# 基于混合抽象的硬件验证算法研究

(申请清华大学工程硕士专业学位论文)


培 养 单 位：　软件学院

工 程 领 域：　软件工程

申　请　人：　杨　　柳

指 导 教 师：　贺　　飞　副教授


二〇二一年五月

# Research on Hardware Verification Algorithm Based on Hybrid Abstraction

Thesis Submitted to

**Tsinghua University**

in partial fulfillment of the requirement

for the professional degree of

**Master of Engineering**

by

**Liu Yang**

**(Software Engineering)**

Thesis Supervisor: Associate Professor He Fei

**May, 2021**

# 学位论文公开评阅人和答辩委员会名单

## 公开评阅人名单

| | | |
|---|---|---|
| 黄民德 | 研究员 | 中国软件评测中心 |
| 周旻 | 副研究员 | 清华大学 |

## 答辩委员会名单

| | | | |
|---|---|---|---|
| 主席 | 黄民德 | 研究员 | 中国软件评测中心 |
| 委员 | 罗平 | 教授 | 清华大学 |
| | 张慧 | 副研究员 | 清华大学 |
| | 张荷花 | 副研究员 | 清华大学 |
| | 姜宇 | 副教授 | 清华大学 |
| 秘书 | 万海 | 副研究员 | 清华大学 |

# 关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：

清华大学拥有在著作权法规定范围内学位论文的使用权，其中包括：（1）已获学位的研究生必须按学校规定提交学位论文，学校可以采用影印、缩印或其他复制手段保存研究生上交的学位论文；（2）为教学和科研目的，学校可以将公开的学位论文作为资料在图书馆、资料室等场所供校内师生阅读，或在校园网上供校内师生浏览部分内容；（3）按照上级教育主管部门督导、抽查等要求，报送相应的学位论文。

本人保证遵守上述规定。

作者签名：_____　　导师签名：_____

日　　期：_____　　日　　期：_____

# 摘 要

各类高度集成化、智能化的硬件设备应用在社会的各个领域中，为我们的生活带来了极大便利。随着硬件设计复杂性和规模的大幅度提升，硬件验证工作更加具有挑战性。模型检测技术是一种自动化检测技术，具有自动化和可以输出反例路径信息的特性，因此成为了当前硬件验证领域内最为重要的研究方向之一。

随着硬件技术的发展，硬件设计理念逐渐从底层的比特级向更高的抽象级别转变，为了将高级抽象语义信息运用到模型检测算法中，提高硬件验证的效果，模型检测算法也逐渐从比特级转为更高的字级。IC3 算法作为最成功的比特级模型检测算法之一，如何将其扩展到字级，近年来受到关注。本文的研究目标也是设计一个新型有效的字级 IC3 算法。

针对研究目标，本篇论文展开研究，主要完成了以下工作：

1. 设计并实现了一种将变量隐藏抽象和隐式抽象结合的字级 IC3 算法——IC3VA，从已有的字级 IC3 算法出现的问题出发，尝试将变量隐藏抽象和 IC3 算法相结合，确定了泛化和精化方案。
2. 设计了基于隐式混合抽象的 IC3 算法。提出了隐式混合抽象的概念，将多种抽象通过隐式抽象进行混合，给出了混合抽象下新的泛化和精化方法，并对算法进行解耦，从而得到一种可扩展的基于隐式混合抽象的 IC3 算法。
3. 给出了基于隐式混合抽象的 IC3 验证工具框架设计，并完成了工程实现，证明了混合抽象方案的可行性。在相关验证测试集上的实验结果表明，相比于被混合的各个抽象域的配置，混合抽象的配置解决的任务个数最多，算法实现了综合各个抽象优势的设计初衷。

**关键词**：硬件验证；字级模型检测；隐式抽象；混合抽象

# Abstract

Nowadays, various highly integrated and intelligent hardware devices are used in various fields of society, bringing great convenience to our lives. At the same time, with the substantial increase in the complexity and scale of hardware design, hardware verification becomes more necessary and challenging. Model checking is a kind of automatic verification technology, which has the characteristics of reliability, automation and having paths of counterexample, so it has become one of the most important research directions in the field of hardware verification.

With the development of hardware technology, the design concept of hardware has gradually changed from the bit-level to a higher level. In order to better apply high-level semantic information to model checking algorithms and improve the effect of hardware verification, the design concept of hardware model checking algorithms also has gradually shifted from bit-level to word-level. The IC3 algorithm is one of the most successful bit-level model verification algorithms, so how to extend it to the word-level has gradually attracted attention in recent years. Therefore, the research goal of this paper is to design a new and effective word-level IC3 algorithm.

Aiming at the research goal, I started the research and mainly completed the following tasks:

1. We proposed the IC3VA algorithm, a word-level IC3 algorithm based on the variable hiding abstraction. Starting from the problems of the existing word-level IC3 algorithm, I propose to combine the variable hiding abstraction with the IC3 algorithm, and design adaptable modifications to the generalization and refinement, and finally get a new word-level IC3 algorithm.
2. The word-level IC3 algorithm based on implicit hybrid abstraction is proposed and implemented. The paper proposes to use implicit hybrid abstraction to mix some abstract-based word-level IC3 algorithms, design a new method of generalization and refinement under hybrid abstraction, and finally get a scalable word-level IC3 algorithm.
3. At the same time, the paper gives the IC3 verification tool framework design based on implicit hybrid abstraction, and completes the engineering implementation, which proves the feasibility of the proposed hybrid scheme. The experimental

results show that compared with each configuration of the single abstraction, the configuration of hybrid abstraction solves the largest number of verification tasks, which indicates that the original design intention of integrating the advantages of each abstraction has been achieved.

**Keywords:** hardware verification; word-level model checking; implicit abstraction; hybrid abstraction

# 目 录

# 插图和附表清单

# 符号和缩略语说明

| | |
|---|---|
| HDL | 硬件描述语言（Hardware Description Language） |
| VHDL | 超高速集成电路硬件描述语言（Very-High-Speed Integrated Circuit Hardware Description Language） |
| RTL | 寄存器传输级别（Register Transfer Level） |
| BMC | 限界模型检测（Bounded Model Checking） |
| PDR | 属性引导的可达性（Property Directed Reachability） |
| ALU | 算术逻辑单元（Arithmetic and Logic Unit） |
| CEGAR | 反例制导的抽象精化（Counterexample-guided Abstraction Refinement） |
| SMT | 模块可满足性理论（Satisfiability Modulo Theories） |
| IC3PA | 基于隐式谓词抽象的 IC3 算法 |
| IC3VA | 基于隐式变量隐藏抽象的 IC3 算法 |

# 第 1 章　引言

## 1.1　研究背景与意义

### 1.1.1　硬件设计与硬件验证

自 20 世纪中期第一台计算机诞生以来，短短七八十年间，计算机与信息化技术得到了爆炸式的发展，其催生出的强大革命浪潮席卷全世界，让人类社会形态快速从工业社会迈入信息社会。时至今日，信息时代也已经逐步进入到智能化阶段，各种各样的智能化电子设备充斥在我们身边，为我们的生活和生产带来了极大便利。这种跨越式发展固然与应用端的计算机软件技术的蓬勃发展十分相关，但其实现基础仍然是各类电子元器件为代表的硬件技术取得了长足的进步。硬件技术的发展深刻影响着信息化社会的走向，也决定着未来社会智能化水平的上限。

从电子管、晶体管时代到中小规模集成电路时代，再到如今的大规模、超大规模集成电路时代，工业制造能力的提升使得电路的规模和复杂度大幅度增长，因而与之配套的硬件设计技术也发生了翻天覆地的变化。因为电路规模较小、功能简单，早期的硬件设计只需要以各类电子元件为主，充分考虑它们的特性按照需要的功能设计电路即可，而随着电路规模和功能复杂度的增长，这种设计方式变得不再实用，设计人员此时需要在更高的抽象级别看待硬件设计。为了应对这种实际需求，硬件描述语言和工具被设计开发出来。硬件描述语言定义了不同抽象级别的硬件设计表示，并允许用户使用高级别的描述定义硬件设计，且可以自动将其转换为低级别的等效电路表示。因此设计人员只需要考虑更高层的逻辑和时序设计即可。不用过多考虑底层的电路设计和实现，这无疑大大增加了硬件设计的效率。在集成电路功能需求日益多样化，规模和复杂度不断增长的现代，通过硬件描述语言来进行硬件设计已经成为最主流的一种设计方式。

电子硬件设备的一大特性是较高的精确性要求，而在一些安全性、可靠性攸关领域，例如航空航天、铁路运输、高精度仪器等，其对使用的电子设备的安全性和准确性的要求更是达到了一个极其高、甚至是绝对可靠的程度。因此，硬件设计方式固然重要，更重要的是，需要确保按照硬件设计生产的硬件设备是完全可靠的，否则一旦投入使用，就可能会造成不可预估、甚至是无法挽回的损失。所以自硬件设计出现以来，硬件验证就是一个同样受到高度关注的研究问题。通俗地说，硬件验证就是检查硬件设计对应的电路或系统是否按照指定的要求运行，这里的要求可能是功能性要求，也可能是安全性要求。

对于早期简单的硬件设计而言，由于功能较为简单，可以通过自行逻辑推导的方式判断设计是否符合要求，然后可以在生产出的样品上进行实际测试，完成硬件的验证。然而随着电路规模和复杂度的急剧增长，这种方式在可行性和成本上都显得不再实用，为此，硬件描述语言的集成开发工具中一般都内置了仿真模块，当设计人员完成高级别的硬件设计描述后，工具会自动将其转换为各个抽象级别上的等效电路表示，然后使用仿真模块对各个级别上的硬件描述进行数字模拟仿真，之后就可以尝试使用各种测试样例对这些模拟电路进行测试，判断其运行是否符合指定要求。这种仿真测试的验证方式大大降低了硬件验证的成本，当测试样例足够丰富时也基本可以保证验证的可靠性，因而成为了工业界中常见的硬件验证方式。

### 1.1.2　硬件模型检测的必要性和发展

虽然仿真测试的验证方式在一些情况下确实可以完成硬件验证，但也存在着明显的缺陷，即仿真测试只能发现硬件设计中可能出现的问题，证明硬件设计不满足要求，却无法证明硬件设计中没有问题。特别是当硬件设计规模较大时，各个模块之间时序和逻辑关系交错，电路的状态空间就会变得规模庞大且十分复杂，仿真测试的范围可能只覆盖其中的一小部分，可靠性就大打折扣。

模型检测技术是由 Edmund M.Clarke，Allen Emerson 和 Joseph Sifakis 等人提出的一种自动验证技术，其核心思想是将系统视为有限状态的转移系统 $M$，将待验证的属性视为逻辑公式 $P$，通过判定 $M$ 是否为 $P$ 的一个模型来进行验证，可以证明对于有限状态空间的系统，这个问题是可判定的，因此可以通过计算机在有限时间内自动完成验证。除此之外，如果验证发现 $M$ 不满足 $P$，算法会给出一条反例路径揭示 $M$ 如何违反属性 $P$。因此，由于具有完备性、自动化和反例路径信息等特性，模型检测技术天然地就与硬件验证产生了强相关性，在硬件验证领域被广泛研究使用。特别是在硬件设计日益复杂，硬件验证要求不断严格的大背景下，使用模型检测技术进行硬件验证就显得十分必要，模型检测技术也因此被广泛应用到嵌入式设备、安全通信协议、电子电路等硬件相关的系统的验证与分析之中，并取得了显著的效果和成功。硬件模型检测因此成为了硬件验证领域中最为热门的研究方向之一，Clarke 等三人也因“提出模型检测技术并将其发展为软硬件领域广泛采用的验证技术”而获得了 2007 年的图灵奖。

在硬件设计的早期，由于硬件功能相对简单，不管是硬件设计人员，还是硬件设计描述语言本身，基本都是从比特级（bit-level）看待和描述电路中的输入输出信息的，所以与之相对应的各类硬件模型检测算法，在早期也都是以比特级的理念设计相关流程进行模型检测。理论上比特级始终是最底层的电路设计表示，因

此时至今日，比特级的硬件模型检测算法依然有人在研究和提出。经过多年的不断发展进步，IC3 算法（Incremental Construction of Inductive Clauses for Indubitable Correctness）[1]是其中最具代表性的一种模型检测算法，它基于归纳推理的思想，通过构造一个归纳不变式来验证属性，该算法在各类测试数据集和模型检测赛事上都取得了不错的成绩，受到了研究者们的认可和关注。

随着硬件技术的不断发展，硬件设计的复杂度和规模急剧增长，硬件设计也因此出现了更高的抽象级别，硬件描述语言也逐渐支持开发者在更高的抽象级别上进行硬件设计。在这种发展趋势下，越来越多的硬件设计通过高级别硬件描述语言被定义出来，虽然相关开发工具可以自动将其转换为电路功能等效的低级别硬件表示，但这个转换过程本质上是有抽象语义信息丢失的，因为在低级别的硬件描述下，难以定义和获取到更高级别的抽象语义。而这种语义信息的丢失对硬件验证来说，很可能就是验证效率和精度的损失。面对这种情况，近年来，字级（word-level）硬件模型检测算法的概念被提出，研究者们希望可以将硬件设计中高于比特级的字级抽象语义信息提取出来，并将其应用到模型检测过程中，从而提升验证的效果。

IC3 算法作为最为成功的比特级硬件模型检测算法之一，如何将其拓展为字级的模型检测算法，自然成为了研究者们目前关注的焦点之一。IC3PA 算法[2-3]正是其中一种十分具有代表性的字级 IC3 算法，它运用了隐式抽象等技术，成功将字级抽象语义信息应用到 IC3 算法中，逻辑完备自洽，也取得了一定的效果。

综上，模型检测所具有的特性使得其成为目前硬件验证领域中最为重要的研究方向之一。近年来，为了更好地适配现代硬件设计的特性和发展趋势，提升硬件验证的效果，硬件模型检测算法的设计理念也逐步从比特级转为字级，鉴于各类传统比特级硬件模型检测算法之前取得的良好验证效果，将他们拓展到字级层面正成为一种字级模型检测算法研究的发展方向。

## 1.2　研究目标与主要思路

一个好的硬件模型检测算法对于硬件验证来说意义重大，相比于传统的比特级模型检测方法，字级模型检测算法运用了高级别的抽象语义信息，一方面更深刻地理解了硬件设计信息，另一方面也更加符合硬件设计的发展趋势。IC3 作为最为成功的比特级模型检测算法之一，将字级理念与之结合自然成为了一个值得考虑的研究方向。因此本文的研究目标为：设计并实现一种新型有效的字级 IC3 算法。主要思路是通过隐式抽象混合多种抽象，综合各种抽象的特点来增加可解任务的数量。

## 1.3　本文主要工作与贡献

在本篇论文中，通过对 IC3PA 这个已有的字级模型检测算法的分析，发现了其在一些特定测试样例会出现状态空间爆炸的情况，并分析了成因。受此启发，论文提出一种新的字级硬件模型检测算法——IC3VA。为了进一步综合各种字级 IC3 算法的优势，论文设计了一种新的基于混合抽象的 IC3 算法，实验也证明该方法有效综合了抽象域的优势，取得了更好的效果。

本文的主要贡献如下：

1. 设计并实现了基于隐式变量隐藏抽象的 IC3 算法——IC3VA。通过对已有的字级 IC3 算法 IC3PA 中存在的问题进行分析思考，论文尝试将变量隐藏抽象和 IC3 算法结合，给出了适配的泛化和精化方案，得到了一种新的字级 IC3 算法，并进行了实验验证。
2. 设计并实现了基于隐式混合抽象的 IC3 算法。论文设计了隐式混合抽象方法，将多种抽象进行混合，并给出了混合抽象下新的泛化和精化方案，最终设计得到一种可扩展的基于隐式混合抽象的 IC3 算法。
3. 给出了基于隐式混合抽象的验证工具的框架设计，并完成了工程实现，证明了方案的可行性。在相关验证测试集上的实验结果表明，相比于被混合的两个抽象域，基于隐式混合抽象的 IC3 算法解决的验证测试样例个数最多，算法实现了综合各个抽象方式优势的设计初衷。该框架是完全解耦的，未来也可以加入其它抽象域，提升验证效果，进一步发挥出框架的价值。

## 1.4　本文结构安排

本篇论文的整体组织结构安排如下。

第一章对本篇论文的研究工作做了整体的介绍。首先介绍了硬件验证对于硬件设计的重要性，模型检测技术在硬件验证领域中的重要地位以及硬件模型检测研究目前的发展状况。然后对本篇论文的研究目标、研究内容和主要贡献进行了简要的叙述。

第二章介绍了本篇论文的相关工作。以字级硬件模型检测为导向，按照逻辑的先后顺序，第二章中较为详细地阐述了硬件设计、硬件验证、模型检测、比特级模型检测、抽象与精化、字级模型检测等相关工作，为整篇论文的研究目标给出了较为完整的背景信息，也为后续的研究工作作出了理论铺垫。

第三章详细介绍了基于隐式变量隐藏抽象的 IC3 算法——IC3VA。第三章重点介绍了 IC3VA 算法的设计过程、关键部分以及整体流程说明，全面阐述了 IC3VA 算法的字级设计思想并证明正确性。最后设计了相关实验来验证所提出方法的有

效性，根据论文结果，作出了合理的分析和判断。

第四章详细介绍了基于隐式混合抽象的 IC3 组合框架。从对 IC3PA 算法和 IC3VA 算法的分析比较入手，第四章重点介绍了基于混合抽象的 IC3 组合框架的设计初衷、组合方式的设计、精化与泛化的针对性设计等，同时详细给出了 IC3 组合框架的工具实现设计方式，展示出方案的可行性。最后在多个测试数据集上进行了多个模型检测方法的硬件验证效果对比，根据实验的结果，论文从多个角度进行了详细的比较分析，并通过图表等形式进行了展示，并给出了合理的实验理解和结论。

第五章对本篇论文进行了一个较为全面的总结。完整地回顾了整篇论文的目标、思路、过程、贡献和结论，并对论文工作的未来做出了合理的展望。

# 第 2 章　相关工作

本章将详细介绍与论文研究目标相关的各项研究工作的发展和现状，主要包括硬件设计与验证的关系，模型检测技术及其在硬件验证中的应用发展概述，经典的比特级模型检测算法 IC3 算法的详细介绍，抽象与精化技术的介绍，以及一种具有代表性的字级模型检测算法 IC3PA 的详细介绍。本章的主要组织结构如下：第 2.1 节对硬件设计和硬件验证的概念进行了介绍，同时对模型检测技术进行了简要说明，最后阐述了模型检测技术与硬件验证的强相关性，并对硬件模型检测的应用发展作出了说明；第 2.2 节中详细介绍了一种十分有效和经典的比特级模型检测算法—IC3 算法；第 2.3 节对几种当前较为常见的抽象技术进行了介绍，最后重点介绍了反例制导的抽象精化技术；第 2.4 节则是对一种具有代表性的字级模型检测算法 IC3PA 进行了介绍，说明了其如何运用隐式抽象和谓词抽象等技术将 IC3 拓展运用到抽象状态空间；第 2.5 节对本章介绍的所有相关内容进行了总结。

## 2.1　硬件验证与模型检测

### 2.1.1　硬件验证

在电子学领域中，电子电路设计人员在早期是以特定的电子元件（晶体管、CMOS 等）为基础，充分考虑它们的特性和工艺对电路功能的影响，进行复杂电路的设计。而从上世纪 70 年代以来，集成电路的复杂度按照摩尔定律的发展趋势快速增长，电路规模和时序逻辑复杂性大幅度提升，这使得设计人员的工作量急剧增加，传统的设计方案变得不再实用。因此，硬件描述语言（Hardware Description Language，HDL）被制定出来作为对电路系统结构和行为的标准文本描述，它可以在结构级、行为级、寄存器级等几种不同抽象级别上对电路进行设计和描述，设计人员可以通过编写 HDL 的方式描述相关的电路设计，将主要精力投入到电路的逻辑和时序设计方面，提升工作效率。通过多年来的发展，VHDL（Very-High-Speed Integrated Circuit Hardware Description Language）和 Verilog 是电子工程领域内最为通用的两种硬件描述语言，图 2.1是一个简单的 Verilog 语言描述电路设计的样例。

根据论文[4] 中关于硬件验证的定义，硬件验证（Hardware Verification）是电路或系统根据给定的一组要求（Specification）运行的证明。在实际的电路设计中，这里的要求通常是符合某些硬件安全属性，确保电路正确且安全地运行。目前，在大多数 Verilog 或 VHDL 开发工具中，会集成相关的硬件仿真模块，用于模拟硬

mux21模块Verilog代码

```
module mux21(y, a, b, s);
  input a, b, s;
  output y;
    not u1(ns, s);
    and u2(sela, a, ns);
    and u3(selb, b, s);
    or u4(y, sela, selb);
endmodule
```

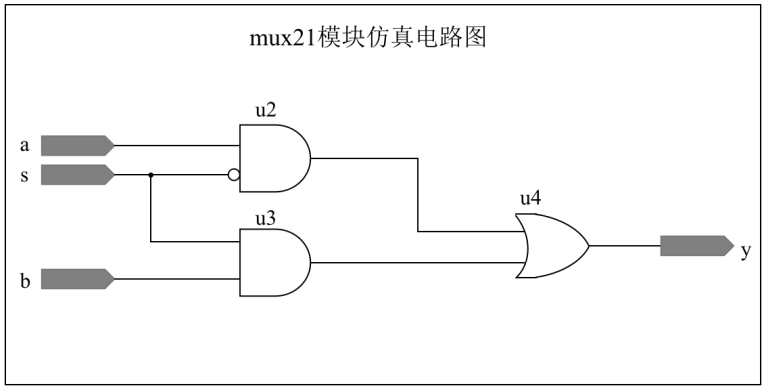



图 2.1　Verilog 语言描述电路设计示例

件描述语言对应的电子电路功能。按照论文[5]对于数字电路综合的介绍，数字电路可以在不同的抽象级别上进行表示，对于硬件设计人员来说，一般会先在较高级别的抽象层级进行硬件设计，然后硬件描述语言开发工具可以自动将高抽象级别的硬件设计转换为更低抽象级别的功能等效的设计表示，这个过程称之为综合（Synthesis），不同级别抽象与综合的关系如图 2.2所示。所以，经由综合或者人工设计，硬件设计可以在所有级别上都有等效的可仿真的电路表示，其中在设计和仿真验证中应用最多的抽象级别就是寄存器传输级别（Register Transfer Level，RTL）。传统的硬件验证方案就是仿真测试，使用集成开发工具中内置的综合仿真模块，在各个抽象级别模拟仿真硬件描述语言对应的电路功能，通过大量的测试，逐个检查其是否满足要求的安全属性。

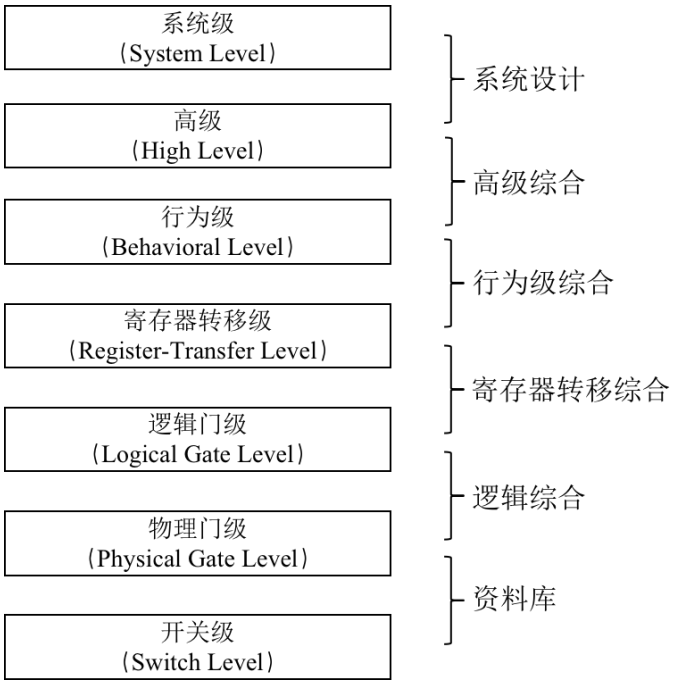



图 2.2　不同级别抽象与综合的关系

不难看出，通过仿真测试的方式进行硬件验证，这种方式只能发现硬件设计

语言中可能存在的问题，但并不能证明硬件设计语言中不存在问题。与此同时，随着集成电路规模的不断增长，大规模电路中会出现越来越多的各类模块，它们通过组合逻辑和时序逻辑互相关联交错，使得电路的状态空间变得十分庞大，此时导致系统整体的崩溃的原因可能只是一个小的细节错误。但相对有限的仿真测试样例，很难全面覆盖如此庞大的状态空间，电路设计中的深层次问题也就难以被发现。因此，考虑到电路安全的重要性，从应用角度来看，相比于测试仿真，必须采用一个更加严格、更具有可信性的方法来进行硬件验证。

