇编，具有较大的代码规模。
- 除了功能正确的要求外，操作系统的丰富的系统级性质和系统不变式需要得到保证。

这些特点给操作系统的证明带来了巨大的挑战，我们将从实现、规约、正确性定义、证明以及软件工程的角度分别进行讨论。

形式化证明中，我们需要对系统实现进行精确的数学建模，而大多数操作系统的实现基于 C 语言和内联汇编，这会带来三方面挑战，首先，C 语言的语义通过自然语言文档描述，缺乏严格的定义，C 语言丰富的语言特性会给建模带来困难，其次，C 语言和汇编语言所操作的程序状态在不同的抽象层次上，汇编语言需要访问硬件状态（如寄存器），更加底层，C 语言的内存模型则更加高层，操作系统的实现混合使用两种语言，如何统一建模它们也就成为了一

个难点，第三，操作系统实际运行时还会受到硬件中断的影响，建模硬件中断的行为会使得系统建模更加复杂。

从规约和正确性定义的角度来看，我们需要定义出使用者看到的操作系统，使用者可以将操作系统理解为一系列系统调用的集合，他们通过调用这些 API 来与硬件交互。操作系统的规约通常不会将复杂的实现细节暴露给使用者，即以抽象的形式定义，定义出抽象状态和每一个接口的如何操作抽象状态。如何有效的对操作系统进行抽象是我们定义规约的时候需要考虑的问题。

除了通过抽象刻画出接口功能之外，我们还关心操作系统多方面的系统级性质，比如安全性（如隔离性（Isolation）、不干涉性（Noninterference）、保密性（Confidentiality）、完整性（Integrity）等）、活性（如无死锁性（Deadlock-freedom）、无锁性（Lock-freedom）等）、原子性、实时性、崩溃一致性等，有些系统级性质可以通过抽象体现，比如通过将抽象定义为原子的，可以体现出原子性，但更多的性质无法从抽象中直接体现出来，如崩溃一致性需要考虑系统在执行过程中任意时刻发生崩溃时的系统行为，要想证明这些性质就需要形式化刻画这些性质，并在证明目标（正确性条件）中体现。

证明操作系统最大难点在于并发的证明，操作系统中存在多种形式的并发，包括多核并发、多线程并发、I/O 并发以及中断导致的并发，减轻证明并发的困难的核心是**局部性**以及**可组合性**。直接验证并发环境中的多个线程需要考虑到这些线程所有可能的交错执行情况，使得证明无从下手，所谓局部性指的是我们希望仅仅关注在其中一个线程的验证上，但同时我们需要考虑到其他线程可能的干扰。当我们完成了对于每一个局部的证明，我们希望这个证明结果对于整体的并发系统也是成立的，即证明具有可组合性，这对于我们的证明理论提出了要求。

并发证明结合上系统级性质的证明往往不是简单的加和，许多在串行环境下成立的证明理论，在并发环境下都无法简单成立（如崩溃一致性和安全性），证明并发环境下的系统级性质需要证明理论上更多的创新和进步。

开发操作系统本身是一个系统性工程，而将形式化证明融入开发过程中更是增加了巨大的挑战。软件开发具有快速迭代的特点，而软件证明则耗费时间和人力，如何对快速迭代的软件版本验证正确性？即使投入巨大的人力完成了一个版本的证明，可能下一个版本又改变了原来的设计，又需要修复原有的证明，这样的软件开发过程不具备可持续性。目前在工业界能够得到应用的形式化方法都需要具备**证明自动化**的能力，即证明过程能够尽量自动完成，不需要投入巨大的人力，这样开发人员主要的精力可以主要用于规约的书写。然而能够自动化证明的方法在能够验证的软件功能和性质上具有明显的局限性（如符

