北京航空航天大学计算机学院

硕士学位论文文献综述

论文题目：安全 C 编译器的构建和形式验证方法的研究与实现

专　　业：软件工程

研究方向：软件形式建模与验证

研 究 生：陈志伟

学　　号：SY1406108

指导教师：马殿富 教授

北京航空航天大学计算机学院

2015 年 12 月 23 日

摘　要

编译器是重要的系统开发工具，其安全可靠性对安全攸关软件的开发有着重要影响，为此，人们引入了多种方法来验证编译器的可靠性。本文首先介绍了软件测试的验证方法。编译器的测试通常有两种策略：动态测试和静态测试，对编译器的测试通常采用动态测试的方法。文中详细叙述了动态测试中的白盒测试和黑盒测试技术，及测试用例的自动生成技术。接着介绍了三种形式化验证方法。模型检验方法在工业上硬件的验证上获得了巨大的成功，但在软件方面，特别是编译器这种系统软件，由于其状态空间可以是无限大，会产生状态爆炸的问题。抽象方法是解决状态爆炸问题的一个重要的手段。定理证明方法是现如今编译器形式化验证研究的热点，文中介绍了基于霍尔逻辑和分离逻辑的定理证明思路。使用逻辑框架是定理证明的一个比较好的方式，框架提供了统一的编程语言和逻辑表示，因此表达能力强，可以很方便地证明程序的可靠性。翻译确认是通过证明源代码和目标代码的语义等价性来证明编译器的正确性的一种技术，它对编译器的具体实现不敏感，具备可重用性，因此更易于实现对编译器的形式验证，在编译器验证、测试以及维护中都得到了广泛的应用。最后，本文总结了主要介绍的内容，并指出把当前的处于实验室的试验阶段的形式化验证方法尽快应用于工业界是目前的迫切需要解决的问题。

关键词：安全攸关软件，形式化验证，编译器

Abstract

The compiler is an important development tool, its trust has an important impact on the safety-critical software development, to this end, and people have introduced a variety of methods to verify the trust of the compiler. This paper firstly introduces the verification method of software testing. Compiler test usually includes two strategies: dynamic testing and static testing. The compiler test commonly used method of dynamic testing. This paper describes a technique to automatically generate dynamic testing case and white box and black box testing method. Then three formal verification methods are introduced. The model checking method gains a great success on the industrial hardware verification, but in terms of software, especially a system software like compiler, because its state space can be infinite, will cause state explosion problems. Abstract method is an important method to solve the problem of state explosion. Theorem proving method is now a hot field in compiler formal validation. The paper describes the proof ideas based on Hoare logic and discrete logic theorem. Using a logical framework is a good way to prove theorems. The framework provides a unified programming language, logic representation, and strong communication skills, so you can easily prove the trust of the program. Translation Validation is a technology which is used to prove the correctness of compiler through semantic equivalence between source code and object code. It's not sensitive to the specific implementation of compiler. With reusability, it is easier to achieve the compiler formal verification. What's more, Translation Validation has been widely used in the compiler verification, testing, and maintenance. Finally, the paper summarizes the main content, and noted that the urgent issue now is that the formal verification should be applied in industry as soon as possible.

Keywords: the safety-critical software, formal verification, compiler

目　　录

安全 C 编译器的构建和形式验证方法的研究与实现

1 概述

随着计算机应用的飞速发展，软件已渗透到国民经济和国防建设的各个领域，在信息社会中扮演着至关重要的角色。安全攸关软件，如航空机载软件，作为各类安全关键系统的构成部分，其内部结构越来越复杂、应用环境越来越开放，这些因素使得人们更加关注其安全可靠性问题。因此，对航空机载软件尤其是大型客机机载软件进行安全性分析、设计以及适航验证变得尤为重要。

目前航空领域中主要采用的验证标准是美国航空无线电委员会（RTCA）于 1992 年 12 月发布的航空适航认证标准体系 DO-178B[1]《机载系统和设备认证中的软件要求》标准。DO-178B 规定了机载软件的设计和开发进程，并描述目标的可追踪性过程，按照可能引起航空器不同的失效状态将机载软件划分为 A、B、C、D、E 五个软件等级，分别对应灾难性的，严重的、较重的、较轻的和无影响的五类失效状态。然而随着软件开发新技术新方法的不断涌现，需要对 DO-178B 作一定的补充和修订以适应当前机载软件的开发。RTCA 于 2012 年发布了 DO-178C[2]。DO-178C 对 DO-178B 的补充有四个方面：软件工具验证、基于模型的开发和验证、面向对象编程、形式化方法。DO-178C 弥补了 DO-178B 的不足，在 MBD 和 OO 方面进行明文规定，强调了双向追溯性，对详细的 MBD 和 OO 设计标准详细规定，明确指出类型的一致性等。同时，对于在 DO-178B 中没有明确标准的内存管理，也在 DO-178C 中另立条款，做详细的要求。

编译器作为软件开发过程中的关键工具，其安全性如何？是进行软件开发所面临的重要课题。特别在安全关键领域中，如航天、核工业等，编译器的安全可信有着至关重要的作用，编译错误可能会带来灾难性的后果。然而随着计算机技术的发展，对编译的要求越来越高，例如高级语言编译器中增加了大量的优化，

而优化又可能带来不可预测的问题，导致编译过的可执行代码在运行过程中产生非期望的输出。因此在软件开发阶段必须对编译器的正确性进行充分验证。

编译器通常被认为是语义等价转换的一个过程，但是编译器特别是优化编译器是个复杂的符号转换程序。消除编译错误的传统方法是大量的测试，但是测试难以达到完全覆盖，并不能充分地保证编译器的安全可信。近年来，形式化验证方法在编译器的正确性验证中得到了持续的关注。形式化验证方法可以从数学角度对编译器进行描述，对编译过程的语义和语言属性的等价性进行证明，能够充分地保证编译器安全可靠。形式化验证是分析系统是否具有所期望性质的过程，主要分有两种途径：模型检验和定理证明。在本文中，还将介绍另一种形式验证方法——翻译确认技术。翻译确认通过证明源代码和目标代码的语义等价性来证明编译器的正确性，避免了对编译器自身的验证，同时又具有很好的可重用性，近年来在编译器验证领域得到了广泛研究，已取得令人瞩目的成果。