### 2.1.2　模型检测

模型检测（Model Checking）[6]是由 E.M.Clarke, E.A.Emerson, J.P.Queille 和 J.Sifakis 等人提出的一种自动验证技术，它的主要思想是通过有限状态的迁移系统 $M$ 来定义系统的行为，通过逻辑公式 $P$ 来定义系统需要满足的规范和属性，从而可以通过判断 $M$ 是否为 $P$ 的一个模型，来确定系统是否满足给定的规范和属性，可以证明，对于有限状态空间的系统而言，这个问题是可判定的，因此这种验证技术可以在有限时间内完成。除此之外，模型检测技术的另一大特性是，如果系统 $M$ 不满足给定的属性 $P$，算法会给出一条 $M$ 如何违反属性 $P$ 的信息，即反例路径。而解决这个问题涉及到迁移系统的状态空间中的可达性计算，假设迁移系统中有 n 个状态元素，那么该系统的状态空间大小将为 n 的指数倍，当 n 的值比较大时，状态空间爆炸会给可达性计算带来巨大的挑战和问题。

关于这个问题，最早的进展之一是符号化模型检测[7]（Symbolic Model Checking），它用二进制决策图表示一组可达状态，并执行不动点算法来验证时序属性。另一个主要的进展是限界模型检测（Bounded Model Checking，BMC）[8]，它通过展开 k 次过渡关系生成命令公式，并调用 SAT 求解器检查该公式的可满足性，从而证明系统在 k 步或者 k 步内反例路径是否存在。虽然限界模型检测可以很快地找到反例，但是它只有在知道系统转移多少步能到达所有可达状态时才能给出属性的正确性证明，而该条件很难满足。而基于归纳推理的模型检测技术不仅可以检测属性是否成立，还能在属性成立时给出证明。归纳推理技术主要是得到一个归纳不变式来证明属性。比较高效的归纳推理模型检测算法有 k 步归纳算法[9]（K-induction）, 和 IC3 算法[1]。IC3 算法也被称为 PDR 算法（Property Directed Reachability，属性定向的可达性）[10], 由于其高效性，基于 IC3 衍生出的一系列算法已经成为模型检测领域中应用最成功的算法之一。

### 2.1.3　硬件模型检测

如 2.1.1 和 2.1.2 中所介绍的，随着电路复杂性的不断提升，传统的仿真测试用于硬件验证，不仅效率难以保证，方案本身的可信性也不够高，难以符合电路设计的高精密度要求，相比而言模型检测算法可以自动执行，在有限时间内完成，并且在验证出系统不满足要求的属性时还能提供反例路径，十分符合硬件验证的需要。因此模型检测技术自诞生起，就与硬件验证关系密切，而随着科技的不断进步，芯片、嵌入式设备等硬件行业的蓬勃发展，在硬件复杂性与日俱增的大环境下，模型检测技术在硬件验证上的优越性进一步彰显出来，被应用到了通信协议、嵌入式系统、电子电路设计等多个硬件相关方面的分析和验证之中，取得了十分不错的效果。因此，基于模型检测的硬件验证算法也逐渐在学术界和工业界受到更多的关注，各类硬件模型检测算法也开始出现在形式化验证领域内，成为主要研究方向之一。

在早期的硬件设计中，功能需求相对简单，电路规模也相对较小，因此整体电路所处理的数据规模也是比较小的，例如设计一个计算两个 8 比特位（bit）二进制数加法的电路，可能设计人员就会从每个比特位输入考虑，通过加入一些比特级的与或非门之间的组合，完成两个二进制数的加法，因此此时开发人员在编写硬件描述语言时，甚至早期硬件描述语言本身的特点，都是从比特级的角度看待数据和逻辑关系，进行代码的编写。AIGER[11]格式是由 Armin Biere 在 2007 年提出的一种比特级别的专门用于模型检测的中间文件格式，它具有简单紧凑的良好特点，得到了模型检测相关的研究人员一致认可，成为了最具代表性的比特级模型检测格式文件，一些硬件模型测试集也基本采用了这种格式来描述测试数据。

而随着社会科技进步和硬件产业的发展，硬件设计需求愈发复杂，与之对应的集成电路的规模陡增，此时比特级的硬件描述方式就难以满足功能和开发效率的需求了，因此 Verilog 等硬件描述语言也根据实际需求进一步发展，可以支持开发者在更高的抽象级别上描述硬件设计。诚然在更高级别上所做的硬件设计，可以等价转换为低级别的比特级硬件描述，但这个过程中是有抽象语义损失的，因为在低级别的硬件描述中，很难定义或者获取更高级别的抽象语义，此时如果我们依然沿用比特级的 AIGER 格式作为验证的中间格式，则会将这种高抽象级别的硬件设计代码中所包含的高级语义丢失掉，而这对于模型检测算法来说，将会是验证精度和效率上的损失，但从硬件发展趋势上看，高级别的硬件描述语言将会是主流。因此，为了保留这些高级别的抽象语义信息，对于验证和模型检测而言，需要新的验证中间文件格式。Btor2[12]格式是由 Niemetz 等人在 2018 年提出，针对 Btor[13]格式的升级版，可以将其视为 AIGER 格式的字级泛化，它支持位向量

和数组上的无量词公式的表示，支持类型声明，以及寄存器和存储器单元的显示的初始化。目前各类硬件模型检测的测试数据集上，都逐步加入了 Btor2 版本的测试数据，Btor2 格式也成为了目前最为主流的字级模型检测文件格式。

```
⟨num⟩      ::= positive unsigned integer (greater than zero)
⟨uint⟩     ::= unsigned integer (including zero)
⟨string⟩   ::= sequence of whitespace and printable characters without '\n'
⟨symbol⟩   ::= sequence of printable characters without '\n'
⟨comment⟩  ::= ';' ⟨string⟩
⟨nid⟩      ::= ⟨num⟩
⟨sid⟩      ::= ⟨num⟩
⟨const⟩    ::= 'const' ⟨sid⟩ [0-1]+
⟨constd⟩   ::= 'constd' ⟨sid⟩ ['-']⟨uint⟩
⟨consth⟩   ::= 'consth' ⟨sid⟩ [0-9a-fA-F]+
⟨input⟩    ::= ( 'input' | 'one' | 'ones' | 'zero' ) ⟨sid⟩ | ⟨const⟩ | ⟨constd⟩ | ⟨consth⟩
⟨state⟩    ::= 'state' ⟨sid⟩
⟨bitvec⟩   ::= 'bitvec' ⟨num⟩
⟨array⟩    ::= 'array' ⟨sid⟩ ⟨sid⟩
⟨node⟩     ::= ⟨sid⟩ 'sort' ( ⟨array⟩ | ⟨bitvec⟩ )
             | ⟨nid⟩ ( ⟨input⟩ | ⟨state⟩ )
             | ⟨nid⟩ ⟨opidx⟩ ⟨sid⟩ ⟨nid⟩ ⟨uint⟩ [⟨uint⟩]
             | ⟨nid⟩ ⟨op⟩ ⟨sid⟩ ⟨nid⟩ [⟨nid⟩ [⟨nid⟩]]
             | ⟨nid⟩ ( 'init' | 'next' ) ⟨sid⟩ ⟨nid⟩ ⟨nid⟩
             | ⟨nid⟩ ( 'bad' | 'constraint' | 'fair' | 'output' ) ⟨nid⟩
             | ⟨nid⟩ 'justice' ⟨num⟩ ( ⟨nid⟩ )+
⟨line⟩     ::= ⟨comment⟩ | ⟨node⟩ [⟨symbol⟩] [⟨comment⟩]
⟨btor⟩     ::= ( ⟨line⟩'\n' )+
```

图 2.3　Btor2 语法

图 2.3是 btor2 文件的语法定义。其中 *const*[*dh*], *one*, *ones*, *zero* 用来声明位向量常量。*input* 关键字用于声明给定类型的位向量或者数组变量。*one*、*ones* 和 *zero* 关键字分别表示最低位为 1、所有位为 1 和所有位为 0 的位向量。*array* 表示数组。*sort* 关键字用于定义位向量和数组类型，不仅可以指定多维数组，还可以支持未解释函数，浮点数以及其他类型。红色关键字表示和验证有关的信息，其中 *state* 用来定义寄存器和存储器，*init* 用来初始化他们，*next* 关键字用于定义它们的转移函数。没有声明转移函数的状态变量看作是基础输入 (primary input)，和用 *inputs* 声明的输入有同样的行为。*bad* 关键字用于声明属性，表示为安全性属性的反。不变式约束用关键字 *constraint* 声明。

综上所述，硬件模型检测技术是伴随着硬件设计技术和产业的发展一起共同进步的，过去的几十年中，在实际工业需求发展的驱动下，硬件描述语言经历了从比特级逐步向字级以及更高的抽象级别转变的过程，而其作为硬件模型检测的处理对象，这种转变也同样驱使着硬件模型检测算法的设计思想逐步从比特级转

为字级，从而更好地适应规模日益庞大的硬件设计的验证需求。而 IC3 算法作为目前应用最为广泛和成功的比特级硬件模型检测算法，如何将其拓展到字级层面，成为了硬件模型检测领域内的研究热点。在后面的章节中，将对此进行介绍。

## 2.2　IC3 算法介绍

IC3 算法[1]是 Aaron Bradley 等人在 2010 年提出的一种比特级的符号化模型检测算法，使用归纳推理的思想，尝试通过构造一个归纳不变式来验证属性。此算法在各类硬件模型检测数据集和赛事上取得了十分良好的表现，因此得到了广泛的关注和研究，各类基于 IC3 的模型检测算法此后也陆续出现，下面将对 IC3 算法做一个较为详细的介绍。

### 2.2.1　归纳不变式属性验证

首先介绍一下不变式验证问题的定义：给定公式 $P(X)$ 和转移系统 $S = \langle X, I, T\rangle$，不变式验证问题 $S \vDash P$ 指验证系统 $S$ 中所有有限路径 $s_0, s_1, ..., s_k$ 中的所有状态 $s_i$ 是否都满足属性 $P$。$P$ 代表好状态的集合，$\neg P$ 代表坏状态的集合。$S \vDash P$ 当且仅当 $S$ 中所有可达的状态都满足 $P$，即没有坏状态可达。

下面介绍归纳不变式和相对归纳不变式的定义：$F$ 是 $S$ 的归纳不变式，当且仅当：

$$\begin{aligned}&(1) I(X) \vDash F(X)\\&(2) F(X) \wedge T\left(X, X'\right) \vDash F\left(X'\right)\end{aligned} \tag{2.1}$$

所以，对于不变式验证问题，常用的一个解法是给系统找到一个归纳不变式 $F$，$F$ 需要满足 $F \vDash P$，又因为 $S \vDash F$，所以 $S \vDash P$。$F$ 归纳相关于公式 $\phi(X)$，当且仅当：

$$\begin{aligned}&(1) I(X) \vDash F(X)\\&(2) \phi(X) \wedge F(X) \wedge T\left(X, X'\right) \vDash F\left(X'\right)\end{aligned} \tag{2.2}$$

IC3 算法是一种基于 SAT 的有限状态系统的归纳不变式属性验证算法，该算法使用的系统状态变量为布尔变量，系统中公式为命题逻辑公式。介绍 IC3 算法之前先给出相关定义：一个布尔型变量叫做一个文字（literal）, 一组文字的合取叫做一个立方体（cube）, 对一个立方体取反，或者说是一组文字的析取叫做子句（clause）。如果一个立方体包含了所有的状态变量，该立方体是一个最小项（*minterm*）。

假设 $S$ 和 $P$ 分别代表转移系统和属性，IC3 算法通过找到一个公式 $F(X)$ 来

辅助证明 $S \vDash P$。其中 $F(X)$ 需要满足三个条件：

$$\begin{aligned}&(1) I(X) \vDash F(X)\\&(2) F(X) \wedge T\left(X, X'\right) \vDash F\left(X'\right)\\&(3) F(X) \vDash P(X)\end{aligned} \tag{2.3}$$

（1）和（2）用来说明 $F(X)$ 是系统的不变式，证明了 $F$ 是不变式后，说明 $S$ 的任何路径上的任何状态上都有 $F$ 成立，又因为（3），所以任何有限路径上的任何状态上都有 $P$ 成立，属性得证。

为了构建上述的归纳不变式 $F(X)$，IC3 算法维护了两个数据结构：

1. 一串公式序列，也可称作路径：$F_0(X), \cdots, F_k(X)$，$F_i$ 表示系统从初始状态迁移 $i$ 步内所能到达的状态的集合，由于 IC3 算法是通过不断加入不可达立方体的方式表示可达状态，所以 $F_i$ 通常表示为若干子句的集合，一个子句表示一个立方体的非，表示排除了一个不可达状态。该序列需要满足以下 4 点：

$$\begin{aligned}&(1) F_0 = I\\&(2) F_i \vDash F_{i+1}\\&(3) F_i(X) \wedge T\left(X, X'\right) \vDash F_{i+1}\left(X'\right)\\&(4) \forall i < k, F_i \vDash P\end{aligned} \tag{2.4}$$

2. 一个维护待阻塞的状态的队列，也叫做证明义务（proof-obligation）的队列。根据前文定义，序列中每个元素 $F_{i+1}$（帧）也是归纳相关于前一个元素 $F_i$。IC3 算法通过检查公式 $RelInd(F, T, c) \doteq F \wedge c \wedge T \wedge \neg c'$ 的可满足性来找到新的归纳相关于 $F$ 的子句，找到子句后，再把子句加入到其中，从而达到增强帧的目的。

### 2.2.2　IC3 算法流程

IC3 算法主要有两个阶段：阻塞阶段（blocking phase）和传播阶段（propagation phase），算法通过交替执行这两个阶段来进行流程的推进。算法2.1给出了一个 IC3 算法的伪码描述。

在阻塞阶段（算法2.1的 7-10 行），IC3 检查 $F_k \wedge \neg P$ 是否可满足，如果可满足，并且解为立方体 $c$，说明当前记录的 k 步内可达状态中有违反属性的坏状态 $c$。IC3 算法尝试阻塞该状态（第 9 行，算法 RECBLOCK）。该算法通过检查公式：

$$F_{k-1} \wedge T \wedge c' \tag{2.5}$$

若该公式不可满足，说明 $F_{k-1} \wedge T \vDash \neg c'$，即 $F_{k-1}$ 足以说明状态 $c$ 在 $F_k$ 的不可达性，所以 IC3 算法将 $\neg c$ 加入到 $F_k$ 中来增强 $F_k$。将公式中的 $F_{k-1}$ 换成 $F_{k-2}$ 得，

算法 2.1　IC3 算法

```
1  Function IC3(I,T,P):
2      if not I ⊨ P: return FALSE // 初始状态即不满足
3      F_0 := I // 路径的第一个元素即为初始公式
4      k := 1, F_k := ⊤ // 向路径添加一个新的帧（frame）
5      while TRUE:
6          // 阻塞阶段
7          while F_k ∧ ¬P is sat :
8              从模型 F_k ∧ ¬P 中抽取一个 cube c
9              if not RECBLOCK(c,k):
10                 return FALSE
11         // 传播阶段
12         k := k + 1, F_k := ⊤
13         for i := 1 to k − 1:
14             for each clause c ∈ F_i:
15                 if RelInd(F_i,T,c) is unsat :
16                     add c to F_{i+1}
17             if F_i = F_{i+1}:
18                 return TRUE // 得到归纳不变式 F_i，属性被证明
19 Function RECBLOCK(s,i):
20     if i = 0: return FALSE // 到达初始状态
21     while RelInd(F_{i−1},T,¬s) is sat :
22         从模型 RelInd(F_{i−1},T,¬s) 中抽取一个 cube c
23         if not RECBLOCK(c,i − 1): return FALSE
       // GENERALIZE 是标准 IC3 泛化
24     g = GENERALIZE(¬s, i)
25     for j := 1 to i: 将 g 加到 F_j
26     return TRUE
```

$F_{k-2} \wedge \neg c \wedge T \wedge c'$。由于序列需满足的条件（2），即 $F_{k-2}$ 比 $F_{k-1}$ 更强，可得该公式不可满足，所以可以将 $\neg c$ 加入到 $F_{k-1}$ 中来增强 $F_{k-1}$。依次类推，只要证明 $I \wedge c$ 不可满足，即从路径中排除 $c$。实际实现中，公式（2.5）会扩展为

$$F_{k-1} \wedge \neg c \wedge T \wedge c' \tag{2.6}$$

即扩展为判断 $c$ 是否归纳相关于 $F_{k-1}$。该扩展有两个考虑：

(1) 给公式加上一个子句后，公式更容易不可满足，提高了排除某些坏状态的机会。

(2) 该扩展可以看作是有限归纳推理：基础情况 $F_0 \wedge c$ 不可满足，假设 $F_{k-1} \wedge \neg c \wedge T \wedge c'$，即 $F_{k-1} \wedge \neg c \wedge T \vDash \neg c'$ 成立，因为 $F_{k-1}$ 比 $F_k$ 更强，所以推出 $F_{k-2} \wedge \neg c \wedge T \vDash \neg c'$ 成立，根据归纳法可得结论，每个帧经过一次转移后都

不会到达 $\neg c$。

反之，若公式可满足，说明 $F_{k-1}$ 近似得太过，不足以展示 $c$ 的不可达性。从 $F_{k-1} \wedge \neg c$ 的解中提取一个最小项 m，这就产生了一个新的证明义务 $(m, k-1)$。IC3 算法则继续对该证明义务检查公式（2.6）（第 23 行），来强化 $F_{k-1}$。如果 $F_{k-2}$ 也不足以证明 m 的不可达，IC3 算法就继续递归地往前找，直到阻塞成功，或者到达 $F_0$ 发现阻塞失败，返回 FALSE，属性被违反。

传播阶段（算法2.1的 12-18 行）从 $F_1$ 开始，将每个帧中的子句向前移动，让 $F_0(X), \cdots, F_k(X)$ 始终保持满足（2.4）的状态。如果在该过程中，有两个连续的帧变得相同，即 $F_i = F_{i+1}$，说明 IC3 算法到达了不动点，程序终止，属性为真，且 $F_i$ 是可以证明属性为真的归纳不变式。

算法 RECBLOCK 的第 24 行，GENERALIZE 算法的作用是对被阻塞成功的状态进行泛化，以得到更多同样不可达的状态。该算法是 IC3 算法的关键步骤，对算法的效率有着重要影响。

## 2.3　抽象和精化

在比特级的模型检测算法中，在接受低抽象级别描述的硬件设计语言（比如 RTL 描述）时，算法也会按照语言中的定义，单纯将数据视为一些比特和比特数组的集合，同时数据上的操作也以比特级的操作来描述和看待。然而在处理一些当下较为复杂的电路设计，如果遇到和数据非常相关的属性时，比如等价性检查，比特级的状态空间规模就会出现爆炸，导致算法的运行遇到阻碍或者效率低下，耗时完全无法接受。

字级的验证思想正是从更高抽象级别的角度看待模型检测，引入更多的高级别抽象语义，来缩减系统的状态空间，降低运算规模和复杂度。因此，在一些字级的模型检测算法中，抽象技术得到了充分使用，即通过分析验证属性和制定一系列抽象规则，来忽略和验证属性无关的部分，同时将符合特定规则的状态合并，从而达到缩减系统状态空间的目的。

然而需要注意的是，抽象技术固然可以缩减系统状态空间，但对原有状态空间如何抽象以及抽象程度如何选择，依据状态空间特性和规模的不同，并没有一个固定的答案。当抽象程度过大时，模型检测如果发现了一个反例，但在实际检查时完全有可能会发现这个反例并不存在于真实的状态空间中，而是由于抽象程度过大导致其出现在一个并不真实存在的抽象状态中，即此反例是一个伪反例。面对这种情况，论文[14]中设计了反例制导的抽象精化方法，依据伪反例信息提取出精度，对抽象空间作出精化（Refinement），降低抽象程度，同时规避此伪反例的

出现，此后不断利用抽象和精化，达到最适合的抽象状态空间完成验证。

在本节后续的几个小节中，将依次介绍几种当前较为流行和有效的抽象方法，在最后一小节中将介绍反例制导的抽象精化技术。

### 2.3.1　变量隐藏抽象

论文[15-16]中设计了变量隐藏抽象方法，变量隐藏抽象将状态变量集合划分为可见变量和不可见变量两个子集，其中可见变量子集被认为是对待验证属性来说很重要的集合。抽象模型中只保留可见变量的转移函数，不可见变量被抽象为无限制的基础输入。具体模型中，在可见变量上取值相同的具体状态被抽象到同一个抽象状态。由于消除了不可见变量上的逻辑限制，这种抽象会比原模型产生更多的执行路径。所以该抽象会导致伪反例出现，可以通过恢复不可见变量为可见的方式消除伪反例。

集合 $X = \{x_1, \cdots, x_m\}$ 是表示模型当前状态的有限变量集，集合 $X' = \{x'_1, \cdots, x'_m\}$ 表示模型下一个状态的变量集。集合 $X$ 和 $X'$ 上的赋值 $\tilde{X}$ 和 $\tilde{X}'$ 分别代表当前状态和下一个状态。模型表示为二元组 $\langle T, I\rangle$，其中 $T\left(X, X'\right)$ 代表转移关系，$I(X)$ 表示初始状态。即 $\tilde{X}$ 是初始状态当且仅当 $I(\tilde{X})$ 成立，$\left(\tilde{X}, \tilde{X}'\right)$ 代表状态转移当且仅当 $T\left(\tilde{X}, \tilde{X}'\right)$ 成立。具体状态转移函数可定义为每个变量的转移函数的合取，即

$$T = \bigwedge_{i=1}^{m} T_i\left(X, X'\right) \tag{2.7}$$

其中 $T_i$ 表示 $x_i \in X$ 上的转移函数，定义为 $x'_i = \delta_i(X)$。

对于变量隐藏抽象，假设 $X_v = \{x_1, \cdots, x_n\} \subseteq X$ 表示可见变量集合，则不可见变量集合为 $X_{inv} = \left(X \backslash X_v\right)$。对于 $x_i \in X_{inv}$，其转移函数被抽象为 $T_i = true$。变量隐藏抽象下的抽象模型 $\left\langle T_{\mathcal{V}}, I_{\mathcal{V}}\right\rangle$ 定义为：

$$\begin{aligned} T_{\mathcal{V}} &= \textstyle\bigwedge_{i=1}^{n} T_i\left(X, X'\right) \\ I_{\mathcal{V}} &= \exists X_{inv}.I(X) \end{aligned} \tag{2.8}$$

变量隐藏抽象的抽象方式的一种直观表述是，在可见变量集合上取值相同的具体状态抽象到同一个抽象状态。如果两个具体状态之间有转移，那么它们所属的抽象状态之间存在抽象转移。由于 $T_{\mathcal{V}}(X, X'_v)$ 可能会依赖于 $X_{inv}$ 中某些隐藏的当前状态的变量，这些变量被当做自由输入，而模型检测中，自由输入在计算时通常会被存在量化。所以可以通过存在量词消除 $T_{\mathcal{V}}$ 中的隐藏变量，即：

$$\widehat{T}_{\mathcal{V}} = \exists X_{inv}.T_{\mathcal{V}}\left(X, X'_v\right) \tag{2.9}$$

