软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2020,31(5):1353−1373 [doi: 10.13328/j.cnki.jos.005957]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

嵌入式实时操作系统内核混合代码的自动化验证框架*

郭 建[1,3], 丁继政[3], 朱晓冉[2]

[1](华东师范大学 软件工程学院,上海 200062)

[2](上海市高可信计算重点实验室(华东师范大学),上海 200062)

[3](软硬件协同设计技术与应用教育部工程研究中心(华东师范大学),上海 200062)

通讯作者: 郭建, E-mail: jguo@sei.ecnu.edu.cn

摘 要: “如何构造高可信的软件系统”已成为学术界和工业界的研究热点.操作系统内核作为软件系统的基础组件,其安全可靠是构造高可信软件系统的重要环节.为了确保操作系统内核的安全可靠,将形式化方法引入到操作系统内核验证中,提出了一个自动化验证操作系统内核的框架.该验证框架包括:(1) 分别对 C 语言程序和混合语言程序(C语言和汇编语言)进行验证;(2) 在混合语言程序验证中,为汇编程序建立抽象模型,并将C语言程序和抽象模型粘合形成基于 C 语言验证工具可接收的验证模型;(3) 从规范中提取性质,基于该自动验证工具,对性质完成自动验证;(4) 该框架不限于特定的硬件架构.成功地运用该验证框架对两种不同硬件平台的嵌入式实时操作系统内核 μC/OS-II 进行了验证.结果显示,利用该框架在对两个不同的硬件平台上内核验证时,框架的可重复利用率很高,高达 83.8%,虽然其抽象模型需要根据不同的硬件平台进行重构.在对基于这两种平台的操作系统内核验证中,分别发现了 10 处~12 处缺陷.其中,在 ARM 平台上两处与硬件相关的问题被发现.实验结果表明,该方法对不同硬件平台的同一个操作系统分析验证具有一定的通用性.

关键词: 实时操作系统;VCC;混合程序验证;自动验证;Z3 求解器

中图法分类号: TP311

中文引用格式: 郭建,丁继政,朱晓冉.嵌入式实时操作系统内核混合代码的自动化验证框架.软件学报,2020,31(5):1353−1373. http://www.jos.org.cn/1000-9825/5957.htm

英文引用格式: Guo J, Ding JZ, Zhu XR. Automated verification framework for mixed code in embedded real time operating system kernel. Ruan Jian Xue Bao/Journal of Software, 2020,31(5):1353−1373 (in Chinese). http://www.jos.org.cn/1000-9825/5957.htm

Automated Verification Framework for Mixed Code in Embedded Real Time Operating System Kernel

GUO Jian[1,3], DING Ji-Zheng[3], ZHU Xiao-Ran[2]

[1](Software Engineering Institute, East China Normal University, Shanghai 200062, China)

[2](Shanghai Key Laboratory of Trustworthy Computing (East China Normal University), Shanghai 200062, China)

[3](Software/Hardware Co-design Engineering Research Center (East China Normal University), Ministry of Education, Shanghai 200062, China)

Abstract: “How to construct a trustworthy software system” has become an important research area in academia and industry. As a basic component of the software system, the operating system kernel is an important component of constructing a trustworthy software system.

* 基金项目: 国家自然科学基金(61532019); 上海市重点项目(19511103602)

Foundation item: National Natural Science Foundation of China (61532019); Major Project of Science and Technology Commitment of Shanghai (19511103602)

本文由“系统软件构造与验证技术”专题特约编辑赵永望副教授、刘杨教授、王戟教授推荐.

收稿时间: 2019-09-05; 修改时间: 2019-10-24, 2019-12-24; 采用时间: 2020-02-10; jos 在线出版时间: 2020-04-07

In order to ensure the safety and reliability of an operating system kernel, this study introduces formal method into OS kernel verification, and proposes an automatically verifying framework. The verification framework includes following factors. (1) Separate C language programs and mixed language programs (for example, mixed language programs written by C and assembly language) for verification. (2) In the mixed language program verification, establish an abstract model for the assemble program, and then glue the C language program and the abstract model to form a verification model received by a C language verification tool. (3) Extract properties from the OS specification, and automatically verify properties based on a verification tool. (4) Do not limit to a specific hardware architecture. This study successfully applies the verification framework to verify a commercial real-time operating system kernel μC/OS-II of two different hardware platforms. The results show that when kernels on two different hardware platforms are verified, the reusability of the verification framework is very high, up to 83.8%. Of course, the abstract model needs to be reconstructed according to different hardware. During verification of operating system kernels based on two kinds of hardware, 10~12 defaults are found respectively. Among them, two hardware-related defaults on the ARM platform are discovered. This method has certain versatility for analysis and verification of the same operating system on different hardware architectures.

Key words: real-time operating system; VCC; mix program verification; automatical verification; Z3 solver

软件系统作为信息技术的核心,其安全可靠性在轨道交通、汽车电子、航空航天以及国防安全等安全攸关领域是非常重要的.其中基础软件,如操作系统的安全是整个软件系统的重中之重.近年来,在国内外因基础软件安全问题而导致的恶劣事件屡见不鲜.2017 年上半年的 WannaCry[1,2]勒索病毒全球大爆发,给全球超过 150 个国家、30 万名网络用户带来了超过 80 亿美元的经济损失.该病毒是由不法分子利用操作系统的漏洞,入侵他人 Windows 操作系统用户,将大量重要资料进行加密,使得众多受感染的用户无法正常工作,影响十分恶劣.2018 年 1 月份,国外安全机构曝出的 CPU 安全漏洞:熔断[3]和幽灵[4].该漏洞容易导致敏感数据被泄露,在全球范围内引起了广泛的关注,尤其是对云服务厂商产生了极大的影响[5].

