北京航空航天大学计算机学院

硕士学位论文开题报告

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

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 课题来源

民机专项“符合 DO-178B/C 的 A 级机载软件开发与认证技术研究”

2 论文选题的背景与意义

随着计算机应用技术的飞速发展，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，通过在 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，是实现软件从设计到能在硬件上运行的桥梁，它是否安全可靠？是进行软件开发所面临的重要难题。如果编译器不可靠，则无法保证其所生成代码的安全，非安全的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。特别在安全关键领

域中，如航天、核工业等，编译器的安全可靠有着至关重要的作用，由编译所引入的错误可能会带来灾难性的后果，如 2010 年，由于软件故障，导致美国海军的一架无人直升机发生系统故障，闯进华盛顿上空的禁飞区，险遭击落。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进行充分验证。

实践中在安全相关系统中使用 C 语言时，必须对语言的使用加以限制，避免那些确实可以产生问题或编译器支持的不完善的地方，直到它是可以安全应用的。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，确保编译后代码的安全可靠。

3 国内外研究现状及发展动态

3.1 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，是可信软件开发中一个行之有效的、必不可少的、客观地评估软件可信性的方法。在高可信软件开发中, 软件测试的开销往往大于 50%。

编译器测试常用的有两种策略：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[3]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。

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。

3.2 模型检验方法

3.2.1 基本思想

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。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）。

模态/时序逻辑是模型检测的基础。常用的模态逻辑有三种，分别是：计算树逻辑[7]、线性时序逻辑[8]和命题 μ 演算逻辑[9~10]。

3.2.2 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]。

3.2.3 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：

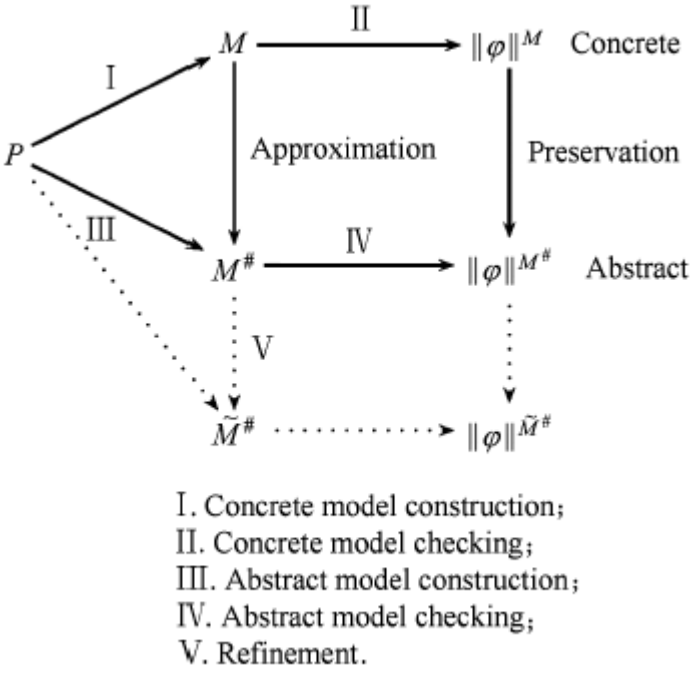

Fig.1 Overview of abstraction in software model Checking.

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）。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]。

3.3 定理证明方法

3.3.1 基本现状

编译器本质上是一个符转换程序，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。人们在几十年前就开展了编译器形式化验证的工作。1967 年 John McCarthy 和 James Painter 首先考察了编译器正确性的问题，对一个简单的从数学表达式到机器语言编译算法的正确性进行了证明。Dave 于 2003 年的综述列举了从 1967 年到 2003 年的大部分相关工作，包含从针对简单语言的单遍编译器到较成熟的代码优化编译器等工作。

近年来，比较具有代表性的是 Xavier Leroy[29~30]带领的 CompCert 项目组所做的工作，他们首次完成了对一个完整且实际的编译过程的正确性形式化验证，整个证明过程完全形式化且是机器自动生成的。为了支持自动的形式化证明，CompCert 先采用辅助定理证明工具 Coq Assistant(http://coq.inria.fr/)对编译过程进行重新构造，此编译过程完成了从一种结构化的函数式语言 Clight 到汇编代码 PowerPC 的转换，整个过程由八种不同的中间语言之间的转换构成，然后使用 Coq Assistant 对整个编译过程的正确性即语义可保持性进行证明。目前 CompCert 编译器只能实现对一个 C 语言子集的编译，还不能完全覆盖所有 C 语言元素，且后端优化程度还比较低，项目也正在进一步研究中。