### 2.3.2　谓词抽象

相比于变量隐藏抽象，谓词抽象[17]是一种更加灵活的抽象方式，普适性较强，已经被应用到软件和硬件验证领域，取得了不错的效果。谓词抽象给出定义在具体状态变量集合上的一组谓词，每个谓词又对应到一个布尔变量。利用这些谓词，将模型从具体的状态空间（以具体变量集合做基）映射到抽象的状态空间上（以布尔变量集合做基）。

集合 $\mathbb{P}$ 表示建立在变量集 $X$ 上的一组谓词，即对于每个 $p \in \mathbb{P}$，都是定义在变量集合 $X$ 上的公式。对每个 $p \in \mathbb{P}$，定义一个新的布尔变量 $x_p$，称作谓词名称或者是抽象变量。这些谓词变量的集合记作 $X_{\mathbb{P}}$。谓词抽象下的抽象关系 $H_{\mathbb{P}}$ 则定义为：

$$H_{\mathbb{P}}\left(X, X_{\mathbb{P}}\right) \doteq \bigwedge_{p \in \mathbb{P}} x_p \leftrightarrow p(X) \tag{2.10}$$

对于一个公式 $\phi(X)$，它相对于谓词集合 $\mathbb{P}$ 的抽象版本 $\widehat{\phi}_{\mathbb{P}}$ 记为：

$$\widehat{\phi}_{\mathbb{P}}\left(X_{\mathbb{P}}\right) \doteq \exists X.\left(\phi(X) \wedge H_{\mathbb{P}}\left(X, X_{\mathbb{P}}\right)\right) \tag{2.11}$$

即在原公式中添加抽象关系 $H_{\mathbb{P}}$，并存在量化变量 $X$。同样的，对于定义在 $X$ 和 $X'$ 上的抽象关系 $\phi(X, X')$，其抽象版本为：

$$\widehat{\phi}_{\mathbb{P}}\left(X_{\mathbb{P}}, X'_{\mathbb{P}}\right) \doteq \exists X, X'.\left(\phi\left(X, X'\right) \wedge H_{\mathbb{P}}\left(X, X_{\mathbb{P}}\right) \wedge H_{\mathbb{P}}\left(X', X'_{\mathbb{P}}\right)\right) \tag{2.12}$$

对于 $X$ 的一个解释 $\mu$，其抽象版本 $\hat{\mu}$ 定义为解释 $\mu_{X_{\mathbb{P}}}\left[x_p := \mu(p)\right]$。所以，对于一个转移系统 $S \doteq \langle X, I, T\rangle$，其相应于谓词集合 $\mathbb{P}$ 的抽象版本是：

$$\widehat{S}_{\mathbb{P}} \doteq \left\langle X_{\mathbb{P}}, \widehat{I}_{\mathbb{P}}, \widehat{T}_{\mathbb{P}}\right\rangle \tag{2.13}$$

例如，假定一个有整数变量 $\{c, d\}$ 的转移系统，每次转移 $d$ 减 1，$c$ 增加当前 $d$ 值，其形式化定义为：$S = \langle\{c, d\}, c = 0 \wedge d = 0, c' = c + d \wedge d' = d + 1\rangle$，属性形式化定义为：$P = d \leqslant 3 \vee \neg(c \leqslant d)$。对于该系统，假设 $\mathbb{P} = \{c = 0, d = 0, d \leqslant 3, c \leqslant d\}$ 是从其初始条件和属性中提取的谓词集合, 并且对应的谓词变量集合定义是 $X_{\mathbb{P}} = \left\{x_{c=0}, x_{d=0}, x_{c\leqslant 0}, x_{c\leqslant d}\right\}$。那么谓词抽象后的系统 $\widehat{S}_{\mathbb{P}}$ 定义为：

$$\left\langle\left\{x_{d=0}, x_{c=0}\right\}, x_{d=0} \wedge x_{c=0}, \left(x_{d=0} \rightarrow \left(\neg x'_{d=0} \wedge x_{c=0} \leftrightarrow\ \ x'_{c=0}\right)\right) \wedge \left(\neg x_{d=0} \rightarrow \left(x_{c=0} \rightarrow \neg x'_{c=0}\right)\right)\right\rangle$$

抽象属性 $\widehat{P}_{\mathbb{P}}$ 定义为：

$$x_{d\leqslant 3} \vee \neg x_{c\leqslant d}$$

这种构造下，若 $\widehat{S}_{\mathbb{P}} \vDash \widehat{P}_{\mathbb{P}}$，则 $S \vDash P$，但是反过来不成立，即 $\widehat{S}_{\mathbb{P}}$ 中可能会存在伪反例。因此谓词抽象虽然灵活，但是抽象计算更加昂贵，抽象状态的个数是谓词个数的指数倍。当谓词数量变多时，抽象模型的规模也会迅速增大，抽象计算的

时间也相应增加。为了缓解这一问题，论文[18-19]提出笛卡尔抽象，但是该抽象会损失一些精度。

### 2.3.3　语法制导的抽象

论文[20]中设计了语法制导的抽象方法，它使用问题的字级描述中出现的术语（terms）的集合来编码抽象状态空间，其中每个抽象状态描述了术语之间的等价关系。

例如, 假设对于系统 $\mathcal{P}$，其形式化定义为：

$$\mathcal{P} = \left\langle \{u, v\}, (u = 1) \wedge (v = 1), \left(u' = ite(u < v, u + v, v + 1)\right) \wedge \left(v' = v + 1\right), ((u + v) \neq 1) \right\rangle$$

其中 $u, v$ 是 $k$ 比特位宽。所以 $\mathcal{P}$ 有一个谓词 $(u < v)$ 和 5 个字 $(1, u, v, u+v, v+1)$。对于一个具体状态，其相应的抽象状态是通过计算出这些谓词和字的值，根据它们之间的等价和不等关系进行分区得到的。例如具体状态 $s := (u, v) = (1, 2)$，由于 $u = 1$、$v = 2 \neq 1$、$u + v = v + 1 = 3$，所以它的抽象状态是 $\hat{s} := (u < v) \wedge \{1, u|v|u+v, v+1\}$，即相同的字分到同一个块。

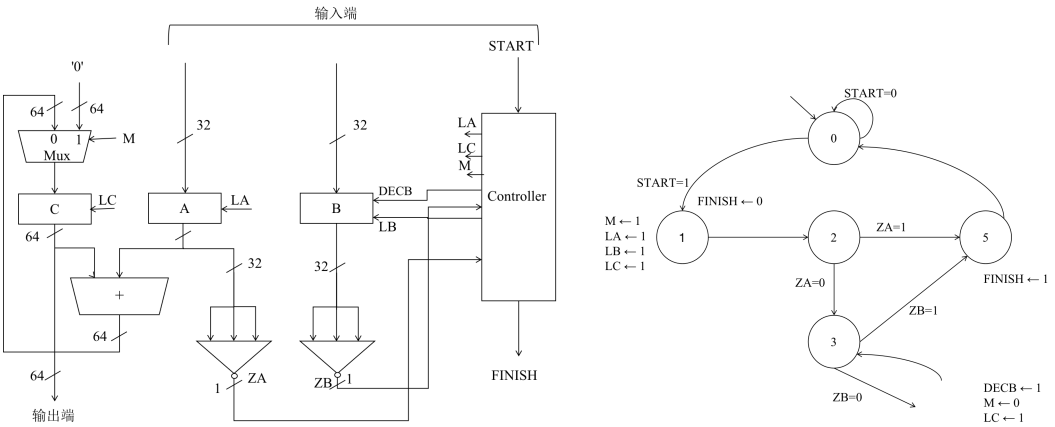
语法制导的抽象的特点是，抽象空间的大小和字的宽度无关，同时也考虑了这些字之间的等价关系。假设一个系统 $\mathcal{P}$ 中所有变量共有 $m$ 个比特，那么具体状态个数为 $2^m$。如果抽象后的系统有 $p$ 个谓词和 $n$ 个字，那么抽象状态的个数为 $2^p \times B_n$，$B_n$ 是贝尔数（在 n 个项上的划分的数量）。可见，基于语法制导的抽象的抽象状态空间大小和变量的位宽无关，该抽象隐藏了不相关的位级细节，又保留了原始问题中更高层次的相等关系。

### 2.3.4　位宽缩减抽象

位宽缩减抽象方法在论文[21-22]中被提出，该抽象将数据路径和控制逻辑分离开来，在保证待验证属性保留的前提下，将数据路径的宽度缩小。该抽象方法不会导致伪反例出现，所以没有精化过程，并且可以很容易地和任何比特级别的模型检测算法相结合。但是此种抽象方法只能在数据路径可以缩减宽度的情况下使用，比较受限。

电路的数据路径（datapath）由执行单元组成，如算术逻辑单元（Arithmetic and Logic Unit，ALU）、寄存器和它们之间的通信路径。控制逻辑实现数据路径操作的排序。该方法把控制逻辑和 datapath 分开，把数据路径的宽度缩小到下界。这个下界可以保证待验证的属性被保留，因此这种方法不会导致伪反例出现。

举例说明位宽缩减抽象，图 2.4是实现 A*B 乘法的电路，图中左边部分是 datapath，右边部分是控制器的状态转移图。实现乘法的方式是把 A 累加 B 次。这个电路三个寄存器 A、B、C 位数一共 128 位，可能的状态数是 2 的 128 次方。敏感




(a) datapath　　(b) 控制器状态转移图

图 2.4　A*B 数据路线与控制器状态转移图示例

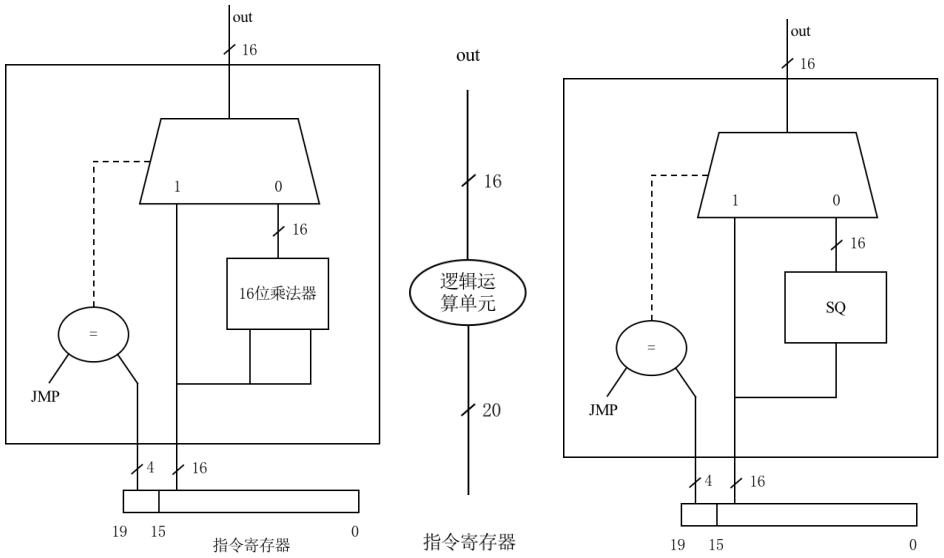
信号 START 为高位时，进入状态 1，LA、LB、LC 和 M 都输出 1，触发相应的单元。若 ZA=1，即 A=0，则直接结束，FINISH 输出 1。否则进入状态 3，若 ZB=1，即 B=0，也直接进入状态 5。当 A，B 都不为 0 时，进入到状态 4，进行加 A 操作并存入 C，并把 B 减 1。如果要验证的属性是 FINISH 信号最终进入高位，这个时候可以把 A 和 B 抽象成 1bit 宽度，C 抽象成 2bit 宽度。则 datapath 可能的状态数量是 2 的 4 次方，即 16 个。如果要验证的属性是：ZB 最终会是高位，即 B 最终会减到 0，这个时候寄存器 A 可以缩减 1bit，寄存器 C 可以缩减到 32bit，寄存器 B 仍需要 32bit。因为这个属性必须对于寄存器 B 任何可能的输入值都要成立，所以 B 的位数不能少。最后 datapath 可能的状态数量缩减到 2 的 65 次方。

### 2.3.5　术语级抽象

论文[23-24]等先后提出并发展了术语级（term-level）抽象方法，它是一种在形式逻辑中抽象词级设计的技术，这种方法抽象掉了数据表示和数据操作的细节，把数据看作是符号术语，将数据用抽象术语建模，函数块建模为未解释函数，存储器用合适的存储器理论进行建模。

如果一个字级的设计使用了以下三种抽象中的一种，就可以看作是使用了术语级抽象：

- 函数抽象（Function abstraction）：将位向量（bit-vector）操作（比如提取和连接）或者是整个用来计算位向量的模块抽象成未解释函数。这个未解释函数的输入或者输出既可以是位向量又可以是抽象的术语。这个未解释函数需要满足函数的要求，即相同的输入有确定且相同的输出。论文[24]中使用图 2.5展示了一个算术逻辑单元的三种版本的设计。图2.5(a)是一个字级的电路：输入是一条指令，前 4bit 是操作码，后 16bit 是数据。如果操作码是 jump




(a) 原始字级电路　　(b) 完全未解释电路　　(c) 部分未解释电路

图 2.5　论文[24]中算术逻辑单元的三种版本设计示意图

操作，则直接输出 16bit 的数据，否则输出 16 位乘法器的结果。这里可以把整个模块看作是一个未解释函数（对应图2.5(b)），或者是把 16 位乘法器抽象成一个未解释函数 SQ（对应图2.5(c)）。如果函数的输出是 1bit，则可以抽象成未解释谓词。

- 数据抽象（Data abstraction）：位向量表达式可以被抽象成某个域上的抽象的术语（比如抽象到整数域），例如，对于常量 001，可以抽象记作 $K1_3$（把 001 抽象到整数域，并用 $K1_3$ 去描述这个常量位向量），对于一个 8bit 位宽的变量 $x$，抽象记做 $x_1$。
- 内存抽象（Memory abstraction）：对内存建模是为了做到读取某个地址的内容和刚写进去的内容是一致的，这样可以避免存取不一致带来的错误。常用的建模方式是用 read/write 函数或者是 lambda 表达式。

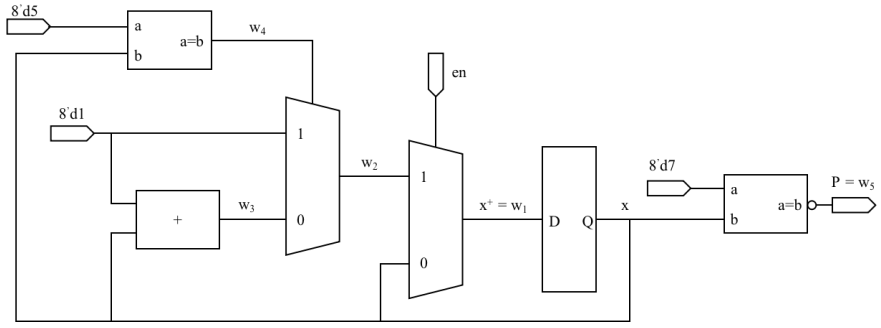



图 2.6　论文[25]中简单时序电路电路图示例

图 2.6是论文[25]中一个简单时序电路电路图例子，用来说明术语级抽象。对

于图中时序电路的转移函数和初始条件及验证属性形式化描述为：

$$\begin{aligned}
&T\left(\mathbf{w},\mathbf{x},en,\mathbf{x}^{+}\right)=\left(w_1=en?w_2:x\right)\wedge\left(w_2=w_4?8'\mathrm{d}1:w_3\right)\wedge\\
&\left(w_3=x+8'\mathrm{d}1\right)\wedge\left(w_4=\left(x==8'\mathrm{d}5\right)\right)\wedge\left(x^{+}=w_1\right)\\
&I(\mathbf{x})=\left(x=8'\mathrm{d}1\right)\\
&P(\mathbf{w},\mathbf{x},en)=w_5\wedge\left(w_5=\neg\left(x==8'\mathrm{d}7\right)\right)
\end{aligned}\tag{2.14}$$

经过函数抽象和数据抽象后的转移系统为：

$$\begin{aligned}
&\hat{T}\left(\hat{\mathbf{w}},\hat{\mathbf{x}},en,\hat{\mathbf{x}}^{+}\right)=\left(\hat{w}_1=en?\hat{w}_2:\hat{x}\right)\wedge\left(\hat{w}_2=w_4?\boldsymbol{K}1:\hat{w}_3\right)\wedge\\
&\left(\hat{w}_3=\mathrm{ADD}(\hat{x},\ \mathrm{K}1)\right)\wedge\left(w_4=(\hat{x}==\mathrm{K}5)\right)\wedge\left(\hat{x}^{+}=\hat{w}_1\right)\\
&\hat{I}(\hat{\mathbf{x}})=(\hat{x}=\mathrm{K}1)\\
&\hat{P}(\hat{\mathbf{w}},\hat{\mathbf{x}},en)=w_5\wedge\left(w_5=\neg(\hat{x}==\mathrm{K}7)\right)
\end{aligned}\tag{2.15}$$

$w_4$ 没有被抽象，因为它只有一比特位。

### 2.3.6　反例制导的抽象精化

正如本节开头时提到的，部分抽象方法，可能因为抽象程度选取过大，会在削减状态空间的同时包含了比原系统更多的路径，从而导致伪反例出现，即抽象模型中可成立但是具体模型中不存在的路径。反例制导的抽象精化技术（Counterexample-guided Abstraction Refinement，CEGAR）[14]是一种自动且动态调整抽象层次的框架，可以应对这种伪反例的情况。

具体来说，该框架是一个在抽象状态空间中找反例的循环的过程。若抽象状态空间下找不到反例，由于抽象状态空间是具体状态空间的上近似，则待验证属性在具体状态状态下也不会被违背。反之，若找到反例路径，则进行可达性检查，即在具体模型上检查该路径是否成立。若成立，则属性被违背，若不成立，则说明抽象程度过大，导致伪反例出现，框架从伪反例中提取精度，排除此反例路径并降低模型的抽象层次，然后进行下一轮找反例的迭代。

因此可以看到，CEGAR 中的如何依据伪反例来提取精度对精化过程是十分关键的，有一些方法可以实现这种精化，例如挖掘谓词[26]，静态计算不变量[27]或不变量生成[28]，但最为成功的依然是基于 Craig 插值的方法[29-30]。

Craig 插值是一种来自于数理逻辑的方法，它可以为两个不能同时成立的公式生成一个插值公式，同时可以保证此插值公式所含信息少于第一个公式，但仍然与第二个公式相矛盾。形式化的描述为：对于一组有序的公式对（$\varphi$，$\psi$），且 $\varphi\wedge\psi$ 不成立，则一个 Craig 插值公式 $\tau$ 必须满足以下三个条件：

(1) 蕴含关系 $\varphi\Rightarrow\tau$ 成立；

(2) $\tau\wedge\psi$ 不成立；

(3) $\tau$ 中只包含同时在 $\varphi$ 和 $\psi$ 中出现的符号；

同样地，可以将其扩展到公式序列中，得到与上述相类似的公式序列的 Craig 插值定义。而公式序列插值则可以通过 SMT（Satisfiability Modulo Theories，模块可满足性理论）技术有效地计算出来[31]。至此，通过 Craig 插值，CEGAR 便可以从伪反例中提取出精度，完成精化过程，继续后续的迭代流程。

## 2.4　IC3PA 算法介绍

IC3 算法作为最具代表性和成功的比特级硬件模型检测算法，顺应当前硬件模型检测研究的发展趋势，如何将其扩展到字级已经成为了研究者们关注的焦点。IC3PA[2-3] 正是一种基于隐式抽象理念和谓词抽象将 IC3 拓展运用到抽象状态空间的方法，并取得了不错的效果，下面将对其进行介绍。

### 2.4.1　隐式抽象

正如 2.2 中提到的，为了将模型检测技术运用到抽象状态空间，需要提前计算出抽象空间中经过抽象后的状态转移函数，例如谓词抽象会出现存在量词，所以需要计算一个表示抽象系统的转换关系的无量词公式。常用的方法是消除抽象定义中的量词。当抽象状态空间有限时，例如谓词抽象，抽象转移函数可以通过枚举满足抽象定义的模型来得到。但是有些情况下，量词消去技术对谓词抽象而言都不可用[32-33]，这就给该技术造成了瓶颈。

隐式抽象技术[34]设计了避免量词消去技术来解决抽象模型检测的方法，其核心思路是将抽象的定义嵌入到模型检测方法的编码中。因此可以在不计算出抽象模型的转移函数的前提下验证抽象系统的正确性。该方法可以用于多种抽象，例如谓词抽象，投影抽象等，下面基于隐式谓词抽象说明。

假设抽象空间中有一条长度为 k 的路径，传统的谓词抽象从初始状态开始将抽象转移关系（公式（2.13））展开 k 次，从而对路径进行编码，而隐式谓词抽象的核心是利用公式（2.16）将抽象路径编码为无量词公式：

$$EQ_{\mathbb{P}}(X,\overline{X}) \doteq \bigwedge_{p\in\mathbb{P}} p(X) \leftrightarrow p(\overline{X}) \tag{2.16}$$

该公式的含义是，当两个具体状态可使得抽象空间的谓词变量取值同真或同假时，它们应该被抽象到一个抽象状态。将该抽象定义嵌入到路径中，得到的路径编码为：

$$Path_{\mathbb{P}}^{k} \doteq \bigwedge_{1\leqslant h<k} \left(T\left(\overline{X}^{h-1},X^{h}\right)\wedge EQ_{\mathbb{P}}\left(X^{h},\overline{X}^{h}\right)\right)\wedge T\left(\overline{X}^{k-1},X^{k}\right) \tag{2.17}$$

图 2.7是该编码方式示意图。其中 $X^{h-1}$ 和 $\overline{X}^{h-1}$ 同属一个抽象状态，$\overline{X}^{h-1}$ 经过一步转移到达具体状态 $\overline{X}^{h}$，$X^{h}$ 经过一步转移到达具体状态 $X^{h+1}$。而 $\overline{X}^{h}$ 和 $X^{h}$ 不一定相等，它们同样只需要满足属于同一个抽象状态即可，即 $EQ_{\mathbb{P}}(X^{h},\overline{X}^{h})$ 成立。

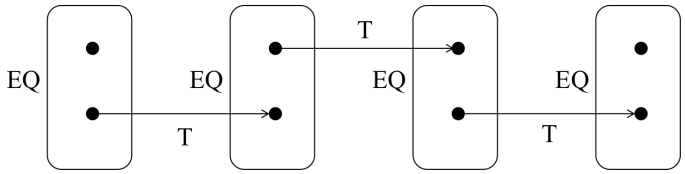



图 2.7　隐式抽象编码方式示意图

可见，隐式谓词抽象定义的路径不是将预先定义好的抽象转移关系进行连续的 k 步展开，而是将其编码为一连串彼此断开的转移序列，序列之间通过 $EQ_{\mathbb{P}}$ 关系进行连接。

所以，基于隐式谓词抽象的 BMC 模型检测公式可定义为：

$$BMC_{\mathbb{P}}^{k} \doteq I\left(X^{0}\right) \wedge EQ_{\mathbb{P}}\left(X^{0},\overline{X}^{0}\right) \wedge Path_{\mathbb{P}}^{k} \wedge EQ_{\mathbb{P}}\left(X^{k},\overline{X}^{k}\right) \wedge \neg P\left(\overline{X}^{k}\right) \quad (2.18)$$