操作系统作为软件系统的核心,其安全性与可靠性是构造高可信软件的最为关键的一步.嵌入式实时操作系统因其具有并发、可抢占的特性以及代码的复杂性给验证工作带来了巨大挑战.

近些年来,学术界有诸多关于将形式化方法运用到操作系统内核验证的研究中,著名的有澳大利亚研究中心 NICTA 的 seL4 项目[6,7],在 2009 年,seL4 项目组使用 Isabelle[8]工具完成了 sel4 的形式化验证,该项目是第一个成功地完成了对操作系统内核的完全形式化验证.由德国联邦教育与研究部资助的 Verisoft 项目[9−11]提出了一个命名为 CVM 的验证框架,通过 CVM 验证框架验证了一个实际运行的操作系统内核.其后续项目 Verisoft-XT 项目[12,13]使用由微软研究院研发的推理证明工具 VCC[14]对 PikeOS[15]进行了形式化的验证.美国耶鲁大学 Zhong Shao 团队运用 Coq[16]工具对其自行开发的 mCertiKOS[17]操作系统进行了端到端的完整形式化验证.Flint 研究组关于操作系统内核的验证[18−20]以及华盛顿大学 Hyperkernel 的项目[21]等在操作系统验证中都做了大量的工作.

国内关于操作系统内核的形式化验证研究也处于较高的水平.中国科技大学的冯新宇团队[22−24]运用 Coq 对商用实时操作系统内核μC/OS-II[25]进行了一个较为完整的形式化验证.除此之外,赵永望团队建立了符合 ARINC653 的分区内核模型,采用了 Event-B 对系统所有的功能和服务进行了形式化建模,该模型实现了 ARINC653 的完整功能和接口.同时,他们利用 Isabelle 工具对操作系统功能规范进行了安全性分析与验证[26].中国科学院软件研究所的周巢尘院士和詹乃军团队成功地开展了对中国铁路控制系统 CTCS-3 的建模和模型层面的验证工作[27].

操作系统内核的验证涉及诸多方面.

- 操作系统内核的系统调用正确性.操作系统内核将具体的硬件结构进行了抽象,只给上层应用开放系统调用接口,上层应用通过系统调用开发各种应用,极大地简化了开发过程.因此,系统调用所提供的功能与其规范一致性是保证上层应用正确性的关键.
- C 语言和汇编语言的混合应用.操作系统在对硬件访问时,为了提高效率,通常利用汇编语言来实现,实现与硬件无关的功能则采用 C 语言编写.因而操作系统内核是由 C 语言和汇编语言互相嵌套来实现的.对由 C 语言和汇编语言构成的混合程序验证是一个难题.

- 程序支持中断和抢占.实时操作系统是一个可抢占式的,能随时被中断服务程序中断的系统,由于系统的可抢占性和中断服务机制使得系统的执行具有不确定性,针对中断机制的验证仍是一个难点.

基于以上 3 个问题,本文提出了一种通用的自动化验证框架,借助于相关工具,使用本验证框架可对由 C 语言和汇编语言实现的实时操作系统内核进行自动化验证,从而实现了 C 和汇编的混合代码验证的目标.与此同时,由于操作系统底层代码与硬件平台相关性,本文所提出的验证框架稍加改动,就可以运用到不同硬件平台上的内核验证中,极大地提高了验证工作的效率.

本文第 1 节详细介绍本文提出的操作系统内核验证框架,首先对所要验证的操作系统内核μC/OS-II进行概述,接着叙述操作系统中的 C 和汇编的混合程序的验证方法.第 2 节介绍对混合语言抽象建模的方法,提出为汇编语言建立抽象模型的方法,针对操作系统内核中的混合程序给出抽象建模的步骤.第 3 节是抽象模型的实现,介绍模型实现语言的语法定义,解释运用该语言对抽象模型的实现方法,说明如何保证抽象模型和 C 代码之间数据的一致性问题.第 4 节描述将该验证框架应用μC/OS-II 内核的验证中,并分析验证结果.第 5 节介绍本验证框架在其他硬件平台上的验证情况,并对比不同平台上的验证结果,对该验证框架做出有效性分析.最后是总结及对未来工作的展望.

1 操作系统内核验证框架

操作系统绝大多数都是采用 C 语言和汇编语言编写的,对操作系统的验证需要分析 C 语言、汇编语言和混合语言的验证.本文以μC/OS-II 为研究对象,提出了一个通用的自动化验证框架[28].该框架如图 1 所示.