2011 年，Yang[28]等人在关于 Csmith 的研究工作中对主流的 C 编译器进行测试，共报告了 325 个 bugs，其中包括 Intel CC，GCC 和 LLVM 等。在所比较的 11 种开源或商用的 C 编译器中，CompCert 表现较为突出，在 6 个 CPU 年中，其中间转换过程没有发现 bugs。

3.3.2 逻辑方法

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。

对程序的定理证明研究开始于 20 世纪 60 年代 Hoare[15]和 Floyd[16]发表的论文。Hoare 在他的论文里提出一个形式系统，称作霍尔逻辑。在霍尔逻辑中存在一组证明规则，称为霍尔规则。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。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。

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，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：

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

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。

3.3.3 定理证明工具

定理证明目前比较好的方式是使用编程和证明统一的工具，如 PVS、Coq 和

Isabelle 等。

3.3.3.1 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。

3.3.3.2 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。

3.3.3.3 Isabelle

Isabelle[21]是一个通用的定理证明助手。它允许的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。

3.4 翻译确认方法

3.4.1 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[22]的成立。

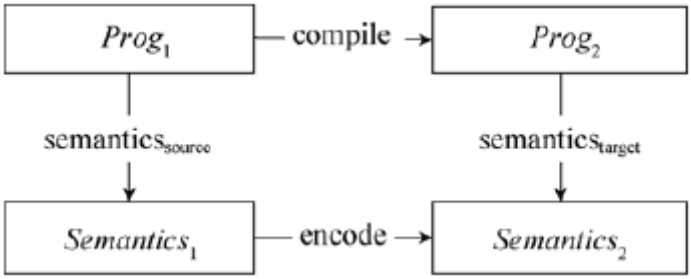

Fig.2 Compiler correctness diagram

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：

$$encode(semanticssource(P)) = semanticstarget(compile(P))$$

程序 P 可以使用不同的语义[23]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。

3.4.2 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。使用翻译确认方法不是直接验证翻译程序，而是用统一的语义框架为某一翻译过程的源和目标代码建模，两个模型之间定义一种求精(refining)等价关系，需要设计一个确认器（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。

3.4.3 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 3 所示：

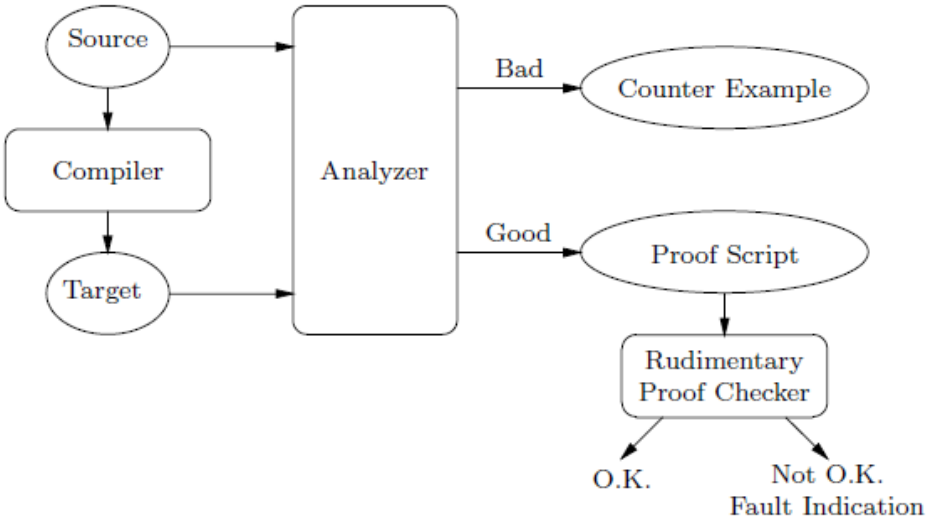

Fig.3 Translation validation process

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。

4 论文的研究内容及拟采取的技术方案

4.1 研究目标

本课题的研究目标是构建一个带有形式验证功能的编译工具。该工具不仅能完成基本的编译功能，如词法分析、语法分析等，还可以检查源代码是否符合安全 C 标准；能够正确的生成目标代码，使用基于语义的形式验证方法来保证编译