该方法只考虑和搜索相关的抽象状态空间，求解器只需要求解可满足性问题，而不需要枚举抽象状态空间上所有可能的转移。

### 2.4.2　IC3PA

IC3 算法中最重要的环节是泛化，如果立方体泛化得越简单，含有的文字越少，那么可以剔除的不可达状态就越多。字级 IC3 算法和比特级 IC3 算法最关键的不同也在于此。若公式（2.6）可满足，可计算出 $s$ 相对于 $T$ 的前像立方体 $p$，即 $p$ 中的每个状态都存在到 $s$ 中的某个状态的转移。在比特级，将公式（2.6）的可满足解 $\mu$ 中出现的表示下一个状态的变量丢弃, 即可得到 $p$。但是，在字级情况下，当前状态和下一个状态变量之间的关系受到 SMT 理论的约束，此时求一个状态的前象变得困难。

IC3PA[2-3]提出一种将隐式谓词抽象、IC3 以及反例制导的抽象精化结合的算法，从系统的字级描述表示出发，在经过谓词抽象后的空间上进行 IC3 算法。IC3PA 的主要思路是模拟 IC3 算法在谓词集合 $\mathbb{P}$ 上执行，并且用隐式谓词抽象去避免抽象转移关系的计算。在 IC3PA 中，子句，帧和立方体都是定义在抽象集合 $X_{\mathbb{P}}$ 上。如果该抽象空间上发现伪反例，从伪反例中提取精度精化抽象，再在新的抽象模型上执行 IC3 算法。该算法的实现基于 SMT，且和具体理论无关，所以可以使用于很多理论，并且实验证明适用于大量谓词的情况。

集合 $\mathbb{P}$ 包含初始条件 $I$ 和属性 $P$ 中的所有谓词，所以 $I$ 和 $P$ 都是 $\mathbb{P}$ 中谓词

的结合。如果一个公式 $\phi$ 是谓词集合 $\mathbb{P}$ 中的谓词的结合，那么他的抽象版本 $\widehat{\phi}$ 逻辑等价于公式

$$\phi\left[X_{\mathbb{P}}/\mathbb{P}\right] \wedge \exists X.H_{\mathbb{P}}\left(X, X_{\mathbb{P}}\right) \tag{2.19}$$

其中 $\phi\left[X_{\mathbb{P}}/\mathbb{P}\right]$ 表示将 $\phi$ 中每个出现在 $\mathbb{P}$ 中的谓词 $p(X)$ 替换成相对应的抽象变量 $X_p$。而 $\exists X.H_{\mathbb{P}}\left(X, X_{\mathbb{P}}\right)$ 和被抽象的公式无关，它定义了谓词和谓词变量之间的关系。当运行在抽象空间 $\widehat{S}$ 上时，IC3PA 将公式（2.6）编码为无量词公式：

$$\begin{aligned} AbsRelInd(F,T,c,\mathbb{P}) \doteq & F\left(X_{\mathbb{P}}\right) \wedge c\left(X_{\mathbb{P}}\right) \wedge H_{\mathbb{P}}\left(X, X_{\mathbb{P}}\right) \wedge H_{\mathbb{P}}\left(X', X'_{\mathbb{P}}\right) \wedge \\ & EQ_{\mathbb{P}}(X, \overline{X}) \wedge T\left(\overline{X}, \overline{X}'\right) \wedge EQ_{\mathbb{P}}\left(\overline{X}', X'\right) \wedge \neg c\left(X'_{\mathbb{P}}\right) \end{aligned} \tag{2.20}$$

因为 $F\left(X_{\mathbb{P}}\right) \wedge c\left(X_{\mathbb{P}}\right) \wedge H_{\mathbb{P}}\left(X, X_{\mathbb{P}}\right)$ 等价于 $F(X) \wedge c(X)$，$H_{\mathbb{P}}\left(X', X'_{\mathbb{P}}\right)$ 等价于 $\neg c(X')$，所以公式（2.20）又可写作：

$$AbsRelInd(F,T,c,\mathbb{P}) \doteq F(X) \wedge c(X) \wedge EQ_{\mathbb{P}}(X, \overline{X}) \wedge T\left(\overline{X}, \overline{X}'\right) \wedge EQ_{\mathbb{P}}\left(\overline{X}', X'\right) \wedge \neg c(X)' \tag{2.21}$$

该公式使用隐式谓词抽象避免了抽象转移关系 $\widehat{T}$ 的构造。

IC3PA 和 IC3 有相同的结构，即外层分为两个阶段，阻塞阶段和传播阶段。由于 IC3PA 利用 CEGAR 框架，所以当找到抽象状态空间上的反例路径时，IC3PA 会检查是否可以具体化，若不可具体化，说明是伪反例，IC3PA 执行精化方法提取谓词消除该反例。注意，IC3PA 中精度是单调增加的，抽象是完全增量的，抽象关系逐步增强，所以之前算出的子句在精化后也可以保留。此外，IC3PA 的精化算法也是独立于 IC3PA 算法的，目前可以适配的各类精化方案有很多[35-36]，只要提取的精度可以消除该伪反例即可。

为了方便理解 IC3PA 算法的流程，论文[3]中举了如下的例子来说明。

假设系统 $S = \langle\{c, d\}, c = 0 \wedge d = 0, c' = c + d \wedge d' = d + 1\rangle$，其中 $u$ 和 $v$ 是整数变量。待验证的属性是 $P \doteq (d \leqslant 3) \vee \neg(c \leqslant d)$。初始时，从 $P$ 和 $I$ 中提取的谓词集合是 $\mathbb{P}_0 \doteq \{(c = 0), (d = 0), (d < 3), (c \leqslant d)\}$，第一个帧 $F_0 \doteq x_{c=0} \wedge x_{d=0}$。则 IC3PA 的算法流程如下：

(1) 算法检查 $F_0 \wedge H_{\mathbb{P}}\left(X, X_{\mathbb{P}}\right) \wedge \neg P\left[X_{\mathbb{P}}/\mathbb{P}\right]$，该公式不可满足，增加一个空的帧 $F_1$。

(2) 算法检查 $F_1 \wedge H_{\mathbb{P}}\left(X, X_{\mathbb{P}}\right) \wedge \neg P\left[X_{\mathbb{P}}/\mathbb{P}\right]$，得到解 $c_0 = x_{c=0} \wedge \neg x_{d=0} \wedge \neg x_{d\leqslant 3} \wedge x_{c\leqslant d}$，满足该公式。所以 IC3PA 试图在 $F_0$ 中阻塞 $\left(c_0, 1\right)$。此时检查公式 $AbsRelInd\left(F_0, T, c_0, \mathbb{P}_0\right)$，发现其实际上如公式（2.22）展开，可知公式 $AbsRelInd\left(F_0, T, c_0, \mathbb{P}_0\right)$ 不可满足，因此 IC3PA 泛化 $c_0$ 到 $\neg x_{d\leqslant 3}$。此时

IC3PA 把 $\neg x_{d\leqslant 3}$ 的反加入到 $F_1$ 中，则 $F_1$ 和 $\neg P$ 无交集。然后 IC3PA 增加帧 $F_2$，并执行传播阶段。

(3) IC3PA 检查 $F_2 \wedge H_{\mathbb{P}}\left(X, X_{\mathbb{P}}\right) \wedge \neg P\left[X_{\mathbb{P}}/\mathbb{P}\right]$，找到解 $\neg x_{c=0} \wedge \neg x_{d=0} \wedge \neg x_{d\leqslant 3} \wedge x_{c\leqslant d}$，执行阻塞算法后发现了伪反例 $(\neg x_{c=0} \wedge \neg x_{d=0} \wedge \neg x_{d\leqslant 3} \wedge x_{c\leqslant d}, 2)), (x_{c=0} \wedge \neg x_{d=0} \wedge x_{d\leqslant 3} \wedge x_{c\leqslant d}, 1))$，$(x_{c=0} \wedge x_{d=0} \wedge x_{d\leqslant 3} \wedge x_{c\leqslant d}, 0))$。经检查，该反例不可具体化，是伪反例，IC3PA 精化求得新的精度 $(d \leqslant 2)$。

(4) IC3PA 继续进行发现坏状态，阻塞，传播的过程。

$$
\begin{aligned}
AbsRelInd\left(F_0, T, c_0, \mathbb{P}_0\right) \doteq & x_{c=0} \wedge x_{d=0} \wedge & \left[F_0\left(X_{\mathbb{P}}\right)\right] \\
& x_{c=0} \wedge \neg x_{d=0} \wedge \neg x_{d\leqslant 3} \wedge x_{c\leqslant d} \wedge & \left[c_0\left(X_{\mathbb{P}}\right)\right] \\
& x_{d=0} \leftrightarrow (d=0) \wedge x_{c=0} \leftrightarrow (c=0) \wedge & \left[H_{\mathbb{P}}\left(X, X_{\mathbb{P}}\right)\right] \\
& x_{d\leqslant 3} \leftrightarrow (d \leqslant 3) \wedge x_{c\leqslant d} \leftrightarrow (c \leqslant d) \wedge & \\
& x'_{d=0} \leftrightarrow \left(d'=0\right) \wedge x'_{c=0} \leftrightarrow \left(c'=0\right) \wedge & \left[H_{\mathbb{P}}\left(X', X'_{\mathbb{P}}\right)\right] \\
& x'_{d\leqslant 3} \leftrightarrow \left(d' \leqslant 3\right) \wedge x'_{c\leqslant d} \leftrightarrow \left(c' \leqslant d'\right) \wedge & \\
& (d=0) \leftrightarrow (\overline{d}=0) \wedge (c=0) \leftrightarrow (\overline{c}=0) \wedge & \left[EQ_{\mathbb{P}}(X, \overline{X})\right] \\
& (d \leqslant 3) \leftrightarrow (\overline{d} \leqslant 3) \wedge (c \leqslant d) \leftrightarrow (\overline{c} \leqslant \overline{d}) \wedge & \\
& \left(\overline{c'}=\overline{c}+\overline{d}\right) \wedge \left(\overline{d'}=\overline{d}+1\right) \wedge & \left[T\left(\overline{X}, \overline{X}'\right)\right] \\
& \left(\overline{d'}=0\right) \leftrightarrow \left(d'=0\right) \wedge \left(\overline{c'}=0\right) \leftrightarrow \left(c'=0\right) \wedge & \\
& \left(\overline{d'} \leqslant 3\right) \leftrightarrow \left(d' \leqslant 3\right) \wedge \left(\overline{c'} \leqslant \overline{d'}\right) \leftrightarrow \left(c' \leqslant d'\right) \wedge & \left[EQ_{\mathbb{P}}\left(\overline{X}', X'\right)\right] \\
& \neg\left(x'_{c=0} \wedge \neg x'_{d=0} \wedge \neg x'_{d\leqslant 3} \wedge x'_{c\leqslant d}\right) & \left[\neg c_0\left(X'_{\mathbb{P}}\right)\right]
\end{aligned}
\tag{2.22}
$$

按照上述算法流程，最终可以求得，最后的谓词集合是：$\{(c=0), (d=0), (d \leqslant 3), (c \leqslant d), (d \leqslant 2), (d \leqslant 1), (1 \leqslant c), (3 \leqslant c)\}$。最后发现的归纳不变式是：$(\neg(c=0) \vee (d \leqslant 2)) \wedge ((d \leqslant 1) \vee (1 \leqslant c)) \wedge ((c=0) \vee \neg(d \leqslant 1)) \wedge (\neg(c=0) \vee (c \leqslant d)) \wedge ((d \leqslant 2) \vee (1 \leqslant c)) \wedge ((d \leqslant 2) \vee (3 \leqslant c)) \wedge ((d \leqslant 3) \vee \neg(c \leqslant d))$。

综上，IC3PA 利用谓词抽象，使得 IC3 算法可以在谓词抽象空间上运行，同时借用隐式抽象的理念，规避了抽象转移关系的计算，并且与反例制导的抽象精化框架相结合，解决了伪反例的情况，最终成为了一个完备自洽的字级 IC3 算法。

## 2.5　本章小结

本章从硬件验证与模型检测的相关基础知识出发，阐述了模型检测与硬件验证之间逻辑上的强相关性以及二者共同发展进步的历史关系，介绍了硬件模型检测研究中输入输出对象的定义，并点明了，随着硬件设计技术和产业的进步，硬件

模型检测算法的理念正逐步从比特级向字级转变的发展趋势。

由于本次研究工作的核心是尝试提出一种新的字级 IC3 算法，因此在后续的几节，本章便按照硬件模型检测算法的历史发展脉络展开叙述。首先对 IC3 这个最具代表性的比特级模型检测算法进行了详细介绍，然后说明了比特级与字级模型检测算法之间的区别与关联，进而对一些字级模型检测算法中经常使用的抽象技术进行了较为系统的介绍，同时也借由抽象空间中的伪反例问题介绍了反例制导的抽象精化这一经典框架。在本章的最后一节，详细介绍了一个基于 IC3、隐式谓词抽象和反例制导的抽象精化的字级 IC3 算法——IC3PA。

# 第 3 章　基于隐式抽象和变量隐藏抽象的 IC3 算法

本章设计并实验验证了基于隐式抽象和变量隐藏抽象的 IC3 算法—IC3VA。本章主要的组织结构如下：第 3.1 节介绍了 IC3PA 算法在一些测试样例的验证过程中出现的极端情况，并对问题进行了分析和对比思考；第 3.2 节中设计了基于隐式抽象和变量隐藏抽象技术的 IC3 算法—IC3VA，并给出了形式化介绍和证明；第 3.3 节详细介绍了 IC3VA 算法的泛化和精化方法；第 3.4 节对 IC3VA 算法整体流程进行了说明，并给出了伪码描述；第 3.5 节设计了相关对比实验来验证 IC3VA 算法的有效性，并对实验结果进行了分析；第 3.6 节对本章工作进行了概括总结。

## 3.1　对于 IC3PA 算法的分析和思考

如 2.4 中所介绍，IC3PA 算法是一种利用隐式抽象、谓词抽象以及 CEGAR 的字级 IC3 算法。然而在使用 IC3PA 对各类样例进行验证测试时，IC3PA 算法会暴露一些问题，为了方便论述，论文将通过下面的一个例子来具体说明。

假设系统 $S = \langle \{c, d\}, c = 0 \wedge d = 0, c' = c + d \wedge d' = d + 1 \rangle$，其中 $c$ 和 $d$ 是宽度为 4 的位向量。待验证的属性是 $P \doteq (c \leqslant d) \Rightarrow d \leqslant 3 \vee 5 \leqslant d$，对于待验证属性的直白表述是在 $c \leqslant d$ 时，$d$ 要么大于等于 5，要么小于等于 3。初始时，从 $P$ 和 $I$ 中提取的谓词集合是 $\mathbb{P}_0 \doteq \{(c = 0), (d = 0), (d < 5), (d < c), (3 < d)\}$。则 IC3PA 算法的迭代验证过程如下：

- 第一轮迭代，找到抽象反例路径如图 3.1所示。其中 $\hat{s}_3$ 中存在具体状态 ($c = 3, d = 4$)，c 虽然小于等于 d，但是 d 处于 3 到 5 之间，违背了属性 $P$。由于具体状态 $(c = 0, d = 1)$ 和 $(c = 0, d = 3)$ 使谓词集合 $\mathbb{P}_0$ 中每个谓词同真同假，所以抽象到同一个抽象状态内，然而 $(c = 0, d = 3)$ 会经过一步迁移到达 $(c = 3, d = 4)$，违反了属性 $P$。但是 $(c = 0, d = 1)$ 和 $(c = 0, d = 3)$ 并无转移，所以该反例是伪反例，调用插值程序求得精度 $(d = 1)$。$(d = 1)$ 可将第 $\hat{s}_2$ 分开，从而消除这条伪反例。更新 $\mathbb{P}_1 = \mathbb{P}_0 \cup (d = 1)$。

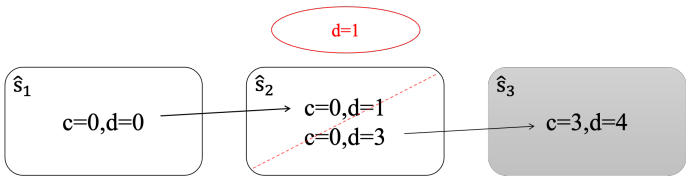



图 3.1　IC3PA 算法样例测试中第一轮迭代抽象反例路径示意图

- 第二轮迭代，找到抽象反例路径如图 3.2所示。其中 $\hat{s}_5$ 中存在具体状态 ($c = 0, d = 4$)，c 虽然小于等于 d，但是 d 处于 3 到 5 之间，违背了属性 $P$。($c = 0, d = 4$) 的前驱是 ($c = 13, d = 3$)，而具体状态 ($c = 4, d = 3$) 和 ($c = 13, d = 3$) 使谓词集合 $\mathbb{P}_1$ 中每个谓词同真同假，都抽象到了 $\hat{s}_4$，而 ($c = 4, d = 3$) 的前驱 ($c = 2, d = 2$) 和 ($c = 1, d = 2$) 同抽象到 $\hat{s}_3$。这两处都导致该抽象反例路径连续。调用插值程序找到精度 ($c = 1$)，将 ($c = 2, d = 2$) 和 ($c = 1, d = 2$) 分开，消除伪反例。更新 $\mathbb{P}_2 = \mathbb{P}_1 \cup (c = 1)$。

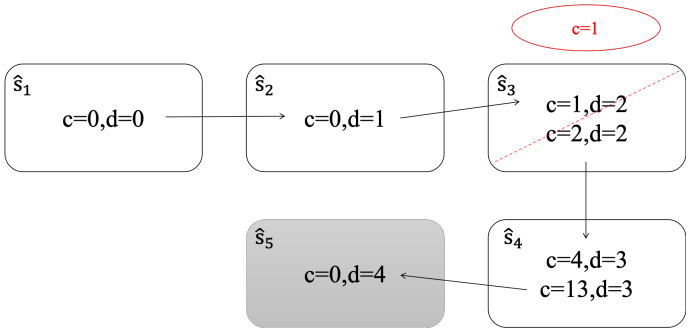



图 3.2　IC3PA 算法样例测试中第二轮迭代抽象反例路径示意图

- 第三轮迭代，找到抽象反例路径如图 3.3所示。其中 $\hat{s}_4$ 中存在具体状态 ($c = 4, d = 4$)，c 虽然小于等于 d，但是 d 处于 3 到 5 之间，违背了属性 $P$。($c = 4, d = 4$) 的前驱是 ($c = 1, d = 3$)，而 ($c = 1, d = 3$) 和 ($c = 1, d = 2$) 被抽象到同一个状态 $\hat{s}_3$，且 ($c = 1, d = 3$) 和 ($c = 1, d = 2$) 之间不存在转移，所以该反例是伪反例，调用插值程序求得精度 ($d = 2$)，可以将 ($c = 1, d = 3$) 和 ($c = 1, d = 2$) 分开，消除伪反例。更新 $\mathbb{P}_3 = \mathbb{P}_2 \cup (d = 2)$。

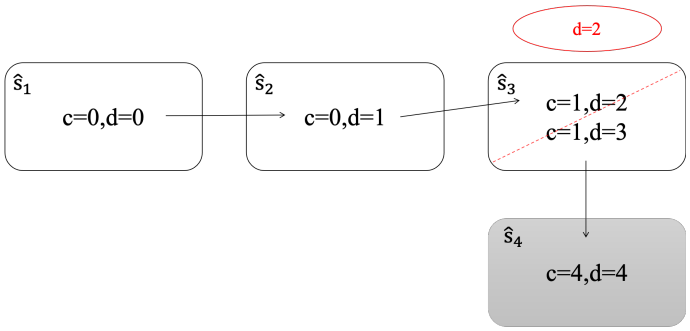



图 3.3　IC3PA 算法样例测试中第三轮迭代抽象反例路径示意图

- 继续按照上述类似的方式进行迭代，算法最终一共会精化 22 次，增加 27 个谓词，并得出属性为真的结论。最后的状态空间大小达到 $2^{32}$。

客观分析，即便把变量 $c$ 和 $d$ 一开始就看作是可见变量，状态空间大小也只是 $2^8$，并且由于所有变量都是可见变量，相当于没有进行抽象，因此也不会出现

伪反例。而在上述样例的测试流程中，IC3PA 算法前四次迭代分别在 $2^5$，$2^6$，$2^7$，$2^8$ 大小的状态空间上进行，最终的状态空间规模更是高达 $2^{32}$，远超 $2^8$。

以上述极端情况的测试样例为代表，可以看出，在一些属性与变量取值强相关的情况下，IC3PA 算法可能会求解出若干形如 $(a=0),(a=1),(a=2),...$ 的谓词，极端情况下，若变量 $a$ 的位宽为 n，IC3PA 算法会进行 $2^n$ 次迭代，每次排除 $a$ 的一种可能的取值，抽象空间中会出现 $2^n$ 个谓词，降低了算法的验证效率。如果直接将变量 $a$ 看作是可见变量，此时需要追踪的比特宽度为 $n$，即 $n$ 个谓词就可以追踪变量的值。综合来说，当发现关于某个变量的谓词个数大于一个阈值时，可以将此变量变为可见变量，这样可以缩小抽象状态空间，减少计算消耗，而对于对属性影响不大或无关的变量，可以直接将其隐藏掉，从而进一步缩减抽象空间。

将具体状态经过变量隐藏抽象后，抽象状态集合是可见变量集合，和 IC3 算法结合时也需要考虑两个问题：

(1) 由于抽象状态集合是可见变量集合，所以抽象的 $X$ 和 $X'$ 之间仍然受限于 SMT 的理论约束，如何求某个状态的前象。

(2) 抽象模型中只保留可见变量的转移函数，隐藏变量被抽象为无限制的基础输入，其转移函数被忽略，相当于被抽象为 $T_i = true$。但是，由于 $T_{\mathcal{V}}(X, X'_v)$ 可能会依赖于 $X_{inv}$ 中某些隐藏的当前状态的变量，而此时这些变量被当做自由输入，在模型检测中，自由输入在计算时通常会被存在量化，所以在求解前驱时，前像计算的理论相关问题依然存在。

## 3.2　基于变量隐藏抽象的 IC3 算法—IC3VA

对于第一个问题，本文尝试将抽象变量集合看作是可见变量的每一位，此时抽象后的状态变量就是布尔型变量，基于这个抽象变量集合求某个状态的前象效率更高；对于第二个问题，本文尝试利用隐式抽象编码来避免抽象转移函数的构造，从而规避量词消去问题。结合这两种改进方式，本文设计了一个基于变量隐藏抽象的 IC3 算法，称之为 IC3VA。下面将对本文设计的 IC3VA 作出形式化介绍，并给出形式化证明。

若 $X_v = \{x_1, \cdots, x_n\} \subseteq X$ 为可见变量集合，初始时，$X_v$ 包含属性 $P$ 中出现的所有状态变量。假设每个 $x_i$ 的宽度是 $n_i$ 个，那么一个变量 $x_i$ 可以拆解为元组 $< x_{i1}, x_{i2}, ..., x_{in_i} >$。所以拆解后的比特集为：

$$B_{X_v} = \bigcup_{i=1}^{n} \{b_{i1}, b_{i2}, ..., b_{in_i}\} \tag{3.1}$$

由于是 1bit，所以可以看作是布尔变量。抽象状态变量集合则为 $B_{X_v}$，此时可见变

量到抽象变量的映射关系是：

$$H_{X_v}\left(X_v, B_{X_v}\right) \doteq \bigwedge_{i=1}^{n}\bigwedge_{j=1}^{n_i} slice(x_i, j, j) \leftrightarrow b_{ij} \tag{3.2}$$

函数 $slice$ 表示对变量进行切片，选中的是变量第 $j$ 位的取值，即集合 $B_{X_v}$ 中的每一个布尔变量对应到原变量的位置上的取值，若取值为 0，则布尔变量为 $false$，反之为 $true$。所以，对于一个具体状态 $s$，其抽象状态 $\hat{s}$ 是 $s$ 在 $B_{X_v}$ 和 $B'_{X_v}$ 上的投影 $proj(s, B_{X_v} \cup B'_{X_v})$。即对于 IC3VA，抽象状态变量是可见变量集合的每一位比特。