号执行不支持无界循环以及并发），这极大制约了形式化证明的广泛应用。

这些挑战不仅仅体现在操作系统的证明上，也体现在其它软件的开发上，他们意味着形式化证明领域仍然充满着机遇。在下一节中，我们将介绍操作系统证明的现状。

18.6.2 操作系统证明现状

操作系统的证明工作最早可以追溯到上世纪 70 年代，早期的工作包括 UCLA [32]、PSOS [27]、KIT [6]、VFiasco [15, 14]、Verisoft [3] 等，他们都在各自方向上做出了贡献，使证明操作系统变得可行，本书不再对他们进行详细的介绍，感兴趣的读者可以参考已有的综述文献 [18] 对他们的对比与总结。seL4 [20] 是操作系统证明历史上的一个里程碑，其率先验证了一个可以实际应用的操作系统内核。在 seL4 之后的工作如 Verve [34]、Hyperkernel [26]、Certiucos [33]、CertiKOS [13] 等又分别从不同的方面进行了创新。总结来说，经历了近半个世纪的探索，在操作系统证明上，从系统规约、证明理论、编程语言到软件工程等方面，我们都取得了长足的进步。

要证明一个操作系统，首先需要明确证什么，如何书写系统规约，如何定义正确性，这也是最早的工作优先突破的挑战，UCLA 和 KIT 都选择了分层的精化关系作为正确性条件，尽管具体分层数不同，他们的分层里都包含了一个抽象层规约，并通过证明精化关系来证明操作系统的功能正确性，除此之外，这两个项目都希望证明操作系统安全相关的性质，于是又提出了一个顶层规约，顶层规约专注于刻画一个安全的模型，抽象掉不相关的功能。他们证明相邻两层间的精化关系，以此来说明最底层的实现符合顶层规约并具有安全性。分层精化证明能将一个困难的证明分成多步解决，从而极大降低每一步证明的复杂性，这个思想也被后续操作系统证明的工作所采纳。

UCLA 和 KIT 在实现的编程语言上都做了一定简化，没有选择 C 语言内联汇编，使其无法验证主流的操作系统，这一点在后续的工作中得到了解决，VFiasco 和 Verisoft 尝试对 C++ 或类 C 的编程语言进行证明，一个主要的难点是内核运行的内存模型比 C 语言的内存模型更底层，通常 C 语言会假设内存访问不会出错，这种不会出错的内存抽象正是内核实现的。面对这个难点，VFiasco 和 Verisoft 建模了更实际的内存模型来考虑到底层硬件的行为，并基于此建模了汇编语言和 C 语言，使其能够支持操作系统的证明。

在之前的验证工作中，要么没能完成大部分证明，要么证明的系统性能不够好，在使用形式化证明的同时如何保证系统性能与开发效率是一个巨大的挑战，seL4 提出了一套快速内核设计开发方法来解决这个难题，首次让证明的

内核能被实际应用，我们将在第18.7节中对 seL4 进行案例分析。

尽管 seL4 展现了形式化证明可以被应用到实际系统中，但 seL4 的证明耗费了巨大的人力，于是更多的工作开始关注证明的效率，使用半自动证明或一键证明来减轻证明的负担。

Verve 是一个使用半自动证明技术证明的类型安全的操作系统，Verve 由两部分组成，Nucleus 部分和基于 Nucleus 实现的内核部分，Verve 对于内核功能的分割与 Verisoft 项目想法类似，其中 Nucleus 负责实现访问硬件与内存的原语，内核部分实现其他的服务功能。Verve 希望从汇编层面保证内核的类型安全，因此 Verve 的实现完全由汇编构成，并且直接在汇编层面进行自动化的验证。其中 Nucleus 直接由汇编实现，并通过 Boogie 工具进行半自动证明，我们只需要在汇编代码中手动注释上前后置以及循环的不变式，Boogie 可以通过 SMT 求解器自动验证汇编代码符合手动注释的规约。内核部分用一个类型安全的 C# 语言实现，通过编译器编译成类型化汇编语言（Typed Assembly Language），通过一个已有的 TAL 检查器自动验证。