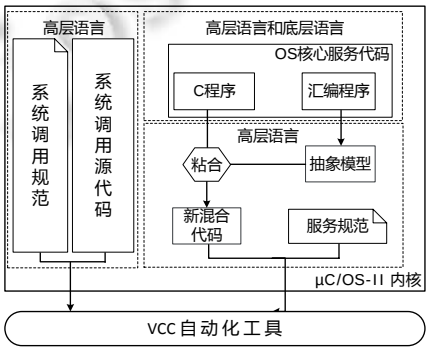

Fig.1 Automatic verification framework of OS kernel

图 1 操作系统内核自动化验证框架

验证工作分为两个部分:一部分对由高层语言(如 C 语言)构成的系统调用进行验证,通过自动化验证工具 VCC 检查系统调用的源代码与其规范的一致性;在另一部分中,对由高层语言和底层语言(如 C 语言和汇编语言)构成的内核服务程序进行验证,通过将汇编语言转换成抽象模型,并实现与 C 语言的粘合,形成符合基于 C 语言的验证工具(如 VCC)能够接收的模型,再利用该工具验证新混合代码.

1.1 μC/OS-II介绍

μC/OS-II[25]是一款开源的基于优先级可抢占式的嵌入式实时操作系统,最早是由美国嵌入式系统专家 Labrosse 在 1992 年发布,因其代码整洁、注释详尽、利于阅读等特点,目前已广泛应用于嵌入式领域,例如航空电子、医疗设备及汽车电子等行业.μC/OS-II 的代码组成结构如图 2 中所示.

μC/OS-II 包括库文件、平台无关的代码和平台有关代码 3 部分.μC/OS-II 绝大部分是由 ANSI C 语言编写的,源代码中只有关于上下文的切换、中断的控制、寄存器的访问等功能是由汇编语言编写.汇编代码总量不超过 200 行,因此其具有极高的可移植性.

μC/OS-II 在不同平台的移植过程中,只需要修改与平台硬件相关的代码文件即可,也就是关于寄存器的访

问、上下文切换以及中断控制等功能的实现代码.系统调用的实现与硬件无关,它包括了任务管理、时钟管理、信号量、互斥量、邮箱、消息队列、事件标志、内存管理等部分.

本文提出的验证框架应用到了飞思卡尔的 ColdFire 和 ARM Cortex-M3 这两种不同硬件平台中,在此对其硬件结构做一简单介绍.

Fig.2 Code constructure of μC/OS-II

图 2 μC/OS-II 代码结构

1.1.1 Freescale MCF 平台

飞思卡尔的 MCF5441x(简称 MCF)系列[29,30]是一个 32 位的处理器内核,由 8 个 32 位的通用数据寄存器(D0~D7)、8 个通用地址寄存器(A0~A7)、1 个 16 位状态寄存器和一些其他的组件构成.

- 这 8 个数据寄存器可以存储相应的数据,它们的数据类型可以是一个字节(8 位)、字(16 位)或者是双字(32 位).
- 8 个地址寄存器可用作软件栈指针、索引寄存器以及基地址寄存器,其中,基地址寄存器只能对字和双字类型进行操作.A7 寄存器是地址寄存器中最为特殊的一个,可用作栈指针.
- 状态寄存器是用于存储处理器状态、中断优先级掩码以及其他控制位.其中,第 8 位~第 10 位属于中断优先级掩码,该掩码用于定义当前中断优先级.中断优先级掩码设定了中断请求优先级的界限,即在有中断请求发生的时候,凡是小于等于当前优先级的所有中断都将予以屏蔽.

1.1.2 ARM Cortex-M3 平台

ARM 架构下的 Cortex-M3(简称 CM3)[31]和 MCF 一样,也属于 32 位处理器内核,有 1 个寄存器组(R0~R15)、中断屏蔽寄存器组(PRIMASK,FAULTMASK,BASEPRI)、中断控制及状态寄存器(ICSR)以及其他组件构成.

在寄存器组(R0~R15)中的 16 个寄存器,其中 13 个(R0~R12)为 32 位通用目的寄存器,其他 3 个有特殊用途.R0~R7 被称作低寄存器(16 位指令只能访问低寄存器),R8~R12 被称作高寄存器.R13(SP)有两个堆栈指针,它们是互斥的,在任意时刻都只能使用其中的 1 个,通过 *PUSH* 和 *POP* 操作实现堆栈的访问.R14 是链接寄存器(LR),用于函数或子程序调用时返回地址的保存;在函数或子程序结束时,程序控制可以通过将 LR 的数值加载到 PC 中,返回到调用程序并继续执行.R15 是程序计数器(PC),指向下一条要取指的指令地址.

PRIMASK 是只使用 1 个位的寄存器,可以关掉除 NMI 和硬 fault 之外所有的异常;FAULTMASK 是只使用 1 个位的寄存器,当其被置位为 1 时,除了 NMI 能够响应,其他的异常都被屏蔽.BASEPRI 寄存器最多使用 9 位,它用于定义被屏蔽优先级的阈值,即当其设为某个值时,所有小于等于此优先级的中断都会被屏蔽(优先级号越高,优先级越低).

中断控制及状态寄存器 ICSR 可以对 NMI、SysTick 定时器以及 PendSV 进行手动悬挂,其中,PendSV 是为系统设备而设的“可悬挂请求”.

PendSV 是一种中断服务,一般设置其为最低的优先级,当其他的所有的中断服务执行完之后才对其进行处理.典型的使用场景是运用于上下文切换中,将 PendSV 进行悬起,以实现延缓执行上下文的切换.