这种方案的另一个好处是使得泛化更灵活，例如：假设可见变量是两个三比特位宽的变量，值分别为 010，110。如果直接采用标准的 IC3 泛化方式，即利用 unsat core 进行泛化，只有 $2^2$ 种泛化结果，分别是：$010110, 010\_\_\_, \_\_\_110$ 和 $\_\_\_\_\_$。但是把可见变量每一位看作是抽象变量时，可以泛化出 $2^6$ 个结果，如：$0\_\_110, \_1\_110, \_\_0110$ 和 $\_\_\_110$ 等。

根据 2.3.1 中提到的隐式抽象的定义，变量隐藏抽象的抽象条件是，若两个具体变量在可见变量集合上的取值相同，则应该被抽象到同一抽象状态，所以对于变量隐藏抽象有：

$$EQ_{X_v}(X, \overline{X}) \doteq \bigwedge_{x \in X_v} x(X) \leftrightarrow x(\overline{X}) \tag{3.3}$$

图 3.4较为形象地解释了隐式变量隐藏抽象的路径。$X$ 有三个变量，宽度均为 3bit，前两个是可见变量，后一个是不可见变量。$X_v$ 和 $X'_v$ 由 $H_{X_v}\left(X_v, B_{X_v}\right)$ 和 $H_{X_v}\left(X'_v, B'_{X_v}\right)$ 约束对应到 $B_{X_v}$ 和 $B'_{X_v}$，而 $X$ 和 $\overline{X}$，$X'$ 和 $\overline{X}'$ 通过 $EQ_{X_v}(X, \overline{X})$ 和 $EQ_{X_v}\left(\overline{X}', X'\right)$ 约束抽象到同一个抽象状态。

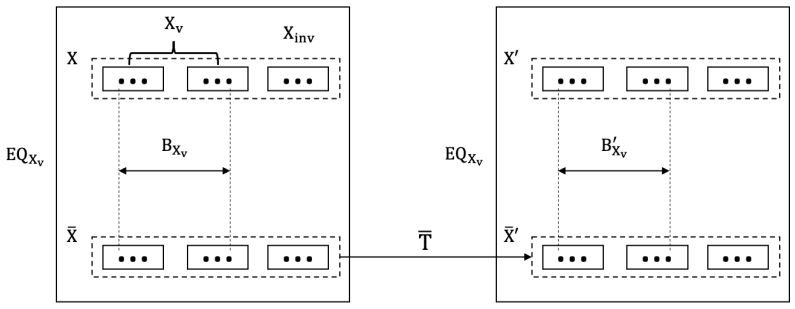



图 3.4　隐式变量隐藏抽象转移关系示意图

综上，此时 IC3 算法中的归纳相关检查公式（2.6）被编码为：

$$\begin{aligned} AbsRelInd(F, T, c, X_v) \doteq & F\left(B_{X_v}\right) \wedge c\left(B_{X_v}\right) \wedge H_{X_v}\left(X_v, B_{X_v}\right) \wedge H_{X_v}\left(X'_v, B'_{X_v}\right) \wedge \\ & EQ_{X_v}(X, \overline{X}) \wedge T\left(\overline{X}, \overline{X}'\right) \wedge EQ_{X_v}\left(\overline{X}', X'\right) \wedge \neg c\left(B'_{X_v}\right) \end{aligned} \tag{3.4}$$

如果将经过变量隐藏抽象后的抽象空间上的归纳相关公式记做 $RelInd(\hat{F}_{X_v},\hat{T}_{X_v},\hat{c}_{X_v})$，可以证明，此时 $RelInd(\hat{F}_{X_v},\hat{T}_{X_v},\hat{c}_{X_v})$ 和公式（3.4）定义的 $AbsRelInd(F,T,c,X_v)$ 等价，即 $RelInd(\hat{F}_{X_v},\hat{T}_{X_v},\hat{c}_{X_v})$ 可满足当且仅当 $AbsRelInd(F,T,c,X_v)$ 可满足，即，若存在解释 $\mu \vDash AbsRelInd(F,T,c,X_v)$，$\mu$ 中包含了 $X$，$X'$，$\overline{X}$，$\overline{X}'$，$B_{X_v}$ 和 $B'_{X_v}$ 的一组解释，将 $\mu$ 在 $B_{X_v} \cup B'_{X_v}$ 上的投影记做 $\mu_{B_{X_v}}$，则 $\mu_{B_{X_v}}$ 表示解释对应的抽象状态，且 $\mu_{B_{X_v}} \vDash RelInd(\hat{F}_{X_v},\hat{T}_{X_v},\hat{c}_{X_v})$，则此时可以通过剔除 $\mu_{B_{X_v}}$ 中出现的 $B'_{X_v}$ 来得到前驱。反之，若 $\mu \vDash AbsRelInd(F,T,c,X_v)$ 不可满足，那么 $\neg c\left(B_{X_v}\right)$ 可以加到 $\hat{F}$ 中，达到强化 $\hat{F}$ 的目的。

下面给出相关形式化证明:

1. 若 $\mu \vDash AbsRelInd(F,T,c,X_v)$，则 $\mu_{B_{X_v}} \vDash RelInd(\hat{F}_{X_v},\hat{T}_{X_v},\hat{c}_{X_v})$。

(1) 先证明：$\mu_{B_{X_v}} \vDash \hat{T}$。把 $\mu$ 在 $X \cup X'$ 和 $\overline{X} \cup \overline{X}'$ 上的投影分别记做：$t$ 和 $\bar{t}$。因为 $\bar{t} \vDash T$，所以它们所属的抽象状态之间也满足抽象转移函数，即 $\hat{\bar{t}} \vDash \hat{T}$。因为 $u \vDash EQ_{X_v}(X,\overline{X}) \wedge EQ_{X_v}\left(\overline{X}',X'\right)$，所以 $\hat{t}$ 和 $\hat{\bar{t}}$ 是同一个抽象转移关系，所以 $\hat{t} \vDash \hat{T}$。又因为 $\mu \vDash H_{X_v}\left(X_v,B_{X_v}\right) \wedge H_{X_v}\left(X'_v,B'_{X_v}\right)$，满足该抽象约束，所以 $\mu_{B_{X_v}}$ 就是 $\hat{t}$，所以 $\mu_{B_{X_v}} \vDash \hat{T}$。

(2) 再证明：$\mu_{B_{X_v}} \vDash \hat{F} \wedge \hat{c} \wedge \neg\hat{c}'$。因为：$t \vDash F \wedge c \wedge \neg c'$，所以 $\hat{t} \vDash \hat{F} \wedge \hat{c} \wedge \neg\hat{c}'$。

由 (1)(2) 可得证 1。

2. 若 $\mu_{B_{X_v}} \vDash RelInd(\hat{F}_{X_v},\hat{T}_{X_v},\hat{c}_{X_v})$，则 $AbsRelInd(F,T,c,X_v)$ 可满足。

证明：若 $\mu_{B_{X_v}} \vDash RelInd(\hat{F}_{X_v},\hat{T}_{X_v},\hat{c}_{X_v})$，则存在一组 $X \cup X'$ 解释 $t$，$t$ 的抽象等于 $\mu_{B_{X_v}}$ 且 $t \vDash T$。所以 $\mu_{B_{X_v}} \cup t \vDash F\left(B_{X_v}\right) \wedge c\left(B_{X_v}\right) \wedge H_{X_v}\left(X_v,B_{X_v}\right) \wedge H_{X_v}\left(X'_v,B'_{X_v}\right) \wedge EQ_{X_v}(X,\overline{X}) \wedge T\left(\overline{X},\overline{X}'\right) \wedge EQ_{X_v}\left(\overline{X}',X'\right) \wedge \neg c\left(B'_{X_v}\right)$，得证 2。

经由上述 1 和 2 成立，则 $RelInd(\hat{F}_{X_v},\hat{T}_{X_v},\hat{c}_{X_v}) \leftrightarrow AbsRelInd(F,T,c,X_v)$ 成立。

所以，对于隐式变量隐藏抽象，可以用 $AbsRelInd(F,T,c,X_v)$ 这个无量词的公式去替代 $RelInd(\hat{F}_{X_v},\hat{T}_{X_v},\hat{c}_{X_v})$，成功规避了抽象转移关系中的量词。

## 3.3　IC3VA 精化和泛化

前面提到，IC3 算法中的泛化算法对算法的效率有着非常大的影响，另外，由于使用了变量隐藏抽象，所以会有伪反例的问题，需要精化以消除伪反例。下面介绍 IC3VA 的泛化和精化算法。

算法 3.1　IC3 算法泛化流程

```
1  void MIC(q: cube ref, i: level):
2      for each literal l in q:
3          q̂ := q\l:
4          if down(q̂,i):
5              q = q̂
6  bool down(q: cube ref, i: level):
7      while true :
8          if I ⇏ ¬q:
9              return false
10         if F_i ∧ ¬q ∧ T ⇒ ¬q′:
11             return true
12         with (F_i ∧ ¬q)-state s:
13             q := q ⊔ s
```

### 3.3.1　IC3VA 的泛化算法

如 2.2 中介绍，IC3 算法流程中，帧 $F_i$ 一般用若干子句的集合来表示系统迁移 $i$ 步可以到达的状态集合的上近似，当某个立方体方 $(c,i)$ 被阻塞成功，说明前 $i$ 步内该立方体表示的状态集合不可达，所以将该立方体取反变成子句加入到帧 $F_i$ 中，达到去掉该立方体的目的。若某个立方体已经被成功阻塞，需要将其泛化成文字更少的立方体，立方体取反则可以去除更多不可达状态。IC3VA 的抽象状态变量是可见变量的每一位比特，因此在抽象泛化部分，IC3VA 可以使用标准的比特级 IC3 算法的泛化算法。

标准的 IC3 算法流程如算法3.1所示，由于立方体可以看作是若干文字的集合，所以主程序 MIC 遍历集合中的文字，对每个文字尝试丢掉它来加强子句，主要是对丢弃后得到的立方体进行算法 down 检查。如果 down 返回 true，则可以丢弃，如果返回 false，则说明不能丢弃，MIC 将其加入到原子句中。down 程序的作用是寻找 $\neg q$ 的最大归纳子句，如果能找到，返回 true，否则返回 false。具体说来，第 k 步迭代时，首先检查有没有包含初始状态，如果有则返回 false，如果没有，继续检查 $q_k$ 是不是归纳相关于 $F_i$，如果是，则返回 true，否则，提取 $q_k$ 的前驱 $s$，$q_{k+1}$ 由 $q_k$ 和 $s$ 的公共的文字组成。所以每次迭代，$q$ 中文字的个数都在逐渐减少，有效的扩展了 $q$ 中的某个状态的转移。

但是，该算法也有缺点，图3.5就是一个该算法泛化不够好的例子。假设 000 是初始状态，001 是违反属性的状态，它有两个前驱：110 和 100。假设 IC3 算法首先找到了 100，由于 100 没有前驱，所以它的反是归纳的，所以 IC3 判定不可达。IC3 使用算法3.1进行泛化，假设丢弃第三个文字，变为 10x。x 表示不在乎该文字，

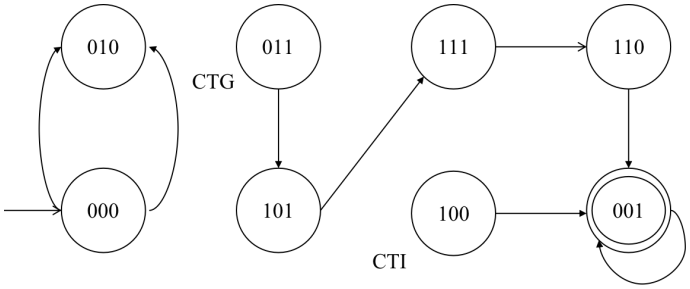



图 3.5　传统 IC3 算法泛化失败的样例

由于 101 有前驱 011，所以 10x 不是归纳的。为了能找到一个可以把 100 和 101 都包含的且不可达的立方体，该立方体必须包含 011，否则它的反就不是归纳的。同时包含 100、101 和 011 的立方体是 xxx，就把初始状态包括进去了。所以 down 会返回 false。类似的原因，第一个和第二个文字不能被丢弃。所以 IC3 算法只能阻塞 100 一个。

为了解决上述问题，论文使用了一种基于泛化的反例（Counterexample to Generalization, CTG）的泛化方法来替代传统的 IC3 泛化算法，算法流程如算法3.2所示，其中 CTG 指类似图3.5中 011 那种导致泛化失败的前驱。

此算法也遍历立方体中的每个文字，并检查剔除后的立方体是不是归纳的（11 到 14 行），如果不是归纳的，不会立即把前驱 s 和 q 连接起来，而是试图在 $F_{i-1}$ 层去阻塞 s（第 18 行），如果尝试成功，就增加 CTG 的个数，并尝试在更高层去 block(第 20-22 行)，找到最高层后，就在最高层递归调用算法去泛化 s, 并增加递归深度 d（第 23 行）。如果一个 CTG 的深度很大，就不进行后续的步骤（第 16 行）。在解决了一个导致 q 不归纳的原因后，算法继续迭代。如果 CTG 的个数太多，超过了限制，就把找到的 CTG 和 q 连接起来，并把 CTG 数目置 0（第 26-27 行）。这种泛化方式比经典的泛化方式泛化的效果更好。

### 3.3.2　IC3VA 的精化算法

根据 2.3.6 中提到的反例制导的抽象精化框架（CEGAR），如果在抽象状态空间中发现反例，首先需要在具体系统中模拟该抽象反例路径的执行，如果该检查确认反例在抽象状态空间可达，则验证为真反例，证明属性违背，否则，证明为伪反例，需要据此伪反例提取精度，继续精化。而对于 IC3VA 算法而言，则需要增加新的可见变量来精化抽象空间。接下来介绍 IC3VA 的精化算法。

假设求得的反例路径是 $\pi = (\hat{s_0}, ..., \hat{s_k})$，$\hat{s}$ 表示一个抽象状态。因为使用了抽象技术，所以这条反例路径实际包含了具体空间上的多条路径。利用限界模型检测将该路径进行展开，具体流程如下：

(1) 首先，将每个抽象状态 $\hat{s}$ 具体化为具体状态，即将抽象状态变量替换成具体

算法 3.2　基于泛化反例的 IC3 算法泛化算法

```
1  void MIC(q: cube ref, i: level):
2      MIC(q, i, 1)
3  void MIC(q: cube ref, i: level, d: recDepth):
4      for each literal l in q:
5          q̂ := q\l:
6          if ctgDown(q̂, i, d):
7              q = q̂
8  bool ctgDown(q: cube ref, i: level, d: recDepth):
9      ctgs := 0
10     while true :
11         if I ⇏ ¬q:
12             return false
13         if F_i ∧ ¬q ∧ T ⇒ ¬q′:
14             return true
15         with (F_i ∧ ¬q)-state s:
16             if d > maxDepth:
17                 return false
18             if ctgs < maxCTGs and i > 0 and I ⇒ ¬s and F_{i−1} ∧ ¬s ∧ T ⇒ ¬s′:
19                 ctgs := ctgs + 1
20                 for j := i to k do:
21                     if F_j ∧ ¬s ∧ T ⇏ ¬s′
22                         break
23                     MIC(s, j − 1, d + 1)
24                     clauses(F_j) := clauses(F_j) ∪ ¬s
25             else:
26                 ctgs := 0
27                 q := q ⊔ s
```

的语义：$s = \hat{s}(B_{X_v}) \wedge H_{X_v}(X_v, B_{X_v}) = \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{n_i} slice(x_i, j, j) \leftrightarrow \hat{s}(b_{ij})$。

(2) 然后，给 $s$ 对应的公式加上“时间”属性，即是迁移中的第几步。

(3) 最后，给具体状态之间加上转移函数，使其连接起来形成具体路径 $Path^{\pi}_{B_{X_v}}$。若 $Path^{\pi}_{B_{X_v}}$ 可满足，说明该抽象反例存在对应的具体反例，若不可满足，说明该抽象反例为伪反例，系统抽象层次太高，需要增加可见变量来消除该反例。

本文设计的求解精度的方式是对 $Path^{\pi}_{B_{X_v}}$ 求插值，由于 $Path^{\pi}_{B_{X_v}}$ 不可满足，所以可以调用求解器求该路径的插值，长度为 $k$ 的路径产生 $k-1$ 个插值，若某插值公式中只有一个隐藏变量，则将该隐藏变量变为可见，消除反例。图3.6展示了 IC3VA 的精化方案。

由于可见变量集合是单调增加的，抽象转移函数也是单调增强，即 $X_v \subseteq X'_v$，所以 $\hat{T}_{X'_v} \rightarrow \hat{T}_{X_v}$。所以精化之前帧中的子句仍然可以保留，算法是增量的。

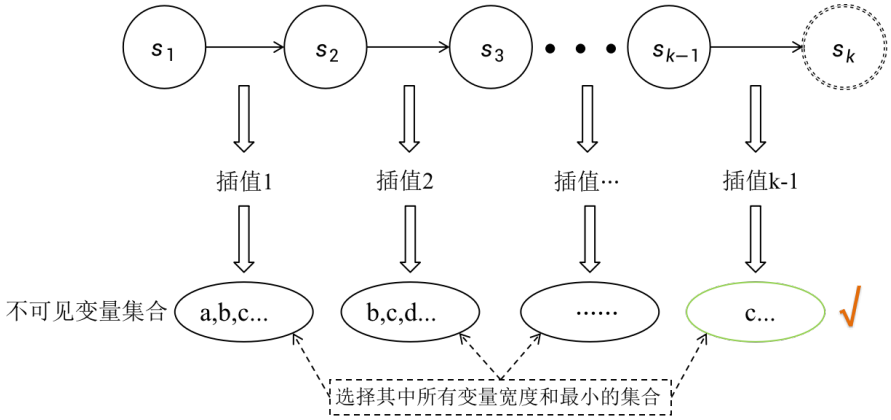



图 3.6　IC3VA 精化示意图

## 3.4　IC3VA 算法流程

根据 3.2 中 IC3VA 的形式化介绍和证明，以及 3.3 中 IC3VA 的泛化和精化算法的介绍，可以给出整个 IC3VA 算法的流程说明，大致伪码描述如算法 3.3所示。

IC3VA 和 IC3 一样，维护了帧的序列 $F$。初始时的精度集合 $X_v$ 为属性 $P$ 中出现的变量。IC3VA 中的状态变量是 $B_{X_v}$，IC3VA 中的立方体是在 $B_{X_v}$ 中变量与或非的合取，子句则是析取。IC3VA 的帧是这些子句的合取。算法是一个大循环，每次迭代首先是阻塞环节，然后是传播环节。

阻塞环节，首先检查上一个帧中违背属性的抽象的布尔立方体——$c(B_{X_v})$(第 8 行)，然后检查公式 $AbsRelInd(F_{i-1}, T, \neg c, X_v)$ 是否可以满足（第 10 行）。如果归纳检查公式可满足，对 $\neg c$ 进行泛化（第 30 行，也就是算法 RECBLOCK 第 5 行），并将泛化后的子句加到 $F_1$ 到 $F_i$ 中，来强化它们。如果这个检查失败了，那么可以从 $AbsRelInd(F_{i-1}, T, \neg c, X_v)$ 的模型中提取出 c 的抽象前驱，也就是 $F_{i-1} \wedge \neg c$ 中可以一步到达 $c$ 的立方体 s，然后递归的 block s。递归函数 RECBLOCK 要么不断加强队列中的帧，要么不断泛化抽象反例。如果发现一个处于第 0 级的需要被阻塞的状态，该递归函数退出（RECBLOCK 第 1 行），说明发现了一个抽象反例。然后尝试对抽象路径进行具体化，如果不能具体化，说明是伪反例，从该抽象路径中提出精度加入到原精度集合 $X_v$ 中出现的变量（第 14 行）。具体化成功，则属性为假，返回 false。

如果当前帧找不到违反属性的抽象立方体了，则算法进入传播阶段。新增一个不包含任何子句的帧（第 1 行）, 并从 $F_1$ 开始检查，将 $F_i$ 中归纳相关于它的子句加入到 $F_{i+1}$。因为 $AbsRelInd(F_i, T, \neg c, X_v) \vDash \bot$，所以 $F_i$ 足以说明 $c$ 在 $F_{i+1}$ 的不可达性。如果在传播阶段中，发现 $F_i = F_{i+1}$（第 23 行），则说明属性为真。

算法 3.3　IC3VA 算法

**Function** IC3VA($I,T,P,X_v$)**:**
  // 初始化可见变量集合为属性 I 和 P 中出现的变量集合
  $X_v = X_v \bigcup \{x | x \in I \vee x \in P\}$
  **if not** $\overline{I} \vDash \overline{P}$: **return** FALSE // 初始状态即不满足
  $F_0 := \overline{I}$ // 路径的第一个元素即为初始公式
  $k := 1, F_k := \top$ // 向路径添加一个新的帧（frame）
  **while** TRUE:
    // 阻塞阶段
    **while** $F_k \wedge \neg\overline{P}$ is sat :
      从模型 $F_k \wedge \neg\overline{P}$ 中抽取一个 cube $c$
      **if not** RECBLOCK($c$,$k$):
        向前追溯找到深度为 0 的抽象初始状态 $\hat{s_0}$
        形成抽象反例 $\hat{\pi} = \hat{s_0}, \hat{s_1}, ...\hat{s_k}$
        **if not** CONCRETIZABLE($I$,$T$,$P$,$X_v$,$\hat{\pi}$):
          $X_v = X_v \bigcup$ REFINE($I$,$T$,$P$,$X_v$,$\hat{\pi}$)
        **else return** FALSE
    // 传播阶段
    $k := k + 1, F_k := \top$
    **for** $i := 1$ **to** $k - 1$:
      **for each** clause $c \in F_i$:
        **if** AbsRelInd($F, T, c, X_v$) is unsat :
          add $c$ to $F_{i+1}$
      **if** $F_i = F_{i+1}$:
        **return** TRUE // 得到归纳不变式 $F_i$，属性被证明

**Function** RECBLOCK($s$,$i$)**:**
  **if** $i = 0$: **return** FALSE // 到达初始状态
  **while** AbsRelInd($F_{i-1}, T, \neg s, X_v$) is sat :
    从模型 AbsRelInd($F_{i-1}, T, \neg s, X_v$) 中抽取一个前驱 cube $c(B_{X_v})$
    **if not** RECBLOCK($c$,$i - 1$): **return** FALSE
  // GENERALIZE 是标准 IC3 泛化
  $g$ = **GENERALIZE**($\neg s$, $i$)
  **for** $j := 1$ **to** $i$: 将 $g$ 加到 $F_j$
  **return** TRUE

## 3.5　实验与分析

为了验证本章提出的基于隐式变量隐藏抽象的 IC3 算法——IC3VA 的有效性，论文按照算法3.3中的流程设计，在基于谓词抽象的 IC3 算法——IC3PA[37] 的开源代码的基础上，做了对应的修改，实现了 IC3VA 算法。同时论文也选取了相关的硬件验证和模型检测测试数据集，并在这些数据集上设计了相关对比实验，测试

并分析了 IC3VA 与 IC3PA 的验证效果。下面将详细介绍实验的设计、结果和分析。

### 3.5.1　实验环境与配置

为了便于公平比较各个算法的验证效果，论文中所有的实验都在同一台机器上完成，机器的配置如表3.1所示。

表 3.1　实验机器的配置信息

| 配置名称 | 配置信息 |
|---|---|
| 操作系统 | 64 位 Ubuntu 20.04.2 LTS |
| CPU | AMD EPYC 7282 16-Core Processor @2.80GHz |
| 内存 | 128GB |
| 硬盘 | 4T 机械硬盘 7200RPM |

为了更好的评估算法本身的验证效果，在实验过程中，所有算法在验证每个测试样例时，将只会被分配一个逻辑运算核用于计算，避免软件工具并行设计或 CPU 的并行计算等外部因素对实验结果带来干扰。同时考虑到不同验证测试样例的复杂度和难度不同，部分验证任务可能会有超时的情况，实验统一将单个测试样例的验证时间上限设为 15 分钟，即 900 秒。