最后，如果为了实践的原因有必要在安全相关系统中使用 C 语言，那么必须对语言的使用加以限制，避免那些确实可以产生问题的地方，直到它是可以应用的。MISRA-C，汽车制造业嵌入式 C 编码标准，从 MISRA-C:2004 开始其应用范围扩大到其他高安全性系统，由该规范定义的C语言被认为是易读、可靠、可移植、易于维护的。C 安全子集将 MISRA-C 与航天型号软件的特点相结合，重新定义了一系列 C 语言软件的编程准则，为安全相关领域的 C 语言软件提供了相应的安全语言规范和编译要求。C 安全子集严格要求了编译器的成熟度及稳定性，编译器必须忠实地反映源语言的代码结构和语义，以方便编译前后的代码审查、比较和追踪，确保编译后代码的可信。本文以此为出发点，尝试以形式化验证技术为基础，将语言规范与经过验证的编译器相结合，以提高编译的安全可靠性，减少工作流程和工作量，达到行业规范要求。

2 基本研究现状及发展趋势

2.1 测试方法

软件测试是通过执行软件来判断软件是否具备所期望的性质,是软件开发中一个行之有效的、必不可少的、客观地评估软件正确性的方法。在高可信软件开发中，软件测试的开销往往大于 50%。软件测试往往需要测试软件的健壮性、可靠性、完备性等性能，具体的来说，即测试软件是否可以处理异常输入，其实际执行结果是否与预期结果相同,测试其功能说明中的应该实现功能是否完全实现，其不该实现的功能是否完全没有实现,对编译器的测试同样需要进行上述几方面。

编译器测试[3]常用的有两种策略:动态测试和静态测试。动态测试是使用一系列由测试环境控制的测试数据执行程序,然后比较测试程序的实际输出结果和测试用例中的预期输出结果。静态测试就是在不执行程序的前提下对程序进行的测试。对编译器的测试往往使用动态测试策略,选择一个已被验证的编译器测试套件对编译器进行测试。动态测试可以被划分为白盒测试技术和黑盒测试技术两种。

白盒测试技术是基于对编译器内部结构的检查，即使用测试数据对程序的控制结构、数据流等逻辑进行检查。白盒测试技术是基于对测试源代码的利用，主要用来测试编译器的单独模块，以保证其每个部分都可以正常运作。相反，基于测试套件的黑盒测试往往被用来对编译器进行性能测试和鲁棒性测试,测试数据只取决于编译器的规格说明书。测试数据不关心被测程序的结构，只关注被测程序的特征和外部行为。黑盒测试技术被用来测试编译器对编程语言声明和对用户的接口,确认编译器对语言标准的实现情况已经成为黑盒测试的一个越来越重要的方向。

为保证编译器的高可靠性，测试人员需要依据编译器的功能说明书或是被编译语言标准对其进行大量的覆盖测试（如规则覆盖测试、分支覆盖测试等）。由于编译器自身的复杂性，进行覆盖测试的耗费的时间和工作量往往不可计量，单纯的手工的测试方法更是难以满足编译器测试的要求,自动的生成测试用例对编译器进行测试已经成为当前的研究热点。

测试用例自动生成技术分为 3 类：随机测试用例生成、静态测试用例生成和动态测试用例生成。其中，随机测试用例生成方法，是通过在输入空间内不断地随机选择有效的输入值以生成测试用例。其优点是快速且易于实现，缺点是其盲目性导致的对复杂程序的测试效果不佳。静态测试用例生成，是指在不执行源程序的情况下，利用符号执行等静态分析方法获取的数据来生成测试用例。其优点是分析速度快，缺点是受到静态分析的影响，精度不够高，往往误报情况较多。动态测试用例生成,是指利用执行已生成的测试用例的反馈信息不断地修正测试用例以生成新的测试用例。该方法优点是生成的测试用例更有针对性，缺点是由

于需要运行待测程序，因而其分析速度较慢，而且其生成的测试用例的质量因其使用的算法的不同而有差异。

目前，对于安全攸关系统，软件测试技术依旧面临着重大挑战。安全攸关软件不仅要求在其环境处于正常状态时保证系统的安全性，而且要求环境处于非正常状态时也能使系统安全地进入安全状态，因而往往难以获得充分的数据来测试软件应付危险情况的能力，不能满足可靠安全性测试的需求。软件工程中分析、设计和测试技术目前仍是在工程上保障软件高可信性的主要手段，但不能满足高可信软件开发的需要。从基础上看，这些技术需要与高可信软件的程序理论结合，提高技术的形式化程度。

2.2 模型检验方法

模型检测[4~6]是一种自动形式化验证技术，用于对一个计算机系统的正确行为属性进行判断。

2.2.1 基本思想

模型检测的基本方法是用一个状态迁移图 M 来表示所要检测的系统的模型，并用模态/时序逻辑（如计算树逻辑（computation tree logic, CTL）、命题线性时序逻辑（linear temporal logic, LTL）、命题 μ 演算等）公式 φ 来描述系统的正确行为属性，然后通过对模型状态空间穷举搜索来判断该公式是否能够在模型上被满足。如果公式在模型上满足，即 M |= φ，则系统的正确性得到证实（verified）；否则，就表明系统中存在错误，即 M |= ~φ，系统正确性被证伪（falsified）。