1.2 高层语言程序的验证方法

绝大多数操作系统内核(如本文中的μC/OS-II)的系统调用都是由高层语言实现的,例如 C 语言.在μC/OS-II 的源代码中,系统调用是用 C 语言编写的,被分为 8 个类别.针对系统调用,验证方法分为 3 步:(1) 从系统调用的自然语言规范中提取性质,并将提取的性质采用霍尔逻辑进行描述;(2) 将霍尔逻辑表达式以注释的形式插入到系统调用的源程序相应位置,这中间也涵盖了循环不变式等;(3) 利用验证工具 VCC 对插入后 C 代码进行验证.如果通过验证,则表示该程序满足性质;如果验证失败,则需要分析反例,对验证代码进行修正,重新验证直至验证通过.

1.3 混合语言程序的验证方法

为了提高操作系统的性能及对计算机硬件的访问,操作系统内核中有一部分采用汇编语言编写.混合语言程序的验证大多采用对两种语言的代码分别验证,或者采用将高层语言映射到底层语言上验证.

μC/OS-II 核心服务程序是用 C 语言和汇编语言编写,而汇编代码的执行通常是内嵌到 C 语言中,因此,这里采取了一种自下而上的策略,通过将底层汇编程序抽象,映射到高层语言中,然后在一致的框架中验证.具体的验证流程如图 3 所示.

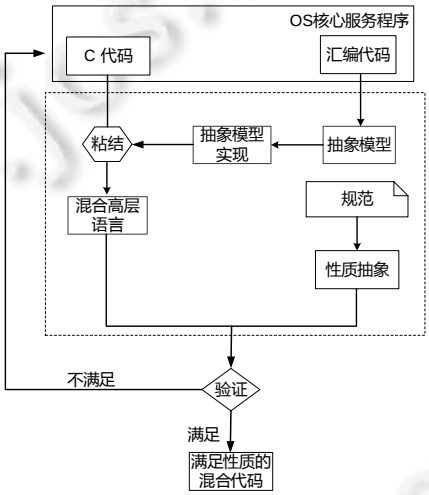

Fig.3 Verification flow of mixed programs

图 3 混合程序的验证流程

首先,对核心服务程序中的汇编语言代码和 C 语言代码进行分离;其次,对汇编语言代码进行抽象和建模,对抽象模型采用高层可验证的语言进行实现;然后,将抽象模型的实现和原混合程序中的 C 代码进行粘合,形成新的混合可验证的高层模型;接着,将内核核心服务的性质从其自然语言规范中提取出来,其中包括 C 语言和汇编语言程序需满足的性质,并采用霍尔逻辑描述;最后,将所要验证的性质插入到新的高层模型中,运用自动化验证工具对其验证.若提示验证错误,则需根据反例,对程序进行分析修正,直至验证通过.若提示验证通过,则表示所验证的程序满足提取的性质.

2 混合语言的抽象建模

操作系统是对资源的管理,不可避免地需要对硬件资源进行访问操作.为提高效率,特别是在上下文切换、中断机制中,通常得使用汇编语言.针对μC/OS-II 内核代码,存在两种混合形态:(1) C 代码内嵌汇编的混合程序,

即在 C 语言编写的程序中调用汇编代码;(2) 汇编内嵌 C 混合程序,即在由汇编语言编写的程序中调用 C 代码.

对于混合代码的验证,有两种方法:一种是将 C 语言编译,然后反汇编,形成汇编代码,在汇编代码的基础上验证;一种是将汇编代码抽象,形成高层模型,能够与 C 语言的代码在基于 C 的验证工具上完成验证.本文将采用第 2 种方法,以便能够使用验证工具 VCC 进行验证.

为了能够对包含汇编代码的混合代码进行验证,需要将汇编程序转换成抽象模型,使其能够在基于 C 程序验证的工具 VCC 中表示,并与 C 程序共同验证.

2.1 汇编语言的抽象建模

为了能够在基于 C 语言程序的验证工具上验证汇编和 C 语言的代码,需要将汇编程序抽象,建立了一个抽象状态机来模拟指令的执行.提取在μC/OS-II 中使用汇编指令,并针对μC/OS-II 中使用的语义对汇编指令抽象,得到一个抽象的汇编语言,这个抽象的汇编语言可以认为是原汇编语言的一个子集.根据抽象的汇编语言,给出了汇编语言的操作语义,然后在操作语义的基础上通过 VCC 建立了他的抽象机模型,以模拟抽象的汇编语言指令的执行.

抽象模型定义为

$$\tau: Prog_{C+ASM} \mapsto Prog_{C+Model}.$$

这里,$Prog_{C+ASM}$ 表示 C 语言和汇编语言的混合程序.混合代码经过转换后,形成了 C 代码和抽象模型混合程序,用 $Prog_{C+Model}$ 符号表示,转换后的程序可以通过基于 C 语言的验证工具 VCC 验证,解决了由 C 语言和汇编语言构成的混合程序无法直接验证的问题.

2.1.1 汇编语言语法

为了对汇编程序进行抽象建模,需要定义所使用的汇编语言的子集.针对不同的操作系统,对应的使用汇编语言的子集也是不同.这里针对μC/OS-II 中所使用的汇编语言来提取子集.由于不同硬件平台的汇编语言是不相同的,但对汇编程序抽象建模的方法始终是不变的.在此,以 MCF 平台上运行的内核代码为例阐述建立抽象模型的方法.下面给出了μC/OS-II 内核代码中所使用到的 MCF 汇编语言子集的 BNF 表示.

$$Instr ::= Opcode\ src, des \mid MOVE.P\ (sp)+, des \mid MOVE.P\ src, -(sp)$$
$$Opcode ::= ORI \mid LEA$$
$$P ::= B \mid W \mid L$$
$$src ::= Dgpr \mid Agpr \mid [X] \mid SR \mid (Val)Agpr \mid Val$$
$$des ::= Dgpr \mid Agpr \mid [X] \mid SR \mid (Val)Agpr$$
$$Var ::= Byte \mid Word \mid Int$$

MCF 汇编语言采用的是三段式结构,从左至右依次为操作符(*Opcode*)、源操作数(*src*)和目的操作数(*des*).语法定义中的操作符涵盖了 MOVE 指令、ORI 指令和 LEA 指令.