### 3.5.2　实验数据说明

为了更加广泛地测试 IC3VA 算法的普适性和有效性，论文从多个来源收集了总共 487 个硬件安全属性验证的任务。

参考论文[38]中的 opensource 任务集的设置，本次实验选取了 vcegar[39]、v2c[40]、verilog2smv[41] 等开源的标准硬件验证测试集的 163 个验证任务，此处记做任务集 A。任务集 A 中初始的硬件设计描述语言均为 verilog 文件，为了便于算法进行验证分析，先通过 yosys 工具[42]将这些任务转成 btor2 文件，再用工具 vmt-tools[43]转成本次实验所采用的统一验证输入格式——vmt 格式。

除此之外，本次实验还选取了 2020 年的硬件模型检测大赛（HWMCC'20）[44]的 bit-vector 分支上的验证任务（共 324 个），此处记做任务集 B。任务集 B 中的 324 个任务原始格式是 btor2 格式，所有这些任务也转成 vmt 格式。

### 3.5.3　实验与分析

开始进行实验分析之前，先设计实验目标。对于此次实验，需要进行重点验证和分析的点主要有以下两个：

1. IC3VA 算法的有效性。即 IC3VA 是否可以解决 IC3PA 不能解决的任务？

2. IC3VA 算法的验证效率。即在 IC3VA 和 IC3PA 都能解决的任务上，两者的用时对比如何？

按照 3.5.1 中统一的实验配置，IC3PA 和 IC3VA 算法在任务集 A 和 B 总计 487 个任务上进行了验证实验，实验总体结果如表3.2所示，其中 safe 表示验证为安全的任务数量，unsafe 表示验证为不安全的任务数量，to 表示验证耗时大于 900 秒被强行终止的任务数量，unk 表示算法无法得出验证结论的任务数量，unique 表示只有当前算法验证出结果而其他验证算法没有得到验证结果的任务数量。

依据实验结果，IC3VA 和 IC3PA 共同可解的任务，在任务集 A 上有 144 个，在任务集 B 上有 66 个，总计 210 个。虽然在 900 秒内可以解决的任务个数上，IC3PA 要多于 IC3VA，然而，对于 IC3VA 算法有效性，从 unique 这项指标可以看到，IC3VA 确实可以解决部分 IC3PA 不能解决的任务，其中任务集 A 中有 3 个，任务集 B 中有 12 个。同时考虑任务集 B 的任务平均验证复杂度要远高于任务集 A，可以看出 IC3VA 的改进策略是有效的，特别是当验证任务本身较为复杂时，体现的会更明显。对于算法验证效率问题，接下来将从时间、精化次数，以及求解次数等角度分析实验结果。

表 3.2　IC3VA 和 IC3PA 在两任务集上的求解情况

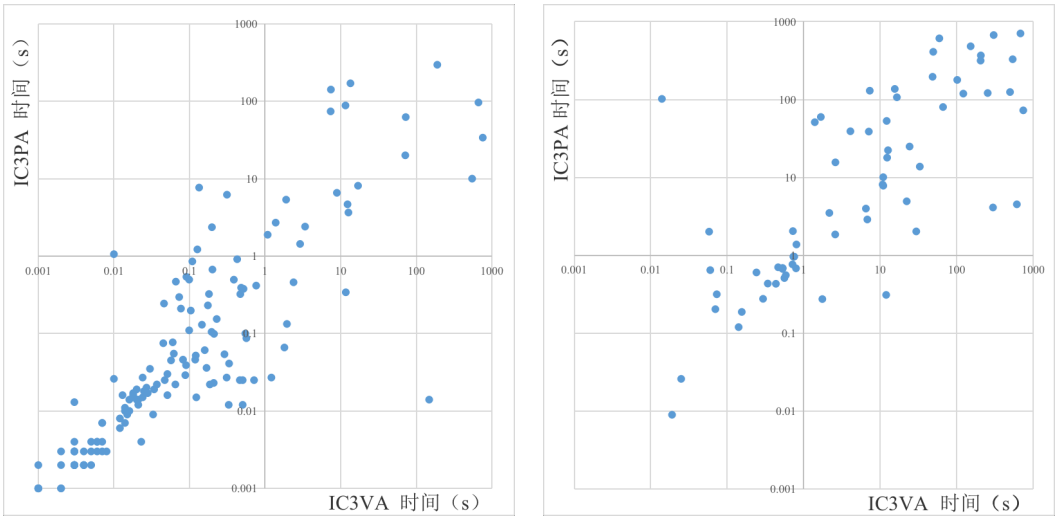
| 任务集 | 算法 | safe | unsafe | to | unk | unique |
|---|---|---|---|---|---|---|
| 任务集 A （共 163 个） | IC3PA | 116 | 40 | 7 | 0 | 12 |
| | IC3VA | 106 | 41 | 12 | 4 | 3 |
| 任务集 B （共 324 个） | IC3PA | 74 | 15 | 230 | 5 | 24 |
| | IC3VA | 66 | 12 | 171 | 75 | 12 |

为了更直观反映两种算法在验证耗时上的对比，对于任务集 A 和 B 上两者共同解决的 144 和 66 个任务，表3.3给出了 IC3VA 和 IC3PA 在共同解决任务上总的时间、精化次数和求解次数的对比，图3.7给出了 IC3VA 和 IC3PA 在 A 和 B 两个任务集上共同解决任务上的耗时对比散点图。

表 3.3　IC3VA 和 IC3PA 在共同解决的任务上的统计对比

| 任务集 | 算法 | 时间/秒 | 精化次数 | 求解次数 |
|---|---|---|---|---|
| 任务集 A （共同解决 144 个） | IC3PA | 1059 | 398 | 869136 |
| | IC3VA | 2584 | 31 | 2499712 |
| 任务集 B （共同解决 66 个） | IC3PA | 5671 | 189 | 1615119 |
| | IC3VA | 5211 | 6 | 1497966 |

图中的横轴都表示 IC3VA 在各个验证任务上的耗时，纵轴表示 IC3PA 的验证




(a) 任务集 A　　(b) 任务集 B

图 3.7　IC3VA 和 IC3PA 在共同解决的任务上的耗时对比散点图

耗时，因此对角线附近的点则表示两种算法在该点对应的验证任务上耗时接近，而位于对角线下方的点表示 IC3PA 在该点对应的验证任务上耗时更短，对角线上方的点则表示 IC3VA 在该点对应的验证任务上耗时更短。考虑测试集中不同任务验证复杂度差异较大，求解时间从 0 到 900 秒跨度较大，为了更好的观感，图中两个轴设为对数坐标。

从表3.3可以看出，在任务集 A 上，对于 IC3VA 和 IC3PA 都可以解决的 144 个任务，IC3VA 的总体耗时要明显大于 IC3PA，是后者的 2.4 倍，同时求解次数也是后者的 2.82 倍；而在任务集 B 上，对于两者都可以解决的 66 个任务，IC3VA 的总体耗时却小于 IC3PA，求解次数也要小于 IC3PA。在不同的任务集上，两者的验证效果对比有所不同，对于共同可解的任务，两者在验证时间和求解次数上表现对比差别较大，值得注意的是，IC3PA 在两个任务集上，平均每个验证任务的精化次数基本一致，都为 2.8 次左右。对于这种对比上的差异表现，通过理论思考以及对一些实际任务的验证过程的具体观察，可以分析出可能的原因。考虑到虽然任务集 A 中的任务相对比较简单，但 IC3PA 验证过程中出现的变量的宽度并不一定很小，因而对于其中大部分任务，更加灵活的 IC3PA 使用较少的谓词就可以解决，从数据上看，144 个任务只精化了 398 次，平均 2.76 次。在这种情况下，IC3VA 中使用的变量隐藏抽象就不具有效率上的优势了，因此整体耗时也会明显增多。但在更为复杂的任务集 B 上，对于其中大部分验证任务，其验证过程更为复杂，且复杂度主要体现在转移函数以及变量数量上，而不是变量位宽上，IC3PA 在两个任务集上的平均精化次数基本一致也侧面印证了这一推论。因此在任务集 B 上两者都可以解决的任务中，相比于 IC3PA 使用的谓词抽象方式，IC3VA 使用变量隐

藏抽象提高了精化部分的效率，总的精化次数仅为 6 次，所以在验证耗时和求解次数上就会更有优势。

图3.7中展示出的结果也符合上述分析。可以看到，图中（a）反映出任务集 A 中大部分的任务耗时都较短，第三象限的点更加密集，且第三象限的点位于对角线下方的更多（上方 66 个，上方 75 个）；图中（b）则反映出任务集 B 中大部分任务耗时较长，图中点多位于第一象限，并且位于对角线上方的点数多于下方（上方 36 个，下方 27 个）。因此，可以看出，IC3VA 的验证效果符合本章开头部分描述的设计初衷，在一些更为复杂的验证任务上，缓解了 IC3PA 验证过程中可能出现的关于某个变量的谓词个数过大的问题，提升了验证效率。

综合上述分析，结合实验目标，可以得出结论如下：

1. IC3VA 算法是有效的。在任务集 A 和 B 上，IC3VA 都能解决部分 IC3PA 无法解决的任务，分别是 3 和 12 个。这说明 IC3VA 算法是有效的。
2. 实验结果显示，对于任务集 B 上的两者共同可解的任务，IC3VA 的验证总耗时减少了 460s，验证效率得到了提升，散点图显示，在个别的例子上甚至是一个数量级的效率提升。事实上，在 IC3PA 超时而 IC3VA 可解的任务上，IC3VA 平均耗时不超过 200s。这说明了 IC3VA 的确实现了算法设计的初衷，在一些 IC3PA 关于某个变量的谓词个数过大的情况下，IC3VA 可以加速验证。
3. IC3VA 算法仍然是受限的。IC3VA 在 75 个 unknown 的任务上都有 1 次精化，但是只有 6 次精化成功，剩余 69 个都精化失败。另外，从整体实验结果来看，IC3VA 在解决任务的个数上是少于 IC3PA 的，这种表现的原因可能是，IC3VA 算法把转移系统的初始条件和属性中的所有变量都变为可见变量的方式，会导致初始抽象程度太低，对于某些任务来说，甚至等同于把所有有关的变量都变为了可见变量，因此有的任务本来只需一两个谓词就可以解决，而 IC3VA 会迅速增大状态空间导致超时。

## 3.6　本章小结

本章首先举出了一个 IC3PA 算法找到的谓词个数大于变量宽度的例子。经过分析发现，这种情况下直接追踪该变量的具体值效率更高。由此引出了基于隐式变量隐藏抽象的字级 IC3 算法——IC3VA，并介绍了 IC3VA 中新的泛化和精化方案，同时给出了 IC3VA 算法的伪码描述。

为了验证 IC3VA 算法的有效性，论文对 IC3VA 算法进行了工程实现，并设置了两个标准的硬件验证测试集，在其上进行了实验和结果分析。实验结果表明，

IC3VA 算法确实可以解决部分 IC3PA 算法超时的任务，并且在任务集 B 中的两者共同可解的任务上，其整体耗时更少。但是实验也暴露出了 IC3VA 算法的缺点，即从初始条件和属性中获取可见变量的方式比较极端，容易造成状态空间迅速增大导致超时，因此设计一个更灵活的精度初始化以及精化方案可以是后续工作的一个发展方向。

# 第 4 章　基于隐式混合抽象的 IC3 算法

本章设计并实现了基于隐式抽象和变量隐藏抽象的 IC3 算法和框架。本章的主要组织结构如下：第 4.1 节设计了基于隐式抽象和混合抽象的方式来综合这两种抽象，给出了形式化描述和证明，并给出了与之适配的泛化和精化方法；第 4.2 节中为了将混合抽象推广到更多的抽象域，设计了一种更加解耦的混合算法，并介绍了与之适配的新的泛化和精化方案；第 4.3 节介绍了基于隐式混合抽象的 IC3 算法验证工具框架，给出了工程对应的框架结构图以及各个模块的功能介绍；第 4.4 节设计了相关对比实验来验证基于隐式混合抽象的 IC3 算法的有效性，在 4.3 节中提出的验证工具框架上，完成了多个配置下的对比实验，并对实验结果进行了分析；第 4.5 节则是对本章的整体工作进行了概括总结。

## 4.1　混合 IC3PA 和 IC3VA 算法

第三章的实验结果显示，IC3VA 和 IC3PA 有各自的特点，IC3VA 相比 IC3PA 而言是一种更具体的抽象方式，而 IC3PA 则更加灵活和普适，所以在实验结果上，两者可以解决的任务类型不尽相同。又由于两者都借用了隐式抽象技术，因此，本节尝试通过混合这两种抽象的方式，将这两种风格不同的字级 IC3 算法进行组合，并给出相关的形式化描述和证明。

### 4.1.1　混合抽象空间

IC3PA 和 IC3VA 都利用了隐式抽象的理念来避免抽象转移关系的提前构建，都将抽象的定义编码到路径中。对于 IC3PA，公式（2.16）定义了谓词抽象的抽象方式：谓词集合 $\mathbb{P}$ 上赋值相同的具体状态应属于同一个抽象状态。对于 IC3VA，公式（3.3）定义了变量隐藏抽象的抽象方式：可见变量集合 $X_v$ 上赋值相同的具体状态应属于同一个抽象状态。无论是对谓词进行赋值，还是对变量进行赋值，本质都是对具体变量的约束，所以可以通过混合这些约束来实现混合抽象。

若 IC3PA 的抽象状态变量集合是 $X_{\mathbb{P}}$，IC3VA 的可见变量集合是 $X_v$，抽象状态变量集合是 $B_{X_v}$，则混合抽象的精度集合是 $\mathbb{P} \cup X_v$，抽象变量集合为 $X_{\mathbb{P}} \cup B_{X_v}$。具体状态 $s$ 在混合抽象上的抽象状态记做投影 $proj(s, X_{\mathbb{P}} \cup B_{X_v} \cup X'_{\mathbb{P}} \cup B'_{X_v})$。

混合抽象的具体变量到抽象状态变量的映射关系为：

$$H_{\mathbb{P}\cup X_v}(X, X_{\mathbb{P}} \cup B_{X_v}) \doteq H_{\mathbb{P}}\left(X, X_{\mathbb{P}}\right) \wedge H_{X_v}\left(X_v, B_{X_v}\right) \tag{4.1}$$

由于 $X_v \in X$，所以这里 $X_v \cup X$ 写做 $X$。

混合抽象的隐式约束为

$$EQ_{\mathbb{P}\cup X_v}(X,\overline{X}) \doteq EQ_{\mathbb{P}}(X,\overline{X}) \wedge EQ_{X_v}(X,\overline{X}) \tag{4.2}$$

IC3 算法中的归纳相关检查公式（2.6）被编码为：

$$\begin{aligned} AbsRelInd(F,T,c,\mathbb{P}\cup X_v) \doteq\ & F(X_{\mathbb{P}}\cup B_{X_v}) \wedge c(X_{\mathbb{P}}\cup B_{X_v}) \wedge \\ & H_{\mathbb{P}\cup X_v}(X, X_{\mathbb{P}}\cup B_{X_v}) \wedge H_{\mathbb{P}\cup X_v}(X', X'_{\mathbb{P}}\cup B'_{X_v}) \wedge \\ & EQ_{\mathbb{P}\cup X_v}(X,\overline{X}) \wedge T(\overline{X},\overline{X}') \wedge EQ_{\mathbb{P}\cup X_v}(\overline{X}', X') \wedge \neg c(X'_{\mathbb{P}}\cup B'_{X_v}) \end{aligned} \tag{4.3}$$

同样，如果将混合抽象后的抽象空间上的归纳相关公式记为 $RelInd(\widehat{F}_{\mathbb{P}\cup X_v},\widehat{T}_{\mathbb{P}\cup X_v},\hat{c}_{\mathbb{P}\cup X_v})$，则可以证明，此时 $RelInd(\widehat{F}_{\mathbb{P}\cup X_v},\widehat{T}_{\mathbb{P}\cup X_v},\hat{c}_{\mathbb{P}\cup X_v})$ 和公式（4.3）中定义的 $AbsRelInd(F,T,c,\mathbb{P}\cup X_v)$ 等价。即，$RelInd(\widehat{F}_{\mathbb{P}\cup X_v},\widehat{T}_{\mathbb{P}\cup X_v},\hat{c}_{\mathbb{P}\cup X_v})$ 可满足当且仅当 $AbsRelInd(F,T,c,\mathbb{P}\cup X_v)$ 可满足。即，若存在解释 $\mu \vDash AbsRelInd(F,T,c,\mathbb{P}\cup X_v)$，$\mu$ 中包含了 $X$，$X'$，$\overline{X}$，$\overline{X}'$，$B_{X_v}$ 和 $B'_{X_v}$，以及 $X_{\mathbb{P}}$ 和 $X'_{\mathbb{P}}$ 的解释，将 $\mu$ 在 $X_{\mathbb{P}}\cup B_{X_v}\cup X'_{\mathbb{P}}\cup B'_{X_v}$ 上的投影记做 $\mu_{X_{\mathbb{P}}\cup B_{X_v}}$，则 $\mu_{X_{\mathbb{P}}\cup B_{X_v}}$ 表示解释对应的抽象状态，且 $\mu_{X_{\mathbb{P}}\cup B_{X_v}} \vDash RelInd(\widehat{F}_{\mathbb{P}\cup X_v},\widehat{T}_{\mathbb{P}\cup X_v},\hat{c}_{\mathbb{P}\cup X_v})$。反之，若 $\mu \vDash AbsRelInd(F,T,c,\mathbb{P}\cup X_v)$ 不可满足，那么 $\neg c(X_{\mathbb{P}}\cup B_{X_v})$ 可以加到 $\widehat{F}$ 中，用以强化 $\widehat{F}$。

下面给出相关形式化证明：

**1. 若 $\mu \vDash AbsRelInd(F,T,c,\mathbb{P}\cup X_v)$，则 $\mu_{X_{\mathbb{P}}\cup B_{X_v}} \vDash RelInd(\widehat{F}_{\mathbb{P}\cup X_v},\widehat{T}_{\mathbb{P}\cup X_v},\hat{c}_{\mathbb{P}\cup X_v})$。**

(1) 先证明：$\mu_{X_{\mathbb{P}}\cup B_{X_v}} \vDash \widehat{T}$。把 $\mu$ 在 $X\cup X'$ 和 $\overline{X}\cup\overline{X}'$ 上的投影分别记做：$t$ 和 $\bar{t}$。因为 $\bar{t} \vDash T$，所以它们所属的抽象状态之间也满足抽象转移函数，即 $\hat{\bar{t}} \vDash \widehat{T}$。因为 $u \vDash EQ_{\mathbb{P}\cup X_v}(X,\overline{X}) \wedge T(\overline{X},\overline{X}') \wedge EQ_{\mathbb{P}\cup X_v}(\overline{X}', X')$，所以 $\hat{t}$ 和 $\hat{\bar{t}}$ 是同一个抽象转移关系，所以 $\hat{t} \vDash \widehat{T}$。又因为 $\mu \vDash H_{\mathbb{P}\cup X_v}(X, X_{\mathbb{P}}\cup B_{X_v}) \wedge H_{\mathbb{P}\cup X_v}(X', X'_{\mathbb{P}}\cup B'_{X_v})$，满足该抽象约束，所以 $\mu_{X_{\mathbb{P}}\cup B_{X_v}}$ 就是 $\hat{t}$，所以 $\mu_{X_{\mathbb{P}}\cup B_{X_v}} \vDash \widehat{T}$。

(2) 再证明：$\mu_{B_{X_v}} \vDash \widehat{F} \wedge \hat{c} \wedge \neg\hat{c}'$。因为：$t \vDash F \wedge c \wedge \neg c'$，所以 $\hat{t} \vDash \widehat{F} \wedge \hat{c} \wedge \neg\hat{c}'$。

由 (1)(2) 可得证 1。

**2. 若 $\mu_{X_{\mathbb{P}}\cup B_{X_v}} \vDash RelInd(\widehat{F}_{\mathbb{P}\cup X_v},\widehat{T}_{\mathbb{P}\cup X_v},\hat{c}_{\mathbb{P}\cup X_v})$，则 $\mu \vDash AbsRelInd(F,T,c,\mathbb{P}\cup X_v)$ 可满足。**

证明：若 $\mu_{X_{\mathbb{P}}\cup B_{X_v}} \vDash RelInd(\widehat{F}_{\mathbb{P}\cup X_v},\widehat{T}_{\mathbb{P}\cup X_v},\hat{c}_{\mathbb{P}\cup X_v})$，则存在一组 $X\cup X'$ 解释 $t$，$t$ 的抽象等于 $\mu_{X_{\mathbb{P}}\cup B_{X_v}}$ 且 $t \vDash T$。所以 $\mu_{X_{\mathbb{P}}\cup B_{X_v}} \cup t \vDash F(X_{\mathbb{P}}\cup B_{X_v}) \wedge c(X_{\mathbb{P}}\cup B_{X_v}) \wedge H_{\mathbb{P}\cup X_v}(X, X_{\mathbb{P}}\cup B_{X_v}) \wedge H_{\mathbb{P}\cup X_v}(X', X'_{\mathbb{P}}\cup B'_{X_v}) \wedge EQ_{\mathbb{P}\cup X_v}(X,\overline{X}) \wedge T(\overline{X},\overline{X}') \wedge EQ_{\mathbb{P}\cup X_v}(\overline{X}', X') \wedge \neg c(X'_{\mathbb{P}}\cup B'_{X_v})$，得证 2。

经由上述 1 和 2 成立，则 $RelInd(\widehat{F}_{\mathbb{P}\cup X_v},\widehat{T}_{\mathbb{P}\cup X_v},\hat{c}_{\mathbb{P}\cup X_v}) \leftrightarrow AbsRelInd(F,T,c,\mathbb{P}\cup$

$X_v$) 成立。

所以，对于这种混合抽象空间，可以用 $AbsRelInd(F, T, c, \mathbb{P} \cup X_v)$ 这个无量词的公式去替代 $RelInd(\hat{F}_{\mathbb{P}\cup X_v}, \hat{T}_{\mathbb{P}\cup X_v}, \hat{c}_{\mathbb{P}\cup X_v})$，成功规避了抽象转移关系中的量词。

### 4.1.2　泛化和精化

因为 IC3PA 和 IC3VA 的抽象变量都是布尔变量，即集合 $X_{\mathbb{P}\cup B_{X_v}}$ 中每个变量都是布尔类型。所以可以使用比特级 IC3 算法的标准泛化算法。并且，在尝试移除某个文字时，不需要考虑该文字来自于哪个抽象域，这也为两者的混合带来了便利。

从第三章的实验结果来看，谓词抽象比变量隐藏抽象更加灵活，其抽象反例精化得到的精度让抽象程度降低的“步幅”是动态的。相比而言，变量隐藏抽象每次精化的单位是一个变量，这种方式固然更加精细却也显得不够灵活，在属性只需关注变量某一特征的情况下就显得不是很适用。对于变量隐藏抽象而言，如果每次精化迭代都引入一个可见变量，当变量宽度较大，并且只需要关心变量某一特征（比如是否大于某值），而不需要关心具体值时，可能实际效果就不如加入一个谓词的效果好。因此可以考虑设置一个阈值，当关于某一变量的谓词的个数增加到该阈值时，认为该变量关心的特征数太多，可将该变量设为可见。

综上所述，对于混合 IC3PA 和 IC3VA 算法，精化的具体实现方案是：为每个变量维护谓词精度中出现的次数，每次精化默认选择谓词抽象域，每增加一个谓词，谓词中涉及的变量的次数都加 1，若某一变量出现在谓词精度中的次数大于它的位宽，则认为该变量和待验证的属性关联较大，将该变量加入可见变量的集合。

## 4.2　基于隐式混合抽象的 IC3 算法

通过 4.1 中的描述和证明，可以明确，谓词抽象和变量隐藏抽象可以通过隐式抽象的方式混合，更进一步，为了提升这种混合算法的普适性，我们希望可以将这种混合方式推广到更多其他类型的抽象。