2.2.2 模态逻辑

模态/时序逻辑是模型检测的基础。下面将主要介绍三种常用的模态逻辑：计算树逻辑[7]、命题线性时序逻辑[8]和命题 μ 演算逻辑[9~10]。

2.2.2.1 计算树逻辑

一个系统的运行可以看成是系统状态的变化。系统状态变化的可能性可以表示成树状结构。

计算树逻辑（CTL-Computation Tree Logic）是一种分叉时序逻辑。 CTL 可以描述状态的前后关系和分枝情况。描述一个状态的基本元素是原子命题符号.。CTL 公式由原子命题，逻辑连接符和模态算子组成.。CTL 的逻辑连接符包括：

¬（非），∨（或），∧（与）。它的模态算子包括：E（Exists），A（AIways），X（Next-time），U（Until），F（Future），G（Global）。E 表示对于某一个分枝，A 表示对于所有分枝，X 表示下一状态，U 表示直至某一状态，F 表示现在或以后某一状态，G 表示现在和以后所有状态。前两个算子描述分枝情况，后四个算子描述状态的前后关系。CTL 中描述分枝情况和描述状态的前后关系的算子成对出现，即一个描述分枝情况的算子后面必须有一个描述状态的前后关系的算子。

CTL 公式的产生规则如下：原子命题是 CTL 公式；如果 p，q 是 CTL 公式，则¬p, p∨g，p∧g，EXp，E(pUg)，EFp，EGp，AXp，A(pUg)，AFp，AGp 是 CTL 公式。对 CTL 公式存在线性时间的模型检测算法，即算法的最坏时间复杂度与|S|*|F|成正比，这里|S|是状态迁移系统的大小，|F|是逻辑公式的长度。

2.2.2.2 命题线性时序逻辑

命题线性时序逻辑（PLTL-PropositionaI Linear TemporaI Logic）关心的是系统的任意一次运行中的状态以及它们之间的关系。PLTL 公式由原子命题，逻辑连接符和模态算子组成。PLTL 的逻辑连接符包括：¬，∨，∧。它的模态算子包括：◇（EventuaIIy），□（AIways）。◇表示现在或以后某一状态（类似 CTL 的 F），□表示现在和以后所有状态（类似 CTL 的 G）。

PLTL 公式的产生规则如下：原子命题是 PLTL 公式；如果 p, q 是 PLTL 公式，则¬p，p∧g，p∨g，◇p，□p 是 PLTL 公式。例如公式◇□p 表示：某个时刻后所有的状态都满足 p。

PLTL 和 CTL 的表达能力不同，各有各的长处。PLTL 模型检测的常用方法是将所要检测的性质即 PLTL 公式的补转换成 Buchi 自动机，然后求其与表示系统的自动机的交。如果交为空，则说明系统满足所要检测的性质；否则生成一个反例，说明不满足的原因。

2.2.2.3 命题 μ 演算逻辑

系统状态的改变总是某种动作引起的。μ 演算关心的是系统的动作与状态之间的关系。描述动作的基本元素是动作符号。μ 演算公式由原子命题、命题变量、逻辑连接符、模态算子和不动点算子组成。逻辑连接符包括：∨，∧，对于每个动作符号 a，有两个模态算子［a］和〈a〉。［a］表示对所有的 a 动作，〈a〉表示对某个 a 动作。不动点算子有最小不动点算子 μ 和最大不动点算子 v。

μ 演算公式的产生规则如下：原子命题和命题变量是 μ 演算公式；如果 p，q 是 μ 演算公式，则 p∨q，p∧q，［a］p，〈a〉p，μX.p，vX.p 是 μ 演算公式，例如公式 μX.(p∨［a］X）表示：在任何无穷的 a 路径都存在某个满足 p 的状

态。

μ 演算的主要缺点是公式不易读懂（由于最小、最大不动点的交错嵌套），其优点是它的表示能力非常强。CTL 和 PLTL 都可以嵌入到它的真子集中，并且相应的子集具有与 CTL 和 PLTL 相同的复杂度的模型检测算法，这引起了很多学者的注意。

2.2.3 状态爆炸问题

模型检测在实际中应用的主要瓶颈是状态爆炸问题（state - explosion problem）。由于模型检测基于穷举搜索对正确属性进行判断，所以它适用于对有限的、状态空间较小的系统进行分析。然而，在实际应用中经常存在着状态空间庞大的系统。如对于软件系统，由于存在着无限数据域（如整数）、无界数据类型（如链表）、以及复杂的控制结构（如递归），导致软件系统的状态空间可以是无限大，直接对它们进行模型检测在实际中是不可行的。因此，正如图灵奖得主 Clarke 所说，解决状态爆炸问题是模型检测研究中的一个最根本的工作[11]。

2.2.4 抽象方法

抽象方法是解决状态爆炸问题的一个重要的方法。它的基本思想是首先构造一个比原系统的具体模型小的有限抽象模型，然后通过正确属性在抽象模型上的检测结果推测出其在原来的具体模型上是否可满足。抽象方法的依据是，对于所给定的某种正确属性而言，待检测的原系统的许多信息（如某些程序变量的取值、进程的标识符、调用栈中活动记录等）是无关的。因此，这些信息可以从具体模型中抽象出去。这样不仅简化了模型，同时保留了必要的信息，使得抽象的模型检测得以有效地进行[12~13]。