在 MCF 汇编语言中,源操作数可以是一个立即数 *Val*、一个内存区域[*X*]、一个数据通用寄存器(*Dgpr*)、一个带字节偏移或不带字节偏移的地址通用寄存器(*Agpr*)、或者一个状态寄存器(*SR*).目的操作数和源操作数类似,但不能为立即数.在 MCF 汇编语言中,源操作数和目的操作数不能同时为内存区域.

MOVE 指令是将源操作数的值赋给目的操作数.MOVE 指令分为 *MOVE.B*、*MOVE.W*、*MOVE.L* 这 3 种形式,那么 *P* 可以是 *B*,*W* 或 *L*,分别用于对字节类型(8 位)、字类型(16 位)和双字类型(32 位)的操作数进行处理.

MCF 汇编指令中没有 pop 指令和 push 指令来对堆栈操作,而是借用 MOVE 指令来实现堆栈的入栈和出栈.MCF 平台上的堆栈增长方向是从高地址到低地址.MCF 平台规定了 A7 可用作硬件堆栈指针 *sp*.将源操作数写为(*sp*)+是出栈操作,即将堆栈顶部值赋给目的操作数,然后将堆栈指针向下移动一个数据类型单元.同理,将目的操作数写为–(*sp*)是压入堆栈,此时,堆栈指针向上移动一个数据单元.

ORI 指令是将两个操作数进行按位或,其中,源操作数应为立即数,目的操作数为数据通用寄存器.

LEA 指令将有效地址加载到其目标地址中.

2.1.2 辅助函数的定义

抽象建模中,由于汇编语言和高级语言的差异性,以及汇编语句会涉及到诸多关于硬件的操作,我们需要定义一些辅助函数用于建立准确的抽象模型.

MCF 平台的汇编程序中对数据的操作都是在固定长度的寄存器中进行的,最终传递给相应的 C 语言程序变量.但 C 语言程序中具有不同类型的变量,其所对应的存储长度也不一致,因此在对汇编程序中数据与 C 语言变量交互的过程进行建模时,势必存在着数据转换步骤.

定义 1(MCF 数据类型定义). 变量 x 可以为自然数和整型,其中,自然数定义为下列 3 种数据类型之一.

$$\mathbb{N}_{32}(x) \overset{\text{def}}{=} 0 \leqslant x < 2^{32},$$

$$\mathbb{N}_{16}(x) \overset{\text{def}}{=} 0 \leqslant x < 2^{16},$$

$$\mathbb{N}_{8}(x) \overset{\text{def}}{=} 0 \leqslant x < 2^{8}.$$

整型可以定义为下列两种数据类型之一.

$$\mathbb{Z}_{32}(x) \overset{\text{def}}{=} 2^{-31} \leqslant x < 2^{31},$$

$$\mathbb{Z}_{8}(x) \overset{\text{def}}{=} 2^{-7} \leqslant x < 2^{7}.$$

定义 2(数据转换). 下面两个函数分别用于将自然数转换为整数以及将整数转换为自然数.

$$\mathbb{N}_n 2\mathbb{Z}_z(x) \overset{\text{def}}{=} \begin{cases} x - 2^z, & 2^{z-1} \leqslant x < 2^n \wedge n \geqslant z \\ x, & \text{其他} \end{cases},$$

$$\mathbb{Z}_z 2\mathbb{N}_n(x) \overset{\text{def}}{=} \begin{cases} x, & (0 \leqslant x < 2^n \wedge z > n) \vee (0 \leqslant x < 2^{z-1} \wedge z \leqslant n) \\ Invalid, & \text{其他} \end{cases}.$$

其中,n 和 z 的值可以是 8,16,32 中的一个.函数的输出如果为 *Invalid*,则表示转换过程中发生了数据溢出.

硬件访问是由汇编程序进行,μC/OS-II 内核的汇编语句中的操作数全部为整数.寄存器中存储的数据用以下的形式描述,其中,*GPR* 表示通用的寄存器:

$$Type(GPR) \overset{\text{def}}{=} \mathbb{Z}_{32}.$$

谓词逻辑用于对寄存器的个数及其能处理的数据类型进行了描述.在下面的谓词逻辑公式中,*GPR*(*A*)代表地址通用寄存器,*GPR*(*D*)代表数据通用寄存器.谓词逻辑的表述如下所示.

$$Type(GPR(A)) \overset{\text{def}}{=} (|A| = 8) \wedge (\forall i < 8) : \mathbb{Z}_{32}(A[i]),$$

$$Type(GPR(D)) \overset{\text{def}}{=} (|D| = 8) \wedge (\forall i < 8) : \mathbb{Z}_{32}(D[i]).$$

定义 3(GPR 的读写). 对通用寄存器组 r 的第 i 个寄存器的读写操作由以下函数定义:

$$GPR\text{-}read(r,i) \overset{\text{def}}{=} \begin{cases} r[i], & 0 \leqslant i < 8 \\ 0, & \text{其他} \end{cases},$$

$$GPR\text{-}write(r,i,x) \overset{\text{def}}{=} \begin{cases} x \Rightarrow r[i], & 0 \leqslant i < 8 \\ invalid, & \text{其他} \end{cases}.$$

2.2 汇编程序的抽象模型

汇编程序抽象模型的建立与硬件的结构相关,主要是 CPU 所涉及到的寄存器.