通过 2.3 中关于抽象技术的介绍，可以知道，并不是所有抽象方法都有像变量隐藏抽象和谓词抽象那样明确的抽象变量集合。比如前面提到的语法制导的抽象方式，该抽象的精度是一些术语的集合，但是抽象状态是这些术语的不同划分，这种情况，混合抽象的抽象状态不再是具体状态在抽象变量集合上的投影。由于不同抽象可能会有不同的抽象状态表示方式，所以需要定义一种更加解耦的混合算法。据此，本文设计了一种基于隐式混合抽象的 IC3 算法，下面将给出详细介绍。

### 4.2.1　混合抽象空间

若有 n 种抽象，分别记做 $A_1$，$A_2...A_n$，混合抽象记做 $M$，这 n 种抽象的共同特性要求是都含有具体状态到抽象状态的映射关系，分别记做 $H_1$，$H_2...H_n$。和 4.1 类似，本节提出的混合抽象的混合方式是：当所有子抽象都将两个具体状态划分到同一抽象状态时，这两个状态才被划分到同一抽象状态。所以混合抽象的隐式抽象的约束函数是这些子抽象的约束函数的合取，即 $EQ_M(X,\overline{X}) \doteq \bigwedge_{i=1}^{n} EQ_{A_i}(X,\overline{X})$，并且混合抽象的抽象方式是 $H_M(X,X_M) \doteq \bigwedge_{i=1}^{n} H_{A_i}(X,X_{A_i})$。公式（2.6）的抽象版本记做 $AbsRelInd(F,T,c,M)$。

由于不同抽象有不同的立方体的定义，比如谓词抽象和变量隐藏抽象的定义是若干文字的合取，而语法制导的抽象的立方体是一组划分，所以现在没有一个普适的立方体结构，上述公式中的 $X_M$ 也并不一定是抽象变量的集合。本节提出的混合立方体的方案是定义一个混合立方体结构，该结构是一个 vector，其中存储了 n 个抽象中的立方体，记做 $hybrid\_cube_M$，且有 $hybrid\_cube_M \doteq< cube_{A_1}, cube_{A_2}..., cube_{A_n} >$。后期这个 $hybrid\_cube_M$ 会被看做一个整体进行各种操作。例如，对 $hybrid\_cube_M$ 取反操作，以及公式（2.6）中涉及的取 $hybrid\_cube_M$ 下一状态的操作等。

对于取反操作，实际上是对 $hybrid\_cube_M$ 表示的公式取反，假设各个抽象域上立方体转公式的函数分别是 $f_1, f_2..., f_i$，那么混合抽象上的转换函数是：$f(hybrid\_cube_M) = \bigwedge_{i=1}^{n} f_i(cube_{A_i})$，所以对混合立方体取非表示为：$\neg hybrid\_cube_M = \neg f(hybrid\_cube_M)$。

对于混合抽象中立方体的下一状态的表示则记做：$hybrid\_cube'_M \doteq< cube'_{A_1}, cube'_{A_2}..., cube'_{A_n} >$。其中 $cube'_{A_i}$ 表示抽象域 $A_i$ 上 $cube_i$ 经过一次迁移后的表示。

混合抽象空间中的帧的定义不会因为抽象而改变，仍可以用 IC3 算法中若干子句的集合来表示，仍然是若干子句的集合，即若干 $\neg hybrid\_cube_M$ 的集合。

### 4.2.2　混合泛化和精化算法

由于不同的抽象域可能会设计不同的泛化算法。若按照顺序来泛化，先泛化 $cube_{A_1}$，再泛化 $cube_{A_2}$，依次泛化直到 $cube_{A_n}$。在所有立方体都是文字集合的情况下，这种顺序泛化不如直接混在一起进行泛化效率高。

另一种直接的方式是各自泛化成更大的立方体然后拼到一起，但是这样的泛化方式存在问题。假设对 $cube_{A_i}$ 调用第 $i$ 个抽象域进行泛化，那么对这个抽象域来说，它看到的其他的立方体都是没有经过泛化的。在尝试丢弃 $cube_{A_i}$ 中某个文

算法 4.1　混合抽象泛化算法

```
1 Function GENERALIZATION(hybrid_cube,index):
2     tmp_res := hybrid_cube
3     for i := 1 to abstract_list.size:
        tmp_res[i] = abstract_list[i].GENERALIZATION(hybrid_cube, i, index)
4         // 尝试“贪心”泛化
5         if tmp_res ∧ I is sat or AbsRelInd(F_{index−1}, T, tmp_res, M) is sat:
6             // “贪心”失败，进行标准泛化
7             tmp_res[i] = hybrid_cube[i]
8             tmp_res[i] = abstract_list[i].GENERALIZATION(tmp_res, i, index)
9     return tmp_res
```

字时，即使公式（2.6）仍然不满足，但其前提是基于其他的立方体的集合保持不变，可是，各自泛化再合并时，如果有两个或以上的立方体都删去了文字，该前提失效。泛化算法中删除某文字时，公式（2.6）的检查可能因为另一个立方体范围的扩大而变为满足。总之，各自泛化再合并，会导致实际可达的状态也被误以为不可达而被排除，最后泛化得到的立方体会包括更多的状态，可能会删除了可达但是违背属性的坏状态，导致 unsafe 的任务被验证为 safe，出现验证错误。所以泛化时，不能把某个抽象域的立方体单独拿出来泛化。但是，另一方面，这种泛化方法可以快速删除状态，提升效率，只是需要谨慎的检查。可以在快速剔除状态的同时，通过归纳相关性检查来保证不会剔除多余状态。

混合抽象时的泛化算法流程如算法4.1所示。本文中，将直接泛化本抽象域的立方体的泛化方式叫做“贪心”泛化，将看得到其他抽象域立方体的泛化方式叫做标准泛化。首先，顺序遍历每个抽象域的立方体，对于第一个 $cube_{A_1}$，调用抽象域 $A_1$ 的泛化算法进行泛化，得 $\widetilde{cube_{A_1}}$。由于是第一个进行泛化，其他的抽象立方体保持不变，所以其实“贪心”泛化和标准泛化没有差别。可得 $\widetilde{hybrid\_cube}_M =< \widetilde{cube_{A_1}}, cube_{A_2}..., cube_{A_n} >$。从 $cube_{A_2}$ 开始，先使用贪心算法，即抽象域 $A_2$ 基于 $hybrid\_cube_M$，泛化 $cube_{A_2}$，得到 $\widetilde{cube_{A_2}}$，检查公式（2.6）是否为不满足，其中立方体应该为 $\overbrace{hybrid\_cube_M}^{} =< \widetilde{cube_{A_1}}, \widetilde{cube_{A_2}}..., cube_{A_n} >$，若公式不满足，说明贪心成功，更新 $\widetilde{hybrid\_cube}_M =< \widetilde{cube_{A_1}}, \widetilde{cube_{A_2}}..., cube_{A_n} >$。否则让抽象域 $A_2$ 基于 $\widetilde{hybrid\_cube}_M$，泛化 $cube_{A_2}$，即执行标准泛化，并用泛化结果更新 $\widetilde{hybrid\_cube}_M$，标准泛化结果不需要检查。然后继续对 $A_3$ 进行同样的泛化步骤。

可以看出，算法 4.1 区分泛化是“贪心”泛化还是标准泛化主要是通过传入抽象域的泛化函数 GENERALIZATION 中的参数。对于“贪心”泛化，传入的参数是原始的混合立方体，即 $hybrid\_cube$，这种情况下该抽象域看不到其他抽象域的泛化结果；对于标准泛化，传入的参数是当前泛化结果 $tmp\_res$，这种情况下抽象

域的泛化方法可以看到之前的抽象域的泛化结果。

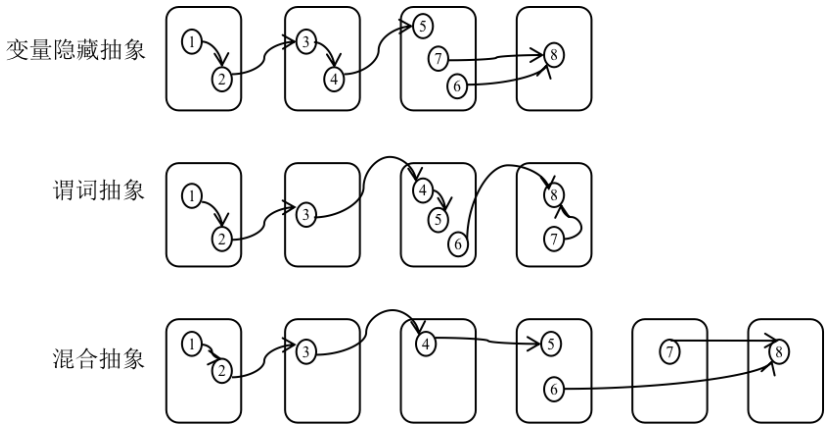



图 4.1　对变量隐藏抽象和谓词抽象进行混合抽象的示例

对于混合算法中的精化问题，这里为了便于理解，先给出一个对变量隐藏抽象和谓词抽象进行混合抽象的示例，如图4.1所示。在此例中，假设具体状态空间大小为 8，这些状态之间之间有 5 条转移，变量隐藏抽象和谓词抽象都将具体状态划算为 4 个抽象状态。保持两种精度不变情况下，将两种抽象混合，则得到的抽象状态数量为 6，比两种单独抽象的状态数都要多。由公式（4.2）知，只有在两个抽象空间里都能被划分到同一个抽象状态的具体状态才会被划分到一起。混合抽象降低了抽象层次，即 $\hat{T}_{\mathbb{P}\cup X_v} \to \hat{T}_{\mathbb{P}}$ 且 $\hat{T}_{\mathbb{P}\cup X_v} \to \hat{T}_{X_v}$。

由上可知，若混合抽象空间中出现了一条伪反例，那么该抽象反例一定在两个子抽象域内都是可以成立的，即也是伪反例。

**证明：**使用反证法，假设混合抽象中，伪反例是 $< \hat{s}_0, \hat{s}_1 ... \hat{s}_n >$。对于其中每个抽象状态 $s_i$ 表示的具体状态的集合，在谓词抽象中，一定有且仅有一个唯一包含该集合的抽象状态，假设是 $p_j$，即 $s_i \subseteq p_j$。即混合抽象中的每个抽象状态一定可以映射到子抽象域中的一个且唯一一个的抽象状态。所以一条混合抽象上的伪反例，可以对应到唯一一条子抽象域上的反例 $< \hat{p}_0, \hat{p}_1 ... \hat{p}_n >$，显然，由于不同的 $\hat{s}$ 可能会对到同一个 $\hat{p}$，这条长度为 n 的反例，可能会存在抽象状态的重复。如果这条反例不成立，说明其中有断点，显然，该断点只会存在于不同的 $\hat{p}$ 之间，那么断点两端的 $\hat{p}$ 对应的 $\hat{s}$ 之间也不会有连接，所以混合抽象上该伪反例就不成立了。所以每个子抽象上该反例都可以成立。

通过上述证明可知，只要混合抽象域中存在一条伪反例，那么对应的子抽象域上该伪反例一定可达。所以两个子抽象域上都可以进行精化，且只要某一个子抽象域上反例被消除，则混合抽象中伪反例也被消除。考虑到目前不同抽象之间精度没有交互模块，所以精化方案主要以单抽象域精化和多抽象域精化的方案为主。例如顺序精化，直到某抽象域精化成功，消除反例，如果所有都无法精化消除

反例，则返回 unknown。

## 4.3　基于隐式混合抽象的 IC3 验证工具框架

本文设计并实现了基于隐式混合抽象的 IC3 算法框架。框架模块及整体流程设计如图 4.2，其中每个模块的结构设计如算法 4.2所示。

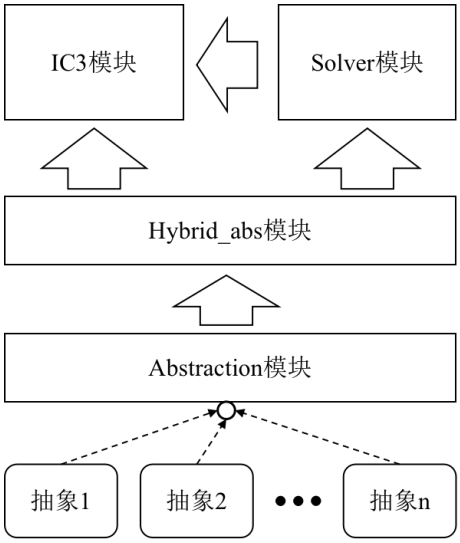



图 4.2　基于隐式混合抽象的 IC3 验证工具框架模块设计图

IC3 模块负责 IC3 算法的主流程，主要负责维护帧的 list，其流程主要有两层循环。

第一层循环：不断调用 Solver 的 *get_bad_cube* 方法得到当前 frame 中的坏的立方体。直到找不到，调用 *new_frame* 增加新的 frame，执行 *propagate_block_cube* 进入传播阶段。传播阶段发现两个 frame 相同时，IC3 返回 safe。

第二层循环：发生在 Solver 的 *get_bad_cube* 方法得到 *bad_cube* 之后，IC3 调用 *rec_block_cube* 方法进入阻塞该立方体的循环。*rec_block_cube* 为该立方体创建一个证明义务的队列并将其插入，然后不断地从队列中弹出级别更低的待证明的义务，调用 Solver 的 block 方法阻塞。如果阻塞失败，Solver 返回前驱，IC3 将其插入队列；如果阻塞成功，IC3 调用 *Hybrid_abs* 泛化该立方体并加入到 frame 中；然后 IC3 继续取出剩余待证明的证明义务直到队列为空。如果弹出一个处于第 0 层的待证明的义务，IC3 调用 *Hybrid_abs* 的 refine 方法尝试精化。*Hybrid_abs* 的精化如果精化成功，则 IC3 继续后面的流程；如果是伪反例且无法从中提出有效消除该反例的精度，则 IC3 返回 unknown；如果确认该反例为真，IC3 返回 unsafe。

Solver 模块给 IC3 模块提供求解功能。*get_bad_cube* 求解当前帧和 $\neg P$ 相交的坏立方体, *is_initial* 判断立方体和初始条件是否有交集, *is_blocked* 函数则通过公

算法 4.2　框架主要模块的内部结构设计

```
1  class Abstraction{
2          Set<Precision> s;\\ 精度集合
3          void initial_abs_space( );\\ 初始化抽象空间
4          bool refine(Vec<cube> cex);\\ 从反例中提取精度
5          Cube generlize(Cube cube);\\ 泛化 cube
6          Cube get_cube_form_model( );\\ 从验证结果中提取 cube
7          Cube get_next(Cube cube));\\ 获得 cube′
8          Cube concretize(Cube cube);\\ 具体化 cube
9  }
10 class Hybrid_Abstraction{
11         Vector<Abstraction> s;\\ 抽象集合
12         void initial_abs_space( );\\ 初始化抽象空间
13         bool refine(Vec<cube> cex);\\ 从反例中提取精度
14         Hybrid_Cube generlize(Cube cube);\\ 泛化 cube
15         Hybrid_Cube get_cube_form_model( );\\ 从验证结果中提取 hybrid_cube
16         Hybrid_Cube get_next(Cube cube);\\ 获得 hybrid_cube′
17         Hybrid_Cube concretize(Cube cube);\\ 具体化 hybrid_cube
18 }
19 class Solver{
20         Hybrid_Abstraction hybrid_abs;\\ 混合抽象对象
21         Hybrid_Cube get_bad_cube( );\\ 得到 F_k ∧ P̄ 的解
22         bool is_initial(Cube cube);\\ 判断 cube ∧ Ī 是否可满足
23         bool is_blocked(Cube cube);\\ 判断 cube 是否被阻塞
24         Hybrid_Cube block(Cube cube);\\ 阻塞 cube
25 }
26 class IC3{
27         Transition_System ts;\\ 转移系统
28         Solver solver;\\ Solver 对象
29         Hybrid_Abstraction hybrid_abs;\\ 混合抽象对象
30         Proof_Queue queue;\\ 证明义务的队列
31         void proof( );
32         void rec_block_cube( );
33         void propagate_block_cube( );
34         void generalize_and_push( );
35 }
```

式 $F \wedge c$ 判断立方体是否已经被阻塞，如果公式检查为不满足，则此时已被阻塞，否则调用 block 方法去阻塞。Block 函数通过公式 $F \wedge \neg c \wedge T \wedge c'$ 来判断，如果该公式检查为不满足，说明阻塞成功，若失败，返回给 IC3 前驱立方体。Solver 模块在求解公式时会调用 $Hybrid\_abs$ 模块对立方体的操作，如 $get\_cube\_from\_model$，$get\_next$，$concretize$ 等。

Hybrid_Abstraction 模块一方面负责对 Solver 模块提供各项操作混合立方体的

功能，如 *get_cube_from_model*，*get_next*，*concretize* 等；一方面负责对 IC3 模块提供泛化和精化功能。其作用主要是管理和协调子抽象域的相关功能，达到对混合的立方体进行操作的目的。*get_cube_form_model* 和 *get_next* 以及 *concretize* 等函数主要是调用各个子抽象的对应方法，然后按序存到 vector 中，形成 *hybrid_cube*。而 *refine* 和 *generalize* 函数负责调度各个子抽象的精化和泛化算法。*generalize* 函数见算法 3.2。

Abstraction 模块负责给其他子抽象提供接口。抽象域需要实现接口定义的函数才可加入到框架中。*initial_abs_space* 负责初始化求解环境，即从转移系统中提取本抽象域可以利用的精度集合 $\mathbb{P}$，对该集合增加相关隐式约束 $EQ_{\mathbb{P}}(X, \overline{X})$。函数 *refine*，从反例中提取新的精度，增加相关隐式约束。函数 *get_cube_from_model* 则是负责从验证为 sat 的公式的模型中提取自己的立方体。

对应上述模块，从整体设计上看，框架主要分为 5 层，第一层是 IC3 算法的主流程，第二层是求解器，第三层是混合的抽象方法，第四层是抽象方法应该实现的接口，第五层则是接口的不同抽象的实现。

因此，对于更多其他符合要求的抽象方法，也可以按照上述方式，实现抽象方法需要实现的接口，就可以加入到混合抽象的框架中来。

## 4.4　实验与分析

本章 4.1 节设计了混合 IC3PA 和 IC3VA 算法，记为 IC3PA+VA，论文按照其对应的流程设计，在 IC3VA 的代码基础上，做了对应的修改，实现了 IC3PA+VA 算法。同时 4.2 节设计了基于隐式混合抽象的 IC3 算法，并给出了其对应的软件工具框架的设计，实现了基于隐式混合抽象的验证工具，记做 IC3IA。为了测试这两种新的字级 IC3 算法的验证效果，论文在第三章所选取的测试数据集上，进行了同样的对比实验。下面将展开介绍。

### 4.4.1　实验环境与配置

为了公平起见，本次实验的环境与配置与第三章完全一致，机器配置也如表3.1所示。同样地，所有算法在验证每个测试样例时，也只会被分配一个逻辑运算核用于计算，单个测试样例的验证时间上限同样也被设为 15 分钟，即 900 秒。

### 4.4.2　实验数据说明

对于本次实验的测试数据，这里同样也选取了第三章实验部分从多个来源收集的总共 487 个硬件安全属性验证的任务，也按照同样的分组方式，分别在较为

简单的任务集 A 和较为复杂的任务集 B 上进行了实验。

### 4.4.3　实验与分析

首先来看 4.1 节中提出的 IC3PA+VA 算法与 IC3PA 算法的对比。实验总体结果如表4.1和表4.2 所示，结合表3.2中 IC3VA 算法的表现对比来看，相比于 IC3VA 算法，此次实验中，IC3PA+VA 算法无论是求解任务个数还是超时情况都有所提升；但是相较于 IC3PA 算法，在较为复杂的任务集 B 上，IC3PA+VA 可解的任务的数量明显更少，同时在两者共同解决的任务上，IC3PA+VA 的耗时也稍微更多一些。总体来看，IC3PA+VA 的表现虽然好于 IC3VA，但整体验证效果还是不如 IC3PA。

分析造成上述对比结果的原因，可能是因为 IC3PA+VA 的算法的核心其实是当 IC3PA 算法中关于某个变量的谓词数增多，就将其变为可见变量，即附加将其可见的约束，所以该算法本质和 IC3PA 接近。特别地，将谓词转向可见变量的阈值设置得比较低时，该算法其实无限接近 IC3PA 算法。值得注意的是，IC3PA+VA 虽然减少了精化迭代次数，但是时间上相比 IC3PA 没有优势，这可能是因为把变量转为可见变量后，为了保证算法是增量的，只含有可见变量的谓词没有删除，造成了状态空间的冗余。因此可以看出，IC3PA+VA 算法仍然具有较大的局限性，由于内部耦合度较高的抽象混合方式，相比更为灵活的 IC3PA 算法，更多的抽象此时带来的增益十分有限，因此仍然需要进一步降低混合的耦合度。

表 4.1　IC3PA+VA 两任务集上的求解情况

| 任务集 | 算法 | safe | unsafe | to | unk | unique |
|---|---|---|---|---|---|---|
| 任务集 A （共 163 个） | IC3PA | 116 | 40 | 7 | 0 | 4 |
| | IC3PA+VA | 115 | 40 | 5 | 3 | 2 |
| 任务集 B （共 324 个） | IC3PA | 74 | 15 | 230 | 5 | 17 |
| | IC3PA+VA | 64 | 12 | 172 | 76 | 3 |

表 4.2　IC3PA+VA 和 IC3PA 在共同解决的任务上的情况

| 任务集 | 算法 | 时间/秒 | 精化次数 | 求解次数 |
|---|---|---|---|---|
| 任务集 A （共同解决 152 个） | IC3PA | 623 | 424 | 392896 |
| | IC3PA+VA | 710 | 281 | 394373 |
| 任务集 B （共同解决 73 个） | IC3PA | 4147 | 302 | 891914 |
| | IC3PA+VA | 4418 | 208 | 996712 |

接下来，对基于隐式混合抽象的 IC3 算法进行实验分析。根据本章 4.3 节中实现的基于隐式混合抽象的 IC3 算法验证工具，可知此算法的一大特点和优势是子

抽象类型和精化方式是可以自由配置的。这里抽象域的配置是谓词抽象和变量隐藏抽象，精度的选择方式比较简单。一共设置了 6 种不同配置的基于隐式混合抽象的 IC3 算法来进行对比实验，依据选择的子抽象类型和精化方式的不同，6 种配置分别是：VA（只进行变量隐藏抽象），PA（只进行谓词抽象），VA+PA（同时进行变量隐藏抽象和谓词抽象，但是精化时首选变量抽象），PA+VA（同时进行谓词抽象和变量隐藏抽象，但是精化时首选谓词抽象），VA+PA+ALL 和 PA+VA+ALL（同时进行变量隐藏抽象和谓词抽象，精化在两个抽象域上都进行，两者都是两个抽象域精化但是精化顺序不同）。这里的首选指的是一个抽象域消除不了反例时选择另一个抽象域。实验结果如表4.3所示。

oracle 指在当前测试集上所有配置所能解决任务的并集。表4.4给出了框架的各个配置与 IC3PA 验证结果对比，对比项由多个不同的验证指标组成，包括两者共同解决的任务个数，在这些任务上的对比耗时（单位为秒，格式为该配置耗时/IC3PA 耗时），在这些任务上的精化次数耗时（格式为该配置精化次数/IC3PA 精化次数）。