Hyperkernel 则是一个使用符号执行完全一键证明的内核，其接口基于 xv6 系统，不过由于符号执行难以处理无界循环，因此 Hyperkernel 的设计需要避免无界循环，其解决办法是借鉴外内核（Exokernel）的想法，将需要使用无界循环的地方移出内核，由用户态程序自行实现。为了避免验证复杂的 C 的语义，Hyperkernel 选择将 C 语言写的内核通过 LLVM 编译器编译，并对 LLVM 中间表示进行验证，Hyperkernel 的规约以抽象的形式给出，以实现和规约的精化关系作为正确性条件。Hyperkernel 使用了符号执行来穷尽抽象和具体所有可能的状态转移，并将符号执行得到的结果交给 SMT 求解器进行验证。Hyperkernel 的后续工作 Nickel 同样采用自动化证明技术验证了操作系统的安全性。

虽然半自动化或一键证明能有效的降低证明的负担，但是 Verve 和 Hyperkernel 也都有明显的局限。它们都受制于证明技术的局限，无法验证完整的复杂的内核功能，其次现有的自动化证明技术基于 SMT 求解器的能力，这使其难以验证并发程序，因为并发程序需要考虑所有可能的并发执行的情况，使 SMT 求解时难以在有限时间内返回。

Certiucos 和 CertiKOS 则是验证操作系统中并发问题的两个代表项目。其中 Certiucos 项目验证了一个商用抢占式操作系统 μC/OS-II 的核心模块，μC/OS-II 内核中所有共享资源的访问都发生在临界区中，不存在细粒度的无锁并发，因此项目验证重点为中断导致的并发。在并发验证中，核心难点是并发验证理论的可组合性与局部性，Certiucos 验证框架建模了简化的 x86 的中

断模型，并扩展已有的并发精化验证理论来支持中断导致的并发的可组合性验证。项目在 Coq 定理证明器中建模了 C 语言的子集和中断相关的汇编原语，实现了整个验证框架并证明了理论的可靠性，Certiucos 的规约同样以抽象的形式定义，并选取了上下文精化作为正确性条件。

在介绍 CertiKOS 之前，我们先简单介绍 PSOS 工作。PSOS 工作严格使用了抽象、模块化、封装、分层等软件设计思想，着眼于硬件、软件与应用层的规约设计，将从硬件一直到应用层的软件划分为 17 个层次，上层仅依赖下层的规约，有效的抽象掉下层的细节。PSOS 仅设计规约，没有将其与实现通过证明连接起来，CertiKOS 则严格形式化了这一想法。

CertiKOS 项目率先验证了首个细粒度并发的通用操作系统内核，其由约 6500 行 C 和汇编组成，并能运行在 x86 机器上。CertiKOS 将操作系统内核的功能划分为了 37 个层次，这些层次被严格的形式化与证明，每一层的实现都被证明与规约具有上下文精化关系，得益于层次的划分，验证并发操作系统的复杂工作量被分摊到了每个层次，并且当一层的规约是原子的，在上层的证明中可以直接使用下层原子的规约，从而不再需要对下层可能的并发执行情况进行讨论。除此之外，CertiKOS 基于 CompCert 开发了 CompCertX，从而能够保证形式化证明的保障能够传递到汇编层面。严格分层的证明理论有效减轻了证明负担，除了证明了内核的功能正确性，CertiKOS 及其后续项目还验证了内核的其他性质，如活性 [17]、安全性 [10] 和实时性 [23]。但同时严格分层的方法也牺牲了软件开发的便利和性能，由于用这种方法开发软件十分困难，现实中很少软件架构是严格分层的架构。

18.7 案例分析：seL4 的形式化证明

本节主要知识点

- ❑ 为什么要形式化证明微内核？
- ❑ *seL4* 使用什么技术进行形式化证明？
- ❑ *seL4* 的证明负担有多大？

seL4 是第一个完全形式化证明过的通用操作系统内核，由澳大利亚新南威尔士大学等机构的研究者使用交互式证明工具 Isabelle/HOL 开发验证 [20]。seL4 采用微内核架构，IPC 性能优于没有完全形式化证明过的微内核 Zircon 和 Fiasco.OC，是一个真正实用的操作系统内核 [24]。下面会从设计开发流程、