汇编程序的执行涉及寄存器、堆栈和内存,而对于内存的读写,是通过 C 语言的变量来实现的.因此,为汇编程序建立的抽象模型类似于状态机,涉及到了程序状态、堆栈指针、以及状态转移关系等.

定义 4(抽象模型). 基于汇编语言子集和 MCF 的硬件组件,其抽象模型 MCF_{ASM} 是一个三元组的结构:

$$MCF_{ASM} \overset{\text{def}}{=} (S, sp, \delta).$$

其中,S 代表抽象模型的状态,sp 表示堆栈指针,δ表示状态转移关系.

定义 5(抽象模型的状态). 抽象模型状态是一个三元组,它的定义为

$$S \overset{\text{def}}{=} Dgpr \times Agpr \times SR.$$

抽象模型的状态包括数据通用寄存器 *Dgpr*、地址通用寄存器 *Agpr*、状态寄存器 *SR*.数据和地址通用寄存器的类型都是 32 位的整型,状态寄存器是 16 位的自然数.数据通用寄存器不仅可以用于存储整数和位域字段,还可以用作索引寄存器.地址通用寄存器可以用作堆栈指针,索引寄存器和基地址寄存器.状态寄存器是一个 16 位寄存器,存储着处理器的状态、中断优先级掩码和其他一些控制位.

定义 6(栈操作). 通过对堆栈指针 *sp* 可以进行入栈与出栈的操作,入栈和出栈的行为可以由下面两个函数来描述:

$$MCF_{push}(GPR) \overset{\text{def}}{=} --sp \wedge (sp) \Leftarrow GPR \tag{1}$$

$$MCF_{pop}(GPR) \overset{\text{def}}{=} GPR \Leftarrow (sp) \wedge ++sp \tag{2}$$

其中,

- 定义 6 中的(1)描述了将通用寄存器中的数据压入到堆栈的过程:第 1 步,先将堆栈指针 *sp* 向上移动一个单位;第 2 步,将将通用寄存器中的数据存入到当前堆栈指针 *sp* 所指向的位置.
- 定义 6 中的(2)描述了出栈的过程:第 1 步,将当前堆栈指针 *sp* 所指向的区域位置中所存的值取出,并传放置到通用寄存器中;第 2 步,再将堆栈指针 *sp* 的指向向下移动一个单位.

定义 7(转移关系). 转移关系δ描述了整个汇编程序的执行,其定义如下.

$$\delta \overset{def}{=} S \times sp \times Instr \to S \times sp \tag{3}$$

定义 7 中的表达式(3)描述了抽象模型状态 *S* 和堆栈指针 *sp* 在执行了汇编指令 *Instr* 后发生的变化.表 1 中定义了汇编语言的子集的操作语义,通过操作语义定义抽象模型的转移关系.总共分为 7 种情况.

Table 1 Transition relation of abstract model

表 1 抽象模型的转移关系

编号	转移关系
1	$(s,sp,MOVE.P\ src,des) \xrightarrow{src\neq sp \wedge des\neq sp} (s[des \Leftarrow src],sp)$
2	$(s,sp,MOVE.P\ src,des) \xrightarrow{src=val(sp) \wedge des\neq sp} (s[des \Leftarrow (src+val)],sp)$
3	$(s,sp,MOVE.P\ src,des) \xrightarrow{src\neq sp \wedge des=val(sp)} (s,(sp+val) \Leftarrow src)$
4	$(s,sp,MOVE.P\ (sp)+,des) \xrightarrow{des\neq sp} (s[des \Leftarrow (sp)],sp \Leftarrow sp+size(P))$
5	$(s,sp,MOVE.P\ src,-(sp)) \xrightarrow{src\neq sp} (s,(sp \Leftarrow sp-size(P);(sp) \Leftarrow src))$
6	$(s,sp,ORI\ src,des) \xrightarrow{src=val} (s[des \Leftarrow des \mid src],sp)$
7	$(s,sp,LEA\ src,des) \xrightarrow{src=val(sp) \wedge des=sp} (s,sp \Leftarrow sp+val)$

表 1 列出了内核代码中汇编程序所涉及到的所有指令的操作语义,通过操作语义,给出在抽象模型中状态转移关系.另外,在汇编程序中还应用了跳转指令,即 *JSR* 跳转指令.关于对 *JSR* 指令的含义及抽象,我们将在后文中单独进行介绍.