软件模型检测中的抽象的主要过程为：

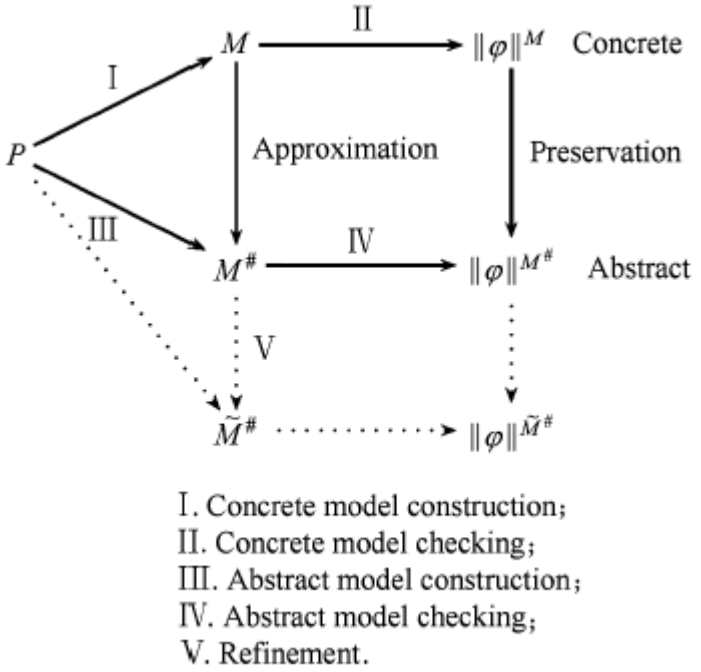

Fig.1 Overview of abstraction in software model Checking.

给定一个程序 P 以及欲对其进行分析的行为属性 φ，具体的软件模型检测的过程主要包括 2 个步骤：1）模型构建。使用一个状态转换系统 M 来表示程序 P 的语义（如图 1 中 I 所示），称其为 P 的模型。2）模型检测。用时序逻辑公式 φ 描述所关心的行为属性，通过计算$\|\varphi\|^M$的值来验证（如图 1 中 II 所示）。由于程序的状态空间巨大甚至是无限，通常无法直接构造模型 *M*。因此我们需要构造一个精简的有限抽象模型 $M^{\#}$（如图中 III 所示），使用它来逼近具体模型 M。通过抽象实现对 φ 在 $M^{\#}$上的有效验证（如图 1 中 IV 所示），并进一步根据 $M^{\#}$与 M 之间的逼近关系所对应的程序属性的保持关系，判断 φ 在 M 上成立与否。当由于抽象导致无法得到确定的模型检测结果时，需要通过抽象精化（abstraction refinement）构造更加精确的抽象模型（如图 1 中 V 所示）。

抽象精化包含了抽象以及精化的自动过程，它从一个粗糙的原始抽象模型开始，通过迭代精化，直到得到一个包含足够信息能够对属性做出确定判断的抽象模型。实际中通常采用一种基于反例引导的抽象精化方法来对抽象模型进行精化[14]。一般来说，抽象反例是指在抽象模型中能够证明属性无法成立的状态路径。精化的目标是通过在程序的具体模型上对反例的分析，去除虚假反

例，获得新的信息来构造更加精确的抽象模型。这些信息可以通过计算最弱前置条件，Craig 插值或者分析由 SAT 判定器产生的不可满足性结果来获得。

2.3 定理证明方法

定理证明技术是将软件系统和性质都用逻辑方法来规约,通过基于公理和推理规则组成的形式系统,以如同数学中定理证明的方法来证明软件系统是否具备所期望的关键性质。基于定理证明的形式化验证技术可以看作是以软件系统为公理获得其性质的证明过程。本质上，编译器是一个符号转换程序，因此可以为编译过程建立完整的数学模型,利用这个模型方便地对编译过程正确性进行形式化证明。

2.3.1 编译器验证方法

2.3.1.1 霍尔逻辑方法

霍尔逻辑[15~16]描述程序正确性的一般形式为：

$$\{Pre\}P\{Post\}$$

其中，Pre 称为前置断言，Post 称为后置断言，P 为程序代码。若 P 的每一次计算开始于满足 Pre 的状态，执行终止且终止时的状态满足 Post，则正确性公式为真，程序 P 具有完全正确性。

在霍尔逻辑中存在一组证明规则，称为霍尔规则。这些规则是语法制导的，它们把证明一条复合命令的部分正确性断言简化成证明它的直接子命令的部分正确性断言。霍尔规则中的赋值规则和推论规则如下所示：

赋值规则：

$$\{P[E/x]\}\ x := E\{P\}$$

表示把 P 中变量 x 的所有自由出现都替换成表达式 E 得到的谓词。赋值公理表示如果执行赋值语句 x := E 后 P 为真，则在执行赋值语句之前 P[E/x]为真。

推论规则：

$$\frac{P' \rightarrow P\ \{P\}\ S\ \{Q\}\ Q \rightarrow Q'}{\{P'\}\ S\ \{Q'\}}$$

表示如果一个 Hoare 三元组为真，那么把前置条件加强或者后置条件减弱也为真。

用霍尔规则进行推导能得到部分正确性断言的形式化证明，所以霍尔逻辑能用于机器证明。在证明的过程中，过于形式化的推导会分散人们在证明过程

上的精力，可以把这方面的工作交给一些辅助证明工具如LCF或HOL去完成。实践中，使用霍尔逻辑证明很小的程序的正确性也不是那么容易。后续的研究工作集中于扩展霍尔逻辑，以便验证更复杂的语言结构，以及寻求更好的方式来构造、表示和检查形式证明。例如最弱后置条件，谓词转移语义的观点，以及验证条件生成器和自动定理证明器进行程序的自动验证等。

2.3.1.2 分离逻辑方法

在命令式程序验证方面，基于经典逻辑的霍尔逻辑得到了广泛的应用。但是，对使用指针的命令式语言程序进行推理验证是困难的。分离逻辑[17~18]是对霍尔逻辑的一个扩展，通过提供表达显示分离的逻辑连接词以及相应的推导规则，消除了共享的可能，能够以自然的方式来描述计算过程中内存的属性和相关操作，从而简化了对指针程序的验证工作。分离逻辑被证明具有更强的验证能力，如对并发程序和资源管理的验证，使得程序验证和推理技术前进了一大步。因此，继霍尔逻辑之后，分离逻辑有望成为程序形式验证的一种重要方法。