证明方法等方面介绍 seL4。

18.7.1 内核设计开发流程

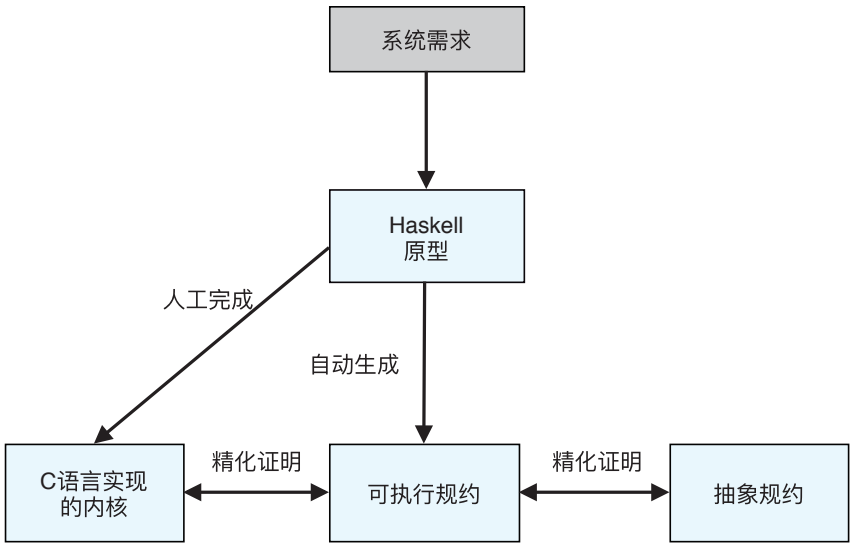

图 18.7: seL4 的开发流程 [19]

本节描述 seL4 的设计开发流程。如果先用 C 语言和汇编实现操作系统，然后再编写形式化规约进行证明，那么可以在实现阶段就能发现系统的性能问题并迅速改正，但是在证明阶段如果发现系统设计有问题，需要更改设计并修改实现，迭代速度太慢。如果先定义规约，然后分析系统设计是否满足规约，设计上有问题可以较早改正，但是如果在实现操作系统的时候，发现这种设计会导致性能特别差，就需要更改设计并重新证明，迭代速度还是比较慢。

所以，seL4 采用了图18.7中的开发流程。首先按照功能性等系统需求设计操作系统，并用函数式编程语言 Haskell 实现一个内核原型。这个原型有两个用处，一方面它是可以实际运行的，研究者实现了一个模拟器可以在原型上面运行应用程序，这样可以尽早发现性能问题。另一方面，Haskell 原型可以自动转成 Isabelle/HOL 代码，进行正确性的验证，如果发现正确性有问题，可以及时修改原型，迭代开发。这种设计开发流程可以克服前面两种流程的缺

点，加快迭代速度 [19]。尽管 Haskell 原型也可以运行，但是最终的 seL4 是用 C 语言重新实现一遍的。

seL4 验证完成后要编译成二进制文件运行，如何证明二进制文件也是正确的呢？最初 seL4 是把编译器、硬件等作为可信基，后来为了避免编译器中的 bug 影响正确性，seL4 尝试使用形式化验证过的 C 语言编译器 CompCert 来编译。

18.7.2 seL4 内核验证

为什么验证微内核？

我们平时常见的操作系统包括 Windows、Linux 等等都是宏内核，seL4 采用微内核的设计是因为微内核比宏内核更适合进行形式化验证。

宏内核包含的功能比较丰富，设备驱动、文件系统等都包括在宏内核里面，所以宏内核的代码量往往十分巨大。而微内核把文件系统、驱动程序这些都放到用户态，只保留 IPC、调度器等最基本的功能，一般代码量在 10000 行左右，比如 seL4 包含大约 8700 行 C 代码和 600 行汇编 [20]。对于交互式证明来说，需要人工编写大量的代码，系统的代码量越大，需要人工编写的代码量就越多。对于自动化验证来说，由于 SMT 求解器的能力有限，如果系统太大太复杂，会导致 SMT 求解器的求解时间过长。所以，目前形式化验证的技术适合用于小体量的系统，微内核的体量小，适合进行形式化验证。