在编号为 1 的转移中,*MOVE* 指令将源操作数 *src* 的值传递给目的操作数 *des*,其中,*src* 和 *des* 均不能为 *sp*.在编号为 2 的转移中,*src* 是带有偏移值的堆栈指针,且 *des* 不能是堆栈指针,*MOVE* 指令从堆栈的指定地址中读取数据.例如,堆栈结构如图 4 所示:在执行 *MOVE.W* 10(*A*7),*D*1 汇编指令之前,如图 4 中左方所示的堆栈指针指向 *A*7+0,且 *D*1=0;而在执行该指令之后,状态 *s* 已经发生了变化,即 *D*1=5,但堆栈指针并没有移动,如图 4 中右方所示.类似地,在编号为 3 中,*src* 不是堆栈指针,此时的 *des* 是带有偏移值的堆栈指针,将源操作数存储到堆栈的指定地址上,堆栈指针的指向不发生改变.

在编号为 4 的转移中描述了出栈的操作,其中,源操作数为堆栈指针后偏移一个单位,目的操作数不能同时为堆栈指针.*size*(*P*)表示一个数据单位,即当前指令所处理数据的字节长度.这一汇编语句表述了先将堆栈指针

指向的数据传给目的操作数,然后堆栈指针向下偏移 *size*(*P*)个字节长度.同样地,在编号为 5 的转移中描述了入栈操作,其中,源操作数不是堆栈指针,目的操作数是一个堆栈指针且要上移一个单位.与出栈不同的是,入栈过程中先将栈指针向上偏移 *size*(*P*)个字节长度后,再将源操作数的数据存储到当前堆栈指针所指向的地方.

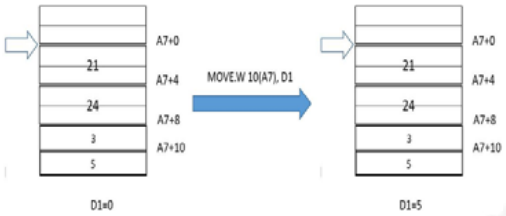

Fig.4 Direction of stack pointer after instruction MOVE

图 4 指令 MOVE 后栈指针指向

编号为 6 的转移中描述了按位逻辑或的操作,源操作数应是一个立即数.执行完逻辑或语句后,将结果存储到目的操作数中.编号为 7 的转移中描述了如何对指针进行调节的操作,其中,*x* 源操作数是一个带有偏移量的堆栈指针,目的操作数为堆栈指针.如图 5 所示的堆栈结构执行 LEA 4(*A*7) *A*7 语句,图 5 中 *A*7+0 表示堆栈指针所指向的地方.在执行语句之前,堆栈指针指向的是数字 21 的正上方;在语句执行之后,堆栈指针向下偏移 4 个字节,堆栈指针指向了数字 21 的正下方,且堆栈中存储的数据没有发生任何变化.注意:在 MCF 硬件平台中,地址寄存器 *A*7 用作硬件堆栈指针.

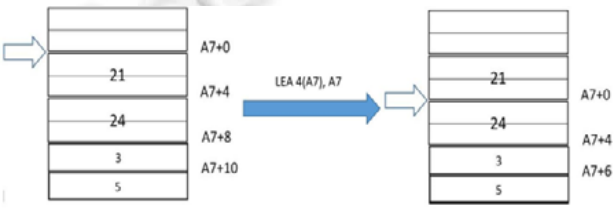

Fig.5 Direction of stack pointer after instruction LEA

图 5 指令 LEA 后栈指针的指向

2.3 汇编语言子集与抽象模型的一致性问题

抽象模型 MCF_{ASM} 的状态是建立汇编语言的可编程模式,对任何一种汇编语言,它的可编程模式包括通用寄存器和状态寄存器.另外,由于堆栈寄存器是一个比较特殊的寄存器,这里把该寄存器单独出来,在后面的上下文切换中可以更清楚地表示.这样,在抽象模型 MCF_{ASM} 中的状态 *S* 和堆栈就建模了汇编语言的可编程模式.

状态转移关系 δ 实际描述了在该编程模式下,汇编语言子集的操作语义.

抽象模型建立了一个抽象机,通过抽象机模拟硬件的汇编语言的执行环境.由于这里用到的汇编语言的子集并不涉及特殊硬件,那么汇编语言子集构成的汇编程序在抽象机上的运行与在具体硬件上的执行是一致的.

2.4 混合语言的建模过程

μC/OS-II 内核的混合代码中的 C 语言程序和汇编语言程序并不是互相独立存在的,而是通过两种代码的交叉协作执行来保证内核的正常运行.我们采用将混合代码中的底层汇编程序抽象到上层来同 C 代码形成新的可验证的高层模型.

在μC/OS-II 内核中,C 语言和汇编程序的交叉协同执行过程如图 6 所示.μC/OS-II 中的混合语言程序分为两种形式:(1) C 内嵌汇编程序;(2) 汇编内嵌 C 程序.在 C 语言的代码中,调用汇编语言函数的程序称为 C 内嵌汇编程序,如图 6 所示的 *task.c* 文件中的 *main*(·)函数调用在汇编语言函数 *OSCtxSw*.在汇编程序中调用 C 语言函数则称之为汇编内嵌 C 程序,例如图 6 中的汇编函数 *OSCtxSw* 中调用了由 C 语言编写的 *OSTaskSwHook* 函

数.我们将分别介绍对这两种形式的混合语言程序建模的步骤.