分离逻辑中，前置条件和后置条件中的程序状态主要由栈 s 和堆 h 构成，栈是变量到值的映射，而堆是有限的地址集合到值的映射。在程序验证时，可以将栈看作对寄存器内容的描述，而堆是对可寻址内存内容的描述。分离逻辑中引入了两个新的分离逻辑连接词：分离合取*和分离蕴含-*。[P*Q] s h 表示整个堆 h 被分成两个不相交的部分 h_0 和 h_1，并且对子堆 h_1 断言 P 成立，而对子堆 h_1 断言 Q 成立。形式化表示如下：

$$[P*Q]\,s\,h \overset{\text{def}}{=} \exists h_0\ h_1.h_0 \perp h_1 \wedge h_0 \cdot h_1 = h \wedge P\,s\,h_0 \wedge Q\,s\,h_1$$

其中 $h_0 \perp h_1$ 表示堆 h_0 和 h_1 不相交，$h_0 \cdot h_1$ 表示堆 h_0 和 h_1 的联合。

[P_*Q] s h 表示如果当前堆 h 通过一个分离的部分 h'扩展，并且对 h'断言 P 成立，则对扩展后的堆(h · h')断言 Q 成立。形式化表示如下：

$$[P\text{-}*Q]\,s\,h \overset{\text{def}}{=} \forall h'.(h' \perp h \text{ and } P\,s\,h') \rightarrow Q\,s\,(h \cdot h')$$

同样。分离逻辑引入了一些新的推导规则。如 Frame 规则

$$\frac{\{P\}C\{Q\}}{\{P*R\}C\{Q*R\}}$$

其中，代码段 C 不会对断言 R 中的自由变量赋值。分离逻辑作为一种近年来提出的逻辑，因其本身所蕴含的分离思想，在验证包含指针的程序时，能够简洁、优雅地支持进行局部推理和模块化推理，已经在程序验证领域得到了重视和广泛

使用。但是，将分离逻辑用于解决更复杂的软件系统的源代码级验证仍然需要进一步解决许多技术难题，如对语言类型的支持范围、验证过程的自动化程度等。

2.3.2 计算机辅助定理证明器

定理证明目前比较好的方式是使用编程和证明统一的框架，如 PVS、Coq 和 Isabelle 等。

2.3.2.1 PVS

PVS[19]是原型验证系统(Prototype Verification System)的缩写。该系统主要包括规约语言和定理证明器两部分，并且还集成了解释器、类型检查器及预定义的规约库和各种工具。

PVS 提供的规约语言基于高阶逻辑，具有丰富的类型系统，是一般适用的语言，表达能力很强，大多数数学概念、计算概念均可用该语言自然直接地表示出来。PVS 的定理证明器以交互方式工作，同时又具备高度的自动化水准。它的命令的能力很强，琐屑的证明细节为证明器的内部推理机制掩盖，使得用户仅在关键决策点上控制证明过程。PVS 为计算机科学中严格、高效地应用形式化方法提供自动化的机器支持。

2.3.2.2 Coq

Coq[20]是一种基于归纳构造演算的高阶逻辑交互式定理证明辅助工具，由法国国家计算机科学与控制研究所（INRIA）开发。它使用形式化语言来编写数学定义，执行算法和证明定理，开发满足规范说明的程序。它是目前国际上交互式定理证明领域的主流工具之一，具有强大的数学模型基础和很好的扩展性，并有完整的工具集。

2.3.2.3 Isabelle

Isabelle[21]是一个通用的定理证明助手。它允许数学公式用形式语言来表示，并提供以逻辑演算的方式来证明这些公式的工具。主要的应用是数学证明的，尤其是形式验证，其中包括证明计算机硬件或软件的正确性，证明计算机语言和协议的特性。

Isabelle 的主要证明方法是基于高阶联合的解析（resolution）。虽然是交互式的，Isabelle 也提供自动化推理工具，例如项重写引擎（term rewriting engine），真理树证明器（tableaux prover）以及各种决策过程。

2.4 翻译确认方法

2.4.1 形式语义

程序设计语言的语义通常会被分为两类：静态语义（static semantics）和运行时语义（runtime semantics）[22]。

2.4.1.1 静态语义

语言的静态语义（static semantics）提供一组规则，说明哪些语法上合法的程序实际上是有效的。这样的规则通常会要求所有标志符都有声明，操作符和操作数类型兼容，而且过程调用的参数个数是正确的。这些规则的共同之处在于它们都无法使用上下文无关文法来表示。因此静态语义扩展了上下文无关的规范，使得有效程序的定义得以完整。

静态语义可以通过形式化或非形式化的方式来说明。非形式化的方式通常比较简短容易阅读，但是往往不够精确。形式化的规范可以使用各种不同的记法来表示，如属性文法可以对编译器中能够找到的很多语义检查进行形式化，但会显得繁杂和冗长，绝大多数编译器开发系统都不会直接使用属性文法。作为替代，它们会通过程序的抽象语法树（AST）来传播语义信息。

2.4.1.2 运行时语义

运行时语义，或者执行语义，用来说明程序计算的是什么。这些语义通常会在语言手册或报告中以一种很不形式化的方式进行说明。存在很多种形式化的方法来定义程序语言的运行时语义，下面将主要介绍三种：公理语义、指称语义以及操作语义。

公理语义

公理语义可以用在比操作模型更为抽象的层次对程序的执行进行建模。它们的基础是形式化说明的关系（relation）或断言（predicate），用来建立程序变量之间的关联。因此语句的定义也就可以表示为它们如何修改这些关系。

公理的方法有利于程序正确性证明的推导，因为它可以避开实现细节，专注于语句的执行会如何改变变量之间的关系。虽然公理可以对程序设计语言语义中的重要属性进行形式化描述，但是它很难被用来完整定义绝大多数的程序设计语