表 4.3　各个配置在两任务集上的求解情况

| | 测试集 A | | | | | 测试集 B | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 配置 | safe | unsafe | to | unk | unique | safe | unsafe | to | unk | unique |
| IC3PA | 116 | 40 | 7 | 0 | - | 74 | 15 | 230 | 5 | - |
| VA | 106 | 41 | 12 | 4 | 2 | 64 | 10 | 187 | 63 | 15 |
| PA | 116 | 42 | 5 | 0 | 2 | 74 | 15 | 229 | 6 | 11 |
| VA+PA | 111 | 42 | 10 | 0 | 2 | 75 | 17 | 231 | 1 | 19 |
| PA+VA | 116 | 42 | 5 | 0 | 2 | 78 | 16 | 229 | 1 | 14 |
| VA+PA+ALL | 111 | 42 | 10 | 0 | 2 | 78 | 14 | 231 | 1 | 20 |
| PA+VA+ALL | 112 | 42 | 9 | 0 | 2 | 72 | 14 | 236 | 2 | 12 |
| *oracle* | 116 | 42 | - | - | - | 94 | 18 | - | - | - |

根据表 4.3中的结果可知，相比于 IC3PA+VA 算法，使用了更低耦合度的抽象组合方式后，基于隐式混合抽象的 IC3 算法在可解决任务个数上有了明显的提升。即便和 IC3PA 算法相比，基于隐式混合抽象的 IC3 算法所能解决的任务总数也明显更多。在任务集 A 上，PA 配置和 PA+VA 配置都比 IC3PA 多解决 2 个 unsafe 任务，在更为复杂的任务集 B 上，PA+VA 配置则多解决了 4 个 safe 任务和 1 个 unsafe 任务。而在唯一性方面，各个配置都解决出了 IC3PA 算法不能解决的验证任务。特别是在任务集 B 上，每个配置都有 10 个以上的唯一解决的任务，而 VA+PA 和 VA+PA+ALL 配置接近 20 个。oracle 配置给出了基于隐式混合抽象的 IC3 算法的实验效果的理论上限，即如果我们可以将这 6 种配置组合在一起，并适配一个最

表 4.4　各个配置下与 IC3PA 共同解决的任务上的时间和精化次数对比

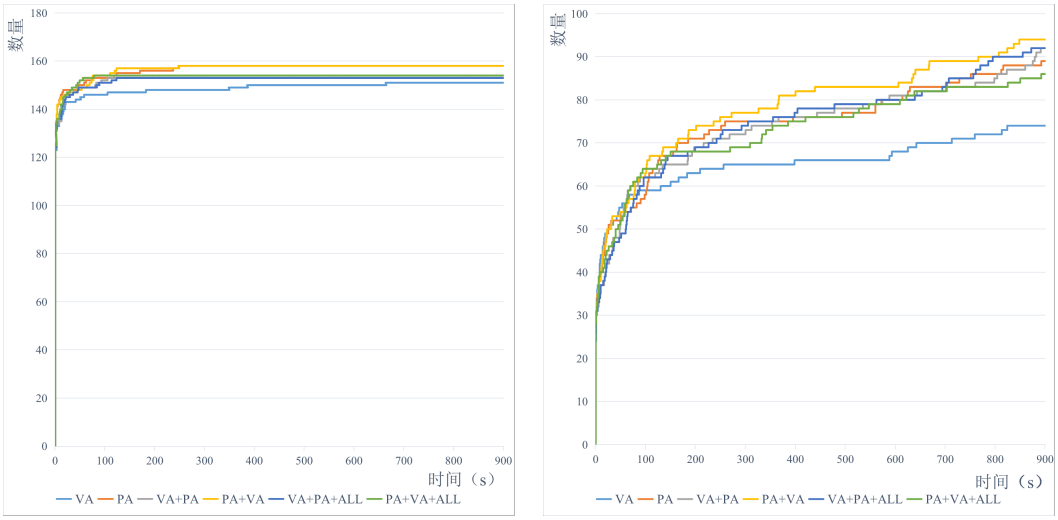
| | 测试集 A | | | 测试集 B | | |
|---|---|---|---|---|---|---|
| 配置 | 同解 | 耗时对比 | 精化对比 | 同解 | 耗时对比 | 精化对比 |
| VA | 145 | 1979/1070 | 32/404 | 59 | 3152/4269 | 0/212 |
| PA | 156 | 1296/1187 | 481/481 | 78 | 7675/6686 | 408/437 |
| VA+PA | 151 | 649/1085 | 55+5/421 | 73 | 7515/6732 | 233+201/435 |
| PA+VA | 156 | 1082/1187 | 0+438/481 | 80 | 8600/6971 | 2+418/456 |
| VA+PA+ALL | 151 | 710/1085 | 55+55/421 | 72 | 6814/6212 | 196+196/391 |
| PA+VA+ALL | 152 | 512/1085 | 56+56/422 | 74 | 6920/6216 | 204+204/393 |

合理的选择策略时，在任务集 B 上算法可以多解决 20 个 safe 任务、3 个 unsafe 任务，可验证任务总数提升 26%。另外，框架版 VA 相比 IC3VA 效果更差，这可能是因为代码解耦之后开销增大。而框架版 PA 和 IC3PA 相比效果不变，说明框架版泛化方法弥补了解耦带来的开销。

算法效率方面，从表 4.4中可以看出对比结果。首先，两个任务集上，框架版 PA 和 IC3PA 在共同可解的任务上精化次数接近，耗时前者略大于 IC3PA，这是由于基于隐式混合抽象的 IC3 算法增添了更多的抽象类型和组合逻辑，算法整体更加复杂，规模也更庞大。在相对简单的任务集 A 上，4 个抽象组合的配置耗时都少于 IC3PA。特别的，在配置 VA+PA 上，其总体精化次数较少，VA 失败了 5 次，但配合 PA 精化之后，最终整体时间明显短于 IC3PA，验证效率提升了 1.67 倍。配置 PA+VA 没有用到 VA 精化，但由于解耦带来的消耗，耗时更多。在更为复杂的任务集 B 上，由配置 VA+PA 可以看到，VA 失败了 201 次，因而此时整体效率上就没有什么优势。这可能是因为，对于较为复杂的任务，变量隐藏抽象的精度不适合用于初始化。

为了更好的展示混合抽象带来的效果，图4.3(a)和图4.3(b) 分别展示了各个配置在任务集 A 和任务集 B 上的解决任务个数随时间变化折线图。横轴表示时间，纵轴表示该时间内可解任务的个数。根据坐标轴的含义可知，曲线越靠近 x 轴效果越差，因为同样时间内解决的任务个数更少。越靠近 y 轴效果越好，代表同样时间内可解任务最多，效率最高。

可以看到，不论是任务集 A 还是任务集 B 上，配置 PA+VA 整体曲线趋势都是最靠近 y 轴的，并且曲线终点的高度也是最高的。说明该配置表现最好。另外，两个任务集上 VA 也是表现最差的，这说明 VA 的效果还有待优化。另外 VA+PA+ALL 在任务集 A 上表现较差，在任务集 B 上表现第二，这可能是因为任务集 A 比较简单，VA 配合 PA 不如 VA 单独抽象效果好。另外，PA+VA+ALL 在两个任务集上表




(a) 任务集 A　　(b) 任务集 B

图 4.3　各个配置在任务集 A 和 B 上解决任务个数随时间变化折线图

现都不算好，这说明如果每一次精化时在每个子抽象上都同时精化，可能会导致冗余，从而导致精度计算效率降低。所以，如果每次精化时，算法可以利用有效的启发式策略去选择合适的抽象域，则可能可以有效提升验证效率。

综合上述分析，依据各个实验的验证结果，可以得出结论如下：

1. 基于隐式混合抽象的 IC3 算法框架是有效的。在任务集 A 和 B 上，各个配置下的基于隐式混合抽象的 IC3 算法都解决了部分 IC3PA 无法解决的任务。具体地，在任务集 A 上，有 2 个配置的任务解决个数多于 IC3PA，在任务集 B 上，有 3 个配置的任务解决个数多于 IC3PA。特别地，各个配置的可解任务的并集数量达到了 112 个，验证效果的理论上限得到了明显提升。
2. 基于混合抽象思路的 IC3 算法是有效的。根据图4.3(a)和图4.3(b)中的折线图显示，虽然在引入更多抽象之后，算法在规模和复杂度提升了不少，但是同样时间内，PA+VA 这个配置可解数量几乎一直保持最多，优于配置 PA 和 VA，而且根据曲线的趋势，可以看出随着验证时间限制的增大，其和单抽象域配置的验证结果差距会更加明显。这种趋势表明，混合抽象的验证效果符合我们希望综合两种子抽象优势的初衷。
3. 基于隐式混合抽象的 IC3 算法仍然存在可提升空间。从实验结果来看，PA+VA+ALL 和 VA+PA+ALL 的配置表现一般。这说明，如果每次精化都在所有抽象域上执行，可能会导致精度冗余不够灵活。精化时抽象域的选择是一个重要的问题，后续可以考虑设计一个精化时抽象域选择的启发式策略来提升验证效果。另外，实验时实现的抽象域是变量隐藏抽象和谓词抽象，后续应该尝试加入更多的抽象域到框架中，进一步提升验证效果。

## 4.5　本章小结

本章首先设计了一种将 IC3VA 和 IC3PA 混合的抽象 IC3 算法，记做 IC3VA+PA，形式化的对混合后的归纳相关检查的公式进行了编码，并给出了方法可行性的证明。同时设计了这两种抽象混合后的泛化和精化方法。对于泛化，由于两种抽象都有比特级的抽象状态变量，所以可以将抽象状态变量混合后，用标准的 IC3 泛化方法进行泛化。对于精化，默认先通过谓词抽象进行精化，当某个变量的相关谓词个数大于变量的位宽时把变量变为可见变量。

进一步，论文设计了将 IC3VA 和 IC3PA 混合推广到多个抽象域混合的 IC3 算法。算法的核心是通过隐式抽象去混合多个抽象域的抽象约束，只有所有的抽象域中都将两个具体状态分到同一抽象状态时才将它们分到同一抽象状态。并给出了混合抽象 IC3 算法中混合立方体的定义，以及对立方体相关操作的定义。由于不同抽象下的 IC3 算法对立方体的泛化方式可能不同，本章还设计了一种通用的混合抽象 IC3 算法的泛化方案，并给出了泛化算法和基于隐式混合抽象的 IC3 算法的流程图。

然后本章给出了基于隐式混合抽象的 IC3 验证工具的框架结构图和各个模块的伪码说明。最后通过实验对比分析了框架版工具的 6 种配置下的实验结果，证明了基于隐式混合抽象的 IC3 算法验证工具框架确实可以提升可解任务的数量，也融合了多个抽象的验证特点，解决出了单抽象算法不能解决的任务。

# 第 5 章　总结与展望

本章首先对论文的研究工作进行总结，然后根据实验表现和工作内容提出未来优化的方向。

## 5.1　研究总结

随着硬件技术的飞速发展，硬件设计的规模和复杂度急剧增长，字级硬件模型检测算法的研究开始受到关注，研究者们希望可以将较高抽象的字级语义信息，加入到模型检测过程中，提升验证的效果。IC3 算法作为传统比特级模型检测算法中应用最为广泛和成功的算法，对其进行字级的拓展改造，成为了一种字级模型检测研究的方向。所以，本篇论文的研究目标定为设计并实现一种新的字级 IC3 算法。本文主要工作可以总结如下：

1. 通过对已有的字级 IC3 算法——IC3PA 进行分析，发现 IC3PA 在特定验证任务中会出现关于某个变量的谓词个数过大的问题，以此为动因，论文尝试将变量隐藏抽象和 IC3 算法相结合，同时为了与变量隐藏抽象适配，论文也给出了对应的新的精化和泛化方案，最终得到了一个新的字级 IC3 算法——IC3VA。为了验证 IC3VA 算法的效果，论文还设计了相关对比实验。实验结果表明，IC3VA 算法是有效的，对于一些 IC3PA 会出现关于某个变量的谓词个数过大的验证任务，IC3VA 可以加速验证。
2. 根据不同基于抽象的字级 IC3 算法具有差异化验证效果的这一观察，论文希望可以综合多种抽象的特点提升验证效果，设计使用隐式混合抽象，将多种基于抽象的字级 IC3 算法进行混合抽象，并给出了隐式混合抽象下的精化和泛化方法，最终设计出一种可扩展的基于隐式混合抽象的 IC3 算法。
3. 为了验证基于隐式混合抽象的 IC3 算法的有效性，论文给出了对应的验证工具的框架设计，并完成了工程实现，证明了方案的可行性。基于此框架，论文实现了谓词抽象、变量隐藏抽象以及混合抽象，并使用多种不同配置进行了对比验证实验。实验结果显示，混合抽象配置下的验证效果要好于其他单一抽象配置，在两个任务集上可解的任务个数最多，证明基于隐式混合抽象的 IC3 算法的确实现了综合各个抽象特点的设计初衷。考虑此框架对于抽象是可扩展的，未来可以继续在框架中加入更多的抽象方式，进一步提升验证效果。

## 5.2　未来工作展望

本篇论文设计了一种基于隐式混合抽象的字级 IC3 算法，实验结果证明了算法的有效性，也体现出算法实现了综合各个抽象方式特点的设计初衷。但是考虑整个研究过程中出现的一些问题，以及对实验结果的分析思考，不难发现当前工作仍然有很多可以进一步优化和完善的地方，未来的工作可能可以从以下几个方面来考虑。

1. 本文设计的 IC3VA 算法实验结果虽反映了 IC3VA 具有和 IC3PA 不同的验证特点，但是整体验证效果仍有待加强。目前发现影响效率的主要原因可能是为了保证算法正确性，初始精度集合是在初始条件和属性中出现的所有变量，这种方式容易导致初始抽象程度过低，进而导致 IC3VA 算法耗时过长。未来可以考虑添加一些更加灵活的抽象程度和精度选取策略，从而进一步提升 IC3VA 的验证效果。
2. 第四章实验中的 4 个组合配置在每次精化时都是固定的抽象域选择策略，但是一个任务验证期间可能会遇到多次精化，每次精化抽象域的选择都对验证过程有影响。未来可以为框架设计更加灵活的精化方式，比如每次精化都动态选择抽象域。
3. 为了增加验证的灵活性，未来可以考虑在框架中增添一个抽象域交互模块，比如精度调整，但是这需要代码进一步解耦，需要对整体设计进行一定的修改。
4. 目前只是在框架中加入了谓词抽象和变量隐藏抽象。未来应该继续尝试在框架中加入更多风格不同的抽象方式，比如基于语法制导的抽象，进一步提升验证效果。

# 参考文献


[1] Bradley A R. Sat-based model checking without unrolling[C]//International Workshop on Verification, Model Checking, and Abstract Interpretation. Springer, 2011: 70-87.

[2] Cimatti A, Griggio A, Mover S, et al. Ic3 modulo theories via implicit predicate abstraction [C]//International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 2014: 46-61.

[3] Cimatti A, Griggio A, Mover S, et al. Infinite-state invariant checking with ic3 and predicate abstraction[J]. Formal Methods in System Design, 2016, 49(3): 190-218.

[4] Kropf T. Introduction to formal hardware verification[M]. Springer Science & Business Media, 2013.

[5] Wolf C. Yosys manual[J]. Retrieved January, 2021, 16: 2021.

[6] Clarke Jr E M, Grumberg O, Kroening D, et al. Model checking[M]. 2018.

[7] Burch J R, Clarke E M, McMillan K L, et al. Symbolic model checking: 1020 states and beyond [J]. Information and computation, 1992, 98(2): 142-170.

[8] Biere A, Cimatti A, Clarke E, et al. Symbolic model checking without bdds[C]//International conference on tools and algorithms for the construction and analysis of systems. Springer, 1999: 193-207.

[9] De Moura L, Rueß H, Sorea M. Bounded model checking and induction: From refutation to verification[C]//International Conference on Computer Aided Verification. Springer, 2003: 14-26.

[10] Eén N, Mishchenko A, Brayton R. Efficient implementation of property directed reachability [C]//2011 Formal Methods in Computer-Aided Design (FMCAD). IEEE, 2011: 125-134.

[11] Biere A. The AIGER And-Inverter Graph (AIG) format version 20071012: Report 07/1[R]. Altenbergerstr. 69, 4040 Linz, Austria: Institute for Formal Models and Verification, Johannes Kepler University, 2007.

[12] Niemetz A, Preiner M, Wolf C, et al. Btor2, btormc and boolector 3.0[C]//International Conference on Computer Aided Verification. Springer, 2018: 587-595.

[13] Brummayer R, Biere A, Lonsing F. Btor: bit-precise modelling of word-level problems for model checking[C]//Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning. 2008: 33-38.

[14] Clarke E, Grumberg O, Jha S, et al. Counterexample-guided abstraction refinement[C]// International Conference on Computer Aided Verification. Springer, 2000: 154-169.

[15] Long D E, Clarke E M, Bryant R E, et al. Model checking, abstraction, and compositional verification[R]. CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE, 1993.

[16] Kurshan R P. Computer-aided verification of coordinating processes: the automata-theoretic approach: volume 302[M]. Princeton university press, 2014.

[17] Graf S, Saidi H. Construction of abstract state graphs with pvs[C]//International Conference on Computer Aided Verification. Springer, 1997: 72-83.

[18] Das S, Dill D L. Successive approximation of abstract transition relations[C]//Proceedings 16th Annual IEEE Symposium on Logic in Computer Science. IEEE, 2001: 51-58.

[19] Ball T, Podelski A, Rajamani S K. Boolean and cartesian abstraction for model checking c programs[C]//International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 2001: 268-283.

[20] Goel A, Sakallah K. Model checking of verilog rtl using ic3 with syntax-guided abstraction [C]//NASA Formal Methods Symposium. Springer, 2019: 166-185.

[21] Johannsen P. Speeding up hardware verification by automated data path scaling[D]. Christian-Albrechts Universität Kiel, 2002.

[22] Bjesse P. Word level bitwidth reduction for unbounded hardware model checking[J]. Formal Methods in System Design, 2009, 35(1): 56-72.

[23] Andraus Z S, Sakallah K A. Automatic abstraction and verification of verilog models[C]// Proceedings of the 41st annual Design Automation Conference. 2004: 218-223.

[24] Brady B A, Bryant R E, Seshia S A, et al. Atlas: automatic term-level abstraction of rtl designs [C]//Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010). IEEE, 2010: 31-40.

[25] Lee S. Unbounded scalable hardware verification.[D]. 2016.

[26] Birgmeier J, Bradley A R, Weissenbacher G. Counterexample to induction-guided abstraction-refinement (ctigar)[C]//International Conference on Computer Aided Verification. Springer, 2014: 831-848.

[27] Jain H, Ivančić F, Gupta A, et al. Using statically computed invariants inside the predicate abstraction and refinement loop[C]//International Conference on Computer Aided Verification. Springer, 2006: 137-151.

[28] Ernst M D, Perkins J H, Guo P J, et al. The daikon system for dynamic detection of likely invariants[J]. Science of computer programming, 2007, 69(1-3): 35-45.

[29] McMillan K L. Interpolation and sat-based model checking[C]//International Conference on Computer Aided Verification. Springer, 2003: 1-13.

[30] Beyer D. Reliable and reproducible competition results with benchexec and witnesses (report on sv-comp 2016)[C]//International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 2016: 887-904.

[31] Cimatti A, Griggio A, Sebastiani R. Efficient generation of craig interpolants in satisfiability modulo theories[J]. ACM Transactions on Computational Logic (TOCL), 2010, 12(1): 1-54.

[32] Clarke E, Kroening D, Sharygina N, et al. Predicate abstraction of ansi-c programs using sat[J]. Formal Methods in System Design, 2004, 25(2): 105-127.

[33] Lahiri S K, Bryant R E, Cook B. A symbolic approach to predicate abstraction[C]//International Conference on Computer Aided Verification. Springer, 2003: 141-153.

[34] Tonetta S. Abstract model checking without computing the abstraction[C]//International Symposium on Formal Methods. Springer, 2009: 89-105.

[35] Ball T, Podelski A, Rajamani S K. Relative completeness of abstraction refinement for software model checking[C]//International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 2002: 158-172.

[36] Henzinger T A, Jhala R, Majumdar R, et al. Abstractions from proofs[J]. ACM SIGPLAN Notices, 2004, 39(1): 232-244.

[37] Griggio A. a simple, open source ic3-based model checker for infinite-state systems.[EB/OL]. 2021[2021-02-02]. https://es-static.fbk.eu/people/griggio/ic3ia/.

[38] Goel A, Sakallah K. Empirical evaluation of ic3-based model checking techniques on verilog rtl designs[C]//2019 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, 2019: 618-621.

[39] Jain H, Kroening D, Sharygina N, et al. Vcegar: Verilog counterexample guided abstraction refinement[C]//International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 2007: 583-586.

[40] Mukherjee R, Tautschnig M, Kroening D. v2c–a verilog to c translator[C]//International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 2016: 580-586.

[41] Irfan A, Cimatti A, Griggio A, et al. Verilog2smv: A tool for word-level verification[C]//2016 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, 2016: 1156-1159.

[42] Wolf C. Yosys open synthesis suite[EB/OL]. 2019[2019-08-26]. http://www.clifford.at/yosys/.

[43] Stefano T, Alessandro C, Alberto G. Verification modulo theories[EB/OL]. 2021[2021-04-21]. https://es.fbk.eu/projects/vmt-lib.

[44] Armin B, Nils F, Mathias P. Hardware model checking competition 2020[EB/OL]. 2020[2020-09-24]. http://fmv.jku.at/hwmcc20/.

# 致　谢

衷心感谢贺飞老师对我科研上的指导，让我体会到了科研的有趣和不易。这三年也让我的心智得到了磨炼，各方面都成长了许多。

衷心感谢实验室全体同学对我的帮助和支持，祝福各位不负所求，终得所寻。

衷心感谢舍友晓昱、麻慧给我生活上的帮助，给我的支持和鼓励，让我的校园生活增添了许多欢乐。

衷心感谢我不在老家的发小们，虽然很久难见一面，还是感谢你们经常线上联系我，关心我，让我的心很温暖，给我带来了很多欢乐。

衷心感谢储超群同学的出现，感谢你对我的包容，感谢你这两年像战友一样陪我并肩作战，陪我笑和哭，感谢我们一起经历的所有。

衷心感谢我的父母这么多年对我支持和爱护，感谢你们对我无私的爱。

最后感谢自己，安然无恙又三年，心智成熟又三年。

# 声　明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签　名：_____ 日　期：_____

# 个人简历、在学期间完成的相关学术成果

## 个人简历

1997 年 08 月 30 日出生于湖北省黄陂区。

2014 年 9 月考入中南财经大学信息管理与信息系统专业，2018 年 7 月本科毕业并获得管理学学士学位。

2018 年 9 月硕士研究生统一招生考试进入清华大学软件学院攻读软件工程硕士学位至今。

## 在学期间完成的相关学术成果

无。

# 指导教师学术评语

本论文研究硬件验证算法，主要完成的工作包括：1. 设计了基于隐式变量隐藏抽象的 IC3 算法；2. 设计了基于隐式混合抽象的 IC3 算法；3. 实现了上述算法，并在实际硬件案例上进行了实验验证。硬件代码验证对于电子设计自动化具有重要意义。该同学在著名的 IC3 硬件验证框架上，实现了一种基于混合抽象的硬件模型检测算法，能够综合多种抽象的验证优势，提升整体验证效率。论文整体结构合理，论述完整，层次明晰，反映论文作者掌握了扎实的软件工程基础理论知识，具备了独立开展工程技术和方法研究工作的能力，达到了软件工程硕士的学术水平，同意组织论文答辩。

# 答辩委员会决议书

杨柳同学的论文利用混合抽象对硬件验证算法展开研究，选题具有理论意义和应用价值。论文的主要工作如下：

1、设计并实现了基于变量隐藏抽象的字级 IC3 算法；

2、设计并实现了基于混合抽象的字级 IC3 算法，并进行了实验验证。

论文结构合理，层次分明。论文反映了杨柳同学掌握了软件工程领域的基础理论和专业知识，以及解决工程问题的技术方法和手段，具备了独立担负工程技术工作的能力。论文答辩清楚，回答问题正确。答辩委员会一致同意通过毕业论文答辩，建议授予杨柳同学工程硕士学位。