过程的正确性；能够实时反馈编译和验证过程的信息；能够从源代码追溯到目标代码，实现编译过程的完整性、一致性和准确性的需求。

4.2 主要研究内容

为了实现上述研究目标，本文拟进行如下几个方面的研究：

（1） 研究如何在编译的初始阶段，即词法分析和语法分析中加入对安全 C 约束规则的检验过程，使得不符合安全 C 标准的源代码在初始阶段就能被识别出，提高编译的效率。

（2） 研究如何获得源代码文法单元的语义和编译后目标代码段语义的方法。

（3） 研究一种基于语义的形式验证方法，能确认生成的目标代码的语义和源代码的语义是否保持一致。若一致，则说明编译过程正确，生成的目标代码安全可靠；反之，则有误。

（4） 研究如何构建一个专用公理集，其中包含了安全 C 所有文法单元的语义和目标代码中指令的语义。

4.3 拟采取的技术方案

4.3.1 形式化验证工具系统框架架构

本课题要实现一个编译验证工具，其系统框架架构如图 4 所示：

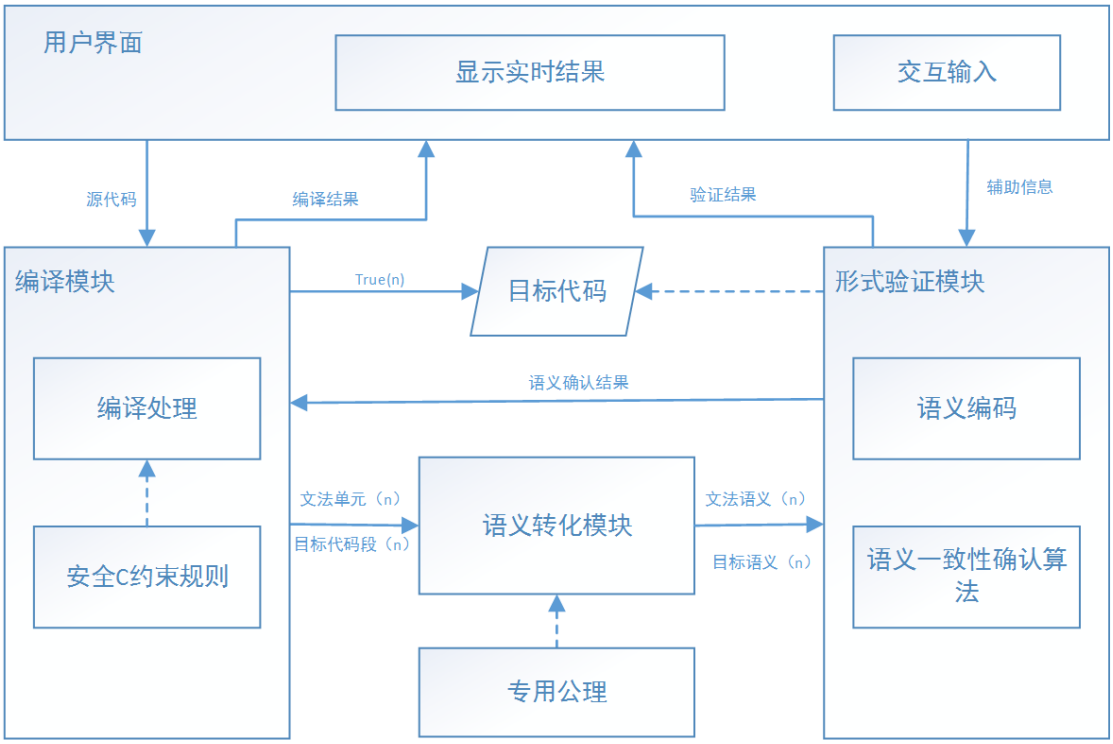

Fig.4 Formal verification tools framework

- 编译模块完成基本的编译处理过程和安全 C 约束规则的检验，同时根据形式验证模块给出的结果决定目标代码的生成；
- 语义转化模块分别求出每个文法单元和和对应的目标代码段的语义；
- 专用公理集提供不同指令集（如 MIPS、PowerPC）的指称语义；
- 形式验证模块实现对文法单元语义和目标代码段语义一致性的确认，但在调用确认算法前，需要使用语义编码消除二者格式和表达上的差异；
- 用户界面辅助用户进行形式化验证。