言。

指称语义

指称语义是采用形式系统方法，用相应的数学对象（如 set, function 等）对一个即定形式语言的语义进行注释的学问。指称语义还可以解释为：存在着两个域，一个是语法域，在语法域中定义了一个形式语言系统；另外一个是数学域（或称之为已知语义的形式系统）。用一个语义解释函数，以语义中的对象（值）来注释语法域中定义的语言对象的语义，即为指称语义。由于指称语义的理论支持是论域方程，在函数空间解这些论域方程需要不动点理论，于是也有人说：“指称语义就是不动点语义”。

指称语义较为常用，并且已经成了程序设计语言严格定义的基础。研究表明，可以自动把指称表示形式转换为等价的可以直接执行的表示形式。

操作语义

操作语义是用机器模型语言来解释语言语义的，也就是将语言成分所对应的计算机的操作作为语言成分的语义，一般来说它与编译程序有直接关系。在定义操作语义时，被采用的机器模型是 SECD 机器。Landin(1964 年)第一个在 SECD 机器上定义了 λ-表达式的操作语义，虽然现在看来这种定义方法并不那么美，也不如后来由 G.Plotkin 采用的规约系统方法清楚，但 Landin 的工作是有重大意义的。

操作语义学除定义“做什么”之外，主要是定义“怎么做”，所以属性文法属于操作语义的范畴。

2.4.2 编译过程正确性

编译过程正确性的形式化定义可用如下图 2 的转换示意图表示，故对其形式化证明就是证明对应的转换示意图[23]的成立。

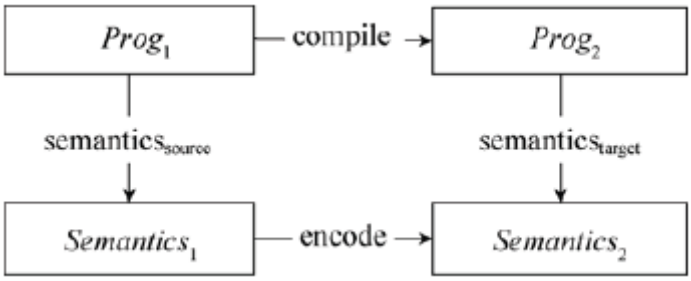

Fig.2 Compiler correctness diagram

图中的箭头可看成是函数映射过程。编译正确性可以用如下等式来表示：

$$encode(semantics_{source}(P)) = semantics_{target}(compile(P))$$

程序 P 可以使用不同的语义来解释，如操作语义、公理语义、指称语义等。这种示意图最早起源于 McCarthy 和 Painter 的工作，他们使用操作语义证明了源语言是简单数学表达式的编译器的正确性[24~25]。

图 2 证明编译过程的正确性具有通用性，但是如何将源程序与编译后目标程序的语义关联对应起来是证明的关键问题。对于命令式语言,程序的语义可以建立在抽象执行机器的基础上。程序的运行可以看作为抽象机器状态的转换。操作语义可以很自然地定义这种转换。设 q 为程序运行过程中某个抽象机器状态，f(q)表示程序在 q 状态下执行下一条指令后进入的状态。源程序与编译后目标程序的语义关联使用图 3 来表示。

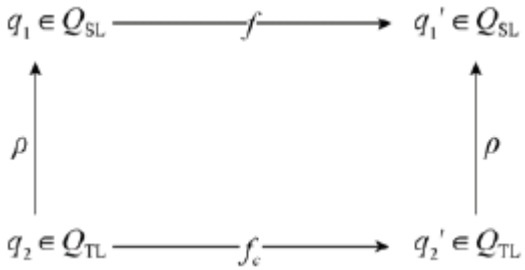

Fig.3 Contact of program semantics

图中 Q_{SL}、Q_{TL} 分别表示源程序和目标程序运行时抽象机器状态集合， 函数 ρ 表示源语言抽象机器状态和目标语言抽象机器状态之间的对应关系。证明编译过程具有语义保持性，即需要证明：

$$\forall q \in Q_{TL},\ \rho \circ f_c(q) = f \circ \rho(q)$$

成立即可。

2.4.3 基本思想

翻译确认[26~27]是一种用于确认编译器或代码生成器的源和目标之间的语义等价性的形式化方法，它通过证明源代码和目标代码的语义等价性来证明编译器的正确性。使用翻译确认方法需要构造一个确认器（validator），确认器在编译器每一次运行后形式化地证明生成的目标代码是源代码的一个正确翻译。确认器不关心编译器的具体实现，只对编译器的源代码和目标代码进行处理，如果验证成功则编译继续进行，如果发现语义矛盾之处则输出一个警报或取消编译。

编译器的形式化验证可以减弱为对确认器进行形式化验证工作。相对于编译器而言，确认器的形式化验证工作是比较简单的，从而大大减轻了证明的难度及工作量。同时，由于确认器不关心编译器的具体实现，因此没有限制编译器的设计以及未来的优化完善等，且确认器是可重用的。

2.4.4 证明过程

一个自动化的翻译确认器应该包括以下要素：（1）一个用于描述源语言和目标语言的公共语义框架；（2）基于公共语义框架形式化地建立的目标代码和源代码之间的“正确执行”定理；（3）一个有效的证明方法，它允许证明代表着生成的目标代码的一个语义框架的模型，正确的实现了代表着源代码的另一个模型；（4）通过 Analyzer 执行证明方法的自动化，如果成功则生成一个证明脚本；（5）一个证明检查器，用于对 Analyzer 产生的证明脚本进行检查。