证明的性质

研究者形式化证明了 seL4 的功能正确性。这里的功能正确性是指 seL4 的代码实现满足规约。另外，形式化验证保证了 seL4 的运行过程中不会崩溃，不会发生空指针访问等错误 [20]。

证明方法

seL4 的主要证明方法是精化关系。前面提到，研究者先用 Haskell 实现一个原型，然后把原型自动转成 Isabelle/HOL 代码，这个称为可执行规约，另外还定义了抽象规约，描述系统的功能。C 语言和汇编实现的 seL4 包含很多细节，可执行规约比它抽象一些，而抽象规约包含的细节最少，只描述系统功能，不描述具体实现方式。证明过程中需要证明 C 语言版本的内核和可执行规

约之间的精化关系，还要证明可执行规约和抽象规约之间的精化关系。因为精化关系具有传递性，所以也证明了具体实现和抽象规约之间的精化关系 [20]。

18.7.3 测试与经验

证明对系统的影响

由于要对 seL4 进行形式化证明，所以在设计和实现系统的过程中要考虑证明的一些需求。

一方面，形式化验证对设计实现 seL4 带来了一些限制。为了便于形式化证明，有些 C 语言的特性是不支持的，比如不能使用 union 数据类型，不能对局部变量取地址等，不过即使有这些语法上的限制，也基本不影响写程序。验证并发是非常困难的，所以 seL4 使用的是一把大锁，不用细粒度锁。seL4 很多时候会关中断，这样可以降低并发带来的复杂度。为了方便验证，函数要尽量没有副作用，全局变量要少用 [20]。

另一方面，形式化验证要求系统的设计要尽可能简洁，这样更方便证明，不过简洁对 seL4 的整体设计却是有利的，而且能帮助开发者对系统有更深刻的理解，减少错误 [20]。

发现的错误

形式化验证和软件测试不同，可以完全保证代码实现符合规约。下面讨论 seL4 中找到的错误。

验证 seL4 包括两步精化证明。第一步是可执行规约和抽象规约之间的精化证明，这一步对代码进行了 500 次修改，大约一半是因为算法和设计有错误，剩余的修改是为了方便验证，第二步是 C 代码和可执行规约之间的精化证明，进行这一步证明时 seL4 已经被一些项目使用，发现了 16 个错误，但是通过验证又找到了 144 个错误 [20]。这些都说明形式化验证对于保证程序正确性是非常有效的。

后续工作

seL4 最初只证明了功能正确性和部分的安全性，但是有一些很严重的安全问题是基于硬件的侧信道攻击导致的，恶意程序可以基于程序使用的带宽等信息来进行攻击，窃取机密数据，比如 cache 侧信道等。后来研究者在 seL4

里面加入了一些措施来防范这类攻击，比如 cache 染色、内核克隆等，并尝试证明 seL4 能够防范这些侧信道攻击 [12]。

证明成本

seL4 的证明是使用交互式工具 Isabelle/HOL 实现的，需要人工编写大量代码。具体来说，编写抽象规约花费 4 人月，开发 Haskell 原型花费 2 人年，编写转化器把 Haskell 原型转成可执行规约花费 3 人月，编写 C 版本的 seL4 花费 2 人月，证明 seL4 花费了 20 人年，整个证明包括大概 20 万行 Isabelle 代码 [20]。

参考文献

[1] Coq reference manual. https://coq.inria.fr/distrib/current/refman/.

[2] Dijkstra (1970) "notes on structured programming" (ewd249), section 3 ("on the reliability of mechanisms"), corollary at the end. http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF.

[3] Eyad Alkassar, Wolfgang J Paul, Artem Starostin, and Alexandra Tsyban. Pervasive verification of an os microkernel. In *International Conference on Verified Software: Theories, Tools, and Experiments*, pages 71–85. Springer, 2010.

[4] Andrew W Appel, Lennart Beringer, Adam Chlipala, Benjamin C Pierce, Zhong Shao, Stephanie Weirich, and Steve Zdancewic. Position paper: the science of deep specification. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 375(2104):20160331, 2017.