4.3.2 专用公理集

- ✓ 开发文法单元和目标代码中指令的语义集，并把其作为专用公理集提供给语义转化模块使用；
- ✓ 语义转化模块根据输入的文法单元或目标代码段从专用公理集中搜索其对

应的语义，处理后输出；

4.3.3 基于语义的形式验证方法

对编译器进行形式验证可以通过验证编译前后源代码和目标代码的语义是否保持一致来实现，但若直接对整个源代码和目标代码进行验证，则太过于复杂和困难，所以本课题拟提出一种新的验证方法来简化这一过程。

首先，把 C 程序的源代码通过编译处理，从中识别出多条具有完整语义的 C 文法单元，编译出每个文法单元对应的目标代码段。然后把文法单元和目标代码段通过专用公理的应用分别转化为指称语义的形式，基于得到的指称语义进行化简和推理，获得最简指称语义的形式。再把文法单元的指称语义和目标代码段的指称语义进行编码，消除二者格式和表达上的差异。最后，通过确认算法判断文法单元的指称语义和目标代码段的指称语义是否一致。

整个验证过程中需要对多条文法单元语义进行验证，但并不需要等所有的文法单元都验证完毕才给出最后的验证结果。只要有一条文法单元的语义验证失败，则可以判断编译过程有误，可以停止验证过程；反之，只有所有的文法单元的语义验证都正确，才可判断编译过程正确。

4.3.4 用户交互界面的开发

- ✓ 加入实时的提示功能，向用户显示程序的运行进度；
- ✓ 开发用户交互接口，使用户能做出一些决策，辅助证明过程；
- ✓ 手动精简证明序列，删除冗余证明项。

5 关键技术与难点

5.1 文法单元

语境表示待证明序列中每一个证明项所在的环境和上下文，蒙太古语用学中指出相同对象在不同语境中语义不同。从 C 源代码中识别出多条 C 文法单元，如 if 语句、while 语句等，由于其语境确定（主要由局部变量和全局变量等组成），则每个 C 文法单元对象的语义就确定，于是便可以基于每个文法单元的语义进

行形式验证。

如何设计从 C 源代码中识别出多个 C 文法单元及生成其对应的目标代码段的算法，这是课题研究的难点。

5.2 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）。

课题中我们将用指称语义的形式来分别表示 C 文法单元和目标代码段的语义，难点在于如何正确的获得二者的指称语义。

5.3 语义一致

编译器的任务是将源代码正确的编译为目标代码，而确认源代码与编译生成的目标代码之间是否具有一致的语义是判断编译过程是否正确的有效方法。因此，课题中需要构造一个确认器（validator）来验证源代码的文法单元和目标代码段的语义是否一致。

虽然二者的语义都是用指称语义的形式表示，但两种语言的抽象层次不同，即目标代码段为汇编语言更接近于硬件，因此它们的指称语义表达形式有较大的差异，如何设计一个语义编码方法来消除这种格式和表达上的差异也是当前研究的难点。

6 论文研究计划

- 2015 年 12 月－2016 年 01 月　研究相关资料和技术
- 2016 年 02 月－2016 年 03 月　技术尝试，概要设计
- 2016 年 04 月－2016 年 05 月　详细设计，撰写小论文
- 2016 年 06 月－2016 年 09 月　编码实现，测试分析

- 2016 年 10 月－2016 年 12 月　整理资料，撰写毕业论文

7 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] 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.

[23] 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.

[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, 刘陶, 吴伟. 可信编译器关键技术研究[J]. 计算机工程与科学, 2010, 32(8): 1-6.