翻译确认的过程如下图 4 所示：

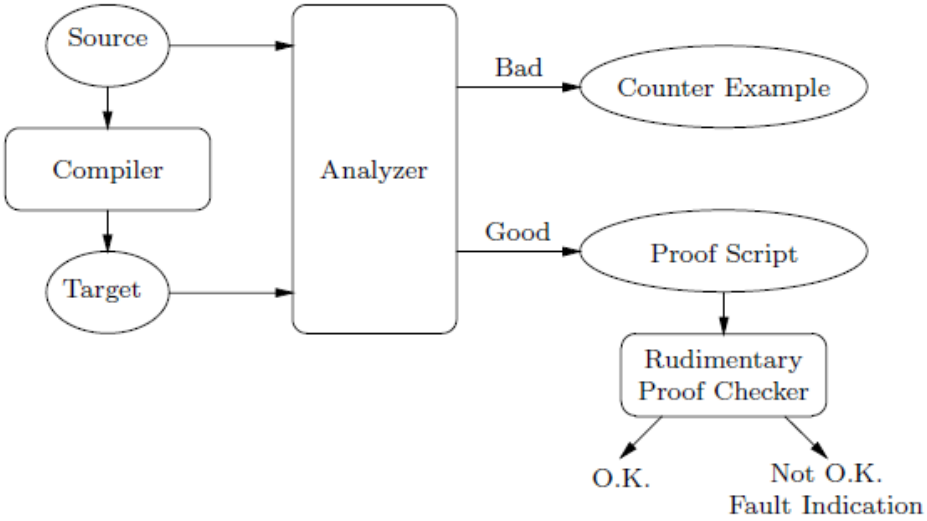

Fig.4 Translation validation process

分析器接收源程序和目标程序作为输入。如果分析器发现生成的目标程序正确的实现了源程序，它会产生一个详细的证明脚本。如果分析器无法建立源程序和目标程序之间的正确对应关系，它会产生一个反例。该反例包括了其中生成的代码行为不同于源代码的情景。因此，该反例提供的证据表明，编译器有故障，需要加以修改。

2.4.5 进展和趋势

翻译确认方法自出现以来得到了广泛的研究与应用。目前，翻译确认方法从不同的角度可以分为以下研究方向。

从使用对象来说，可以分为：1）面向翻译过程的翻译确认，其确认的对象是整个编译器或两种不同中间语言之间的编译过程，确认从源语言到目标语言的编译过程中的语义等价性。2）面向翻译优化[25]的翻译确认。现代的高级编译器中包含着大量的优化，翻译确认更多地应用于对局部编译优化的验证。在对编译优化的验证中，将确认器接收优化前和优化后的代码作为输入，建立证明条件，对二者的语义等价性进行证明，输出验证结果。

从适用范围来说，可以分为：1）对于翻译确认的通用架构的构造；2）用于实际编译器的通用确认算法的开发。虽然可以使用通用技术如符号执行、模型检验和定理证明等算法实现对编译器的确认工作，对应的算法也能适用于大部分的

程序转换，但是这些确认器有着高复杂性且可能会产生错误的警告；3）用于特定编译或优化的专用确认器，是针对某个特殊的优化或一组相关的优化的确认器。

翻译确认方法发展至今，日趋成熟，已呈现出一些新的特点：

（1）算法更多样化，不拘泥于特定的语义框架或模型，将各种技术融入于翻译确认算法中，如模型检验、自动定理证明、数据流分析及符号执行等；

（2）适用范围更广泛，从对同构语言之间的编译到非同构语言之间的编译，从过程内优化的验证到过程间优化的验证，囊括了编译器的方方面面；

（3）部分确认器采用辅助证明系统实现，Analyzer 和证明检查器融为一体，更为完整和正确。

最后，翻译确认是在编译时对编译器进行验证，发现编译错误的一种形式化方法，翻译确认的算法相对简单，且对编译器的具体实现不敏感，因此更易于实现对编译器的形式化验证又不影响编译器未来的优化完善，同时又不会对编译器的性能产生影响，能极大地提升编译器的可信性，同时又具备可重用性，因此在编译器验证、测试以及维护中都得到了广泛的应用。但是，翻译确认方法确认的准确率或者说误报率一直是一个非常值得关注的问题，在目前的应用中，翻译确认方法不仅应用于对编译器的验证，同时还可以用于编译器的测试等工作，但它并不能完全取代其他的验证或测试工作，很大一部分原因就是存在误报率，虽然很小，但也不能忽略。

3 总结与展望

安全攸关软件作为安全关键系统的核心部分，其安全性一直以来都引起了人们的充分重视。编译器作为软件开发的必不可少的工具，其是否可信对安全攸关软件的开发有重要影响。如果编译器不可信，则无法保证其所生成代码的可信性，非可信编译器在对程序代码进行编译的过程中，很可能篡改其原本语义，生成不安全的目标代码，因此人们引入了多种方式来对编译器进行形式验证。

本文主要介绍了 4 种方法，一种为软件工程中的软件测试方法；另外三种都为形式化验证技术，分别为模型检验方法、定理证明方法和翻译确认方法。高可信的软件需要严格的测试，并且需要测出编译器引入的错误。但是，编译器引入的错误通常是很难跟踪的，很难被发现。通过测试来保证软件可靠就变得很脆弱，很复杂，如果加上了优化，测试就更显得力不从心。软件测试只能证明软件有错误，不能保证软件没有错误，所以形式化理论和软件验证技术获得了持续的关注。

编译器形式化验证的方法是利用严格的数学逻辑体系对编译过程语义及语言属性的保持进行证明，以便从根本上实现上述规范标准中对编译器的要求限制，确保语义的正确保持。相比其它形式化验证技术（如模型检查和翻译确认），定理证明更符合编译器验证的要求。这主要取决于语言编译过程有比较明确的语义保持要求及清晰的缺陷定义，这样能够更好地发挥定理证明的优势，从而实现以严谨的数学逻辑证明保证编译的高可信度。

形式化方法将对软件可信性的获得和保证有着不可替代的作用。但是，至今，形式化方法在实际的高可信软件的开发中仍不多见，基本处于实验室的试验阶段，并且，其使用者多是专家型用户。软件开发，包括高可信软件的开发，仍以非形式化软件开发方法为主流。因此，如何以一种工程化的方法研究和应用形式化方法和技术是高可信软件工程的发展方向。

程序设计语言的变革历来是计算机科学中里程碑式的进步。从程序设计语言的角度支持高可信软件开发是一个具有挑战性的课题。将程序设计语言、编译技术和形式化验证结合建立 Verifying Compiler 被认为是一个巨大的挑战。对于网络嵌入式软件，支持高可信软件的面向 Agent 的程序设计语言将是一个趋势，同时程序设计方法学基础层面上的成果（例如形式验证、运行时验证）将更多地呈现在语言设施的设计和实现上，在新的软件开发平台和运行平台上显现出来。

4 主要参考文献

[1] RTCA DO-178B. "Software considerations in airborne systems and equipment certification." Washington, DC: Radio Technical Commission for Aeronautics, Inc (RTCA), 1992.