[5] Roberto Baldoni, Emilio Coppa, Daniele Cono D' elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), May 2018.

[6] William R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, 1989.

[7] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. Vale: Verifying high-performance cryptographic assembly code. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 917–934, Vancouver, BC, August 2017. USENIX Association.

[8] Stephen Brookes. A semantics for concurrent separation logic. *Theoretical Computer Science*, 375(1-3):227–270, 2007.

[9] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Argosy: Verifying layered storage systems with recovery refinement. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 1054–1068, New York, NY, USA, 2019. Association for Computing Machinery.

[10] David Costanzo, Zhong Shao, and Ronghui Gu. End-to-end verification of information-flow security for c and assembly programs. *SIGPLAN Not.*, 51(6):648–664, June 2016.

[11] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[12] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. Time protection: the missing OS abstraction. In *EuroSys Conference*, Dresden, Germany, March 2019. ACM.

[13] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Newman Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. Certikos: An extensible architecture for building certified concurrent OS kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 653–669, 2016.

[14] Michael Hohmuth and Hendrik Tews. The vfiasco approach for a verified operating system. *2nd PLOS*, 2005.

[15] Michael Hohmuth, Hendrik Tews, and Shane G Stephens. Applying source-code verification to a microkernel: the vfiasco project. In *Pro-*

ceedings of the 10th workshop on ACM SIGOPS European workshop, pages 165–169, 2002.

[16] Daniel Jackson. *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

[17] Jieung Kim, Vilhelm Sjöberg, Ronghui Gu, and Zhong Shao. Safety and liveness of mcs lock—layer by layer. In *Asian Symposium on Programming Languages and Systems*, pages 273–297. Springer, 2017.

[18] Gerwin Klein. Operating system verification—an overview. *Sadhana*, 34(1):27–69, 2009.

[19] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an os microkernel. *ACM Transactions on Computer Systems (TOCS)*, 32(1):1–70, 2014.

[20] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220, 2009.

[21] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[22] Hongjin Liang, Jan Hoffmann, Xinyu Feng, and Zhong Shao. Characterizing progress properties of concurrent objects via contextual refinements. In *International Conference on Concurrency Theory*, pages 227–241. Springer, 2013.

[23] Mengqi Liu, Lionel Rieg, Zhong Shao, Ronghui Gu, David Costanzo, Jung-Eun Kim, and Man-Ki Yoon. Virtual timeline: a formal abstraction for verifying preemptive schedulers with temporal isolation. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–31, 2019.

[24] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. Skybridge: Fast and secure inter-process communication for microkernels.

In George Candea, Robbert van Renesse, and Christof Fetzer, editors, *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, pages 9:1–9:15. ACM, 2019.

[25] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling symbolic evaluation for automated verification of systems code with serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 225–242, 2019.

[26] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. Hyperkernel: Push-button verification of an os kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 252–269, 2017.

[27] Peter G Neumann and Richard J Feiertag. Psos revisited. In *19th Annual Computer Security Applications Conference, 2003. Proceedings.*, pages 208–216. IEEE, 2003.

[28] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, and Brent Yorgey. *Programming Language Foundations*. Software Foundations series, volume 2. Electronic textbook, May 2018.

[29] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Logical Foundations*. Software Foundations series, volume 1. Electronic textbook, May 2018.

[30] John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE, 2002.

[31] JM Spivey. The z notation: A reference manual. *PrenticeHall International,*, 1992.

[32] Bruce J Walker, Richard A Kemmerer, and Gerald J Popek. Specification and verification of the ucla unix security kernel. *Communications of the ACM*, 23(2):118–131, 1980.

[33] Fengwei Xu, Ming Fu, Xinyu Feng, Xiaoran Zhang, Hui Zhang, and Zhaohui Li. A practical verification framework for preemptive os kernels. In *International Conference on Computer Aided Verification*, pages 59–79. Springer, 2016.

[34] Jean Yang and Chris Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 99–110, 2010.

形式化证明：扫码反馈