[2] SC-205, "Software Considerations in Airborne Systems and Equipment Certification" (DO-178C), RTCA, Inc. December 2011.

[3] Kossatchev A S, Posypkin M A. Survey of compiler testing methods[J]. Programming and Computer Software, 2005, 31(1): 10-19.

[4] Clarke E M, Grumber O, Peled D. Model Checking[M]. Cambridge: MIT Press, 1999.

[5] Baier C, Katoen J P. Principles of Model Checking[M]. Cambridge: MIT Press, 2008.

[6] Lin Huimin, Zhang Wenhui. Model checking: Theories, techniques and applications[J]. Acta Electronica Sinica, 2002, 30(12): 1907-1912.

[7] Li Y, Li Y, Ma Z. Computation tree logic model checking based on possibility measures[J]. Fuzzy Sets and Systems, 2015, 262: 44-59.

[8] Rozier K Y. Linear temporal logic symbolic model checking[J]. Computer Science Review, 2011, 5(2): 163-203.

[9] Stirling C, Walker D. Local model checking in the modal mu-calculus[J]. Theoretical Computer Science, 1991, 89(1): 161-177.

[10] Emerson E A. Model checking and the mu-calculus[J]. DIMACS series in discrete mathematics, 1997, 31: 185-214.

[11] Clarke E M. My 27-year quest to overcome the state explosion problem[C]//Logic In Computer Science, 2009. LICS'09. 24th Annual IEEE Symposium on. IEEE, 2009: 3-3.

[12] Cousot P, Cousot R, Mauborgne L. Theories, solvers and static analysis by abstract interpretation[J]. Journal of the ACM (JACM), 2012, 59(6): 1-56.

[13] Li Mengjun, Li Zhoujun, Chen Huowang. Program verification techniques based on abstract interpretation theory[J]. Journal of Software, 2008, 19(1): 17-26.

[14] Clarke E, Grumberg O, Jha S, et al. Counterexample-guided abstraction refinement for symbolic model checking[J]. Journal of the ACM, 2003, 50(5): 752-794.

[15] Hoare C A R. An axiomatic basis for computer programming[J]. Communications of the ACM, 1969, 12: 576-580.

[16] Floyd R W. Assigning meanings to programs[C]//Proceedings of Symposium on Applied Mathematics, 1967, 19-31.

[17] Magill S, Nanevski A, Clarke E, et al. Inferring invariants in separation logic for

imperative list-processing programs[C]//Proc of the 3rd Workshop on Semantics, Program Analysis and Computing Environments for Memory Management (SPACE 2006), Charleston, 2006: 47-60.

[18] Ireland A. Towards automatic assertion refinement for separation logic[C]//Proc of the ASE 2006. [S.l.]: IEEE Computer Society, 2006: 309-312.

[19] Owre S, Rushby J, Shankar N. PVS specification and verification system[J]. URL: pvs. csl. sri. com, 2001.

[20] COQ DEVELOPMENT TEAM. The Coq proof assistant reference manual[J]. TypiCal Project, 2012.

[21] Wenzel M, Paulson L C, Nipkow T. The isabelle framework[M]//Theorem Proving in Higher Order Logics. Springer Berlin Heidelberg, 2008: 33-38.

[22] Charles N. Fischer, Ronald K. Cytron, Richard J. LeBlanc, Jr. 编译器构造. 北京: 清华大学出版社, 2012: 7~10.

[23] McCarthy J, Painter J. Correctness of a compiler for arithmetical expressions[C]//Mathematical Aspects of Computer Science 19: Proc of Symposia in Applied Mathematics, 1967: 33−41.

[24] Thatcher J W, Wagner E G, Wright J B. More on advice on structuring compilers and proving them correct[J]. Theoretical Computer Science, 1981, 15(3): 223−249.

[25] Stephenson K. Compiler correctness using algebraic operational semantics, CSR 1-97[R/OL]. University of Wales Swansea, 1997. http://www-compsci.swan.ac.uk/reports/yr1997/CSR1-97.pdf.

[26] Pnueli A, Siegel M, Singerman E. Translation Validation[C]//Proc. 4th Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems. 1998:151-166.

[27] Fang Yi. Translatin of Optimizing Compilers[D]. New York University, 2005.

[28] Yang X, Chen Y, Eide E, et al. Finding and understanding bugs in C

compilers[C]//ACM SIGPLAN Notices. ACM, 2011, 46(6): 283-294.

[29] Leroy X. Formal verification of a realistic compiler[M]//Communications of the ACM, 2009.

[30] Leroy X. A formally verified compiler back-end[J]. Journal of Automated Reasoning, 2009, 43(4): 363-446.

[31] Leroy X. Mechanized semantics for compiler verification[M]//Certified Programs and Proofs. Springer Berlin Heidelberg, 2012: 4-6.

[32]王蕾, 石刚, 董渊, 等. 一个 C 语言安全子集的可信编译器[J]. 计算机科学, 2013, 40(9): 30-34.

[33]何炎祥, 吴伟, 刘陶, 等. 可信编译理论及其核心实现技术: 研究综述[J]. 计算机科学与探索, 2011, 5(1): 1-22.