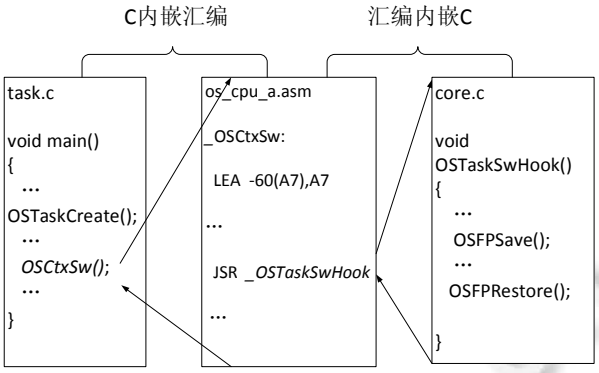

Fig.6 Program calling sequence

图 6 程序调用顺序

2.4.1 C 内嵌汇编程序的建模

在图 6 中,C 语言程序中的 *main*(·)函数调用了汇编函数 *OSCtxSw* 属于 C 内嵌汇编程序.μC/OS-II 内核在创建了多个任务后,一个任务在执行完后或者被另一个任务所抢占,操作系统进行任务切换.在任务运行时,CPU 中的寄存器存储当前任务的相关数据,切换任务后,CPU 的寄存器存储下一个即将执行的任务所需要的数据.为此,执行 *OSCtxSw* 函数进行上下文切换.

图 7 展示了对图 6 中的 C 内嵌汇编程序建模的过程:先从 C 内嵌汇编的混合程序中将汇编程序剥离出来;再对剥离出来的汇编程序建立三元组的抽象模型;然后对汇编程序进行分析,建立其状态转移关系,至此抽象模型建立完成;最后对抽象模型用工具实现,替代原来混合程序中的汇编代码,形成新的可验证的高层语言.

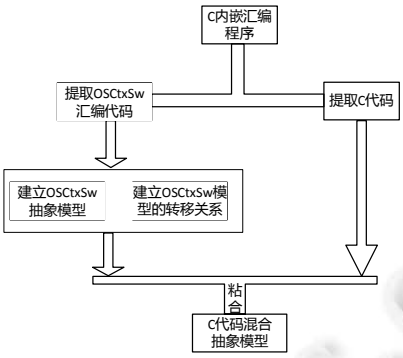

Fig.7 Modeling of C inline assemble program

图 7 C 内嵌汇编程序的建模过程

2.4.2 汇编内嵌 C 程序的建模

在图 6 中,汇编函数 *OSCtxSw* 内部中有一个无条件跳转指令 *JSR*,执行该汇编指令后,将立即跳转到 *JSR* 指令制定的地址,即 C 函数 *OSTaskSwHook* 所在的位置,执行完该 C 函数后,立即返回到原 *JSR* 指令的下一条指令处继续执行汇编程序.实际上,这段代码是汇编函数 *OSCtxSw* 对 C 函数 *OSTaskSwHook* 的调用,即汇编内嵌 C 程序,而 *OSTaskSwHook* 函数是一个 C 内嵌汇编的函数,这里存在着多层的混合调用的关系.

对图 6 中的汇编内嵌 C 程序建模的流程如图 8 中所示.首先,需要对汇编内嵌 C 程序中的 *OSTaskSwHook* 代码剥离出来;其次,由于剥离出来后的 *OSTaskSwHook* 代码是一个 C 内嵌汇编代码,对 *OSTaskSwHook* 代码采用前面提到的“C 内嵌汇编程序建模”的方法,将其转化为 C 代码混合抽象模型;接着,将 C 代码混合抽象模型内

联到 *OSCtxSw* 汇编代码中,形成了汇编内嵌 C 代码混合抽象模型;最后,由于混合代码又被顶层的 C 代码进行调用,因此,结合上层的 C 代码后又可以将其视为 C 内嵌汇编混合程序,再次采用 C 内嵌汇编建模方法,使其最终成为 C 代码混合抽象模型.

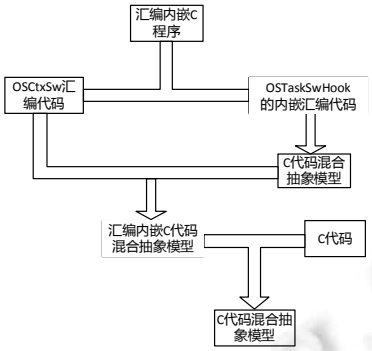

Fig.8 Modeling process of assemble inline C program

图 8 汇编内嵌 C 程序的建模过程

这里定义了汇编程序抽象模型,针对内核代码中两种形态的混合代码,给出了它们抽象建模的方法.

3 抽象模型的实现

有了汇编程序的抽象模型以及混合语言的建模方法,如何将其转换为能够在基于 C 语言的验证工具上实现呢?这里使用自动化验证工具 VCC 对抽象模型实现.

VCC[14,32]是源代码级并发 C 程序的自动化推理验证工具,用于证明 C 语言程序其功能规范的正确性.VCC 工具链允许使用函数合约和数据结构的不变量对并发 C 程序进行模块化验证.函数合约由前置条件和后置条件指定.VCC 是基于注释的系统,即合约和不变量作为注释插入在源代码中,其方式对于常规的非验证的编译器是透明的.验证工作过程分两个步骤.

- 第 1 步,VCC 工具将注释的 C 代码转换为用于验证的 BoogiePL[33]中间语言,然后通过 Boogie[34]工具将其转化为一阶逻辑表达式.其中,BoogiePL 是一种带有嵌入式断言的简单命令式语言,它很容易生成一组一阶逻辑公式,表明程序应该满足性质,并以断言的形式呈现.
- 第 2 步,利用 Z3 求解器对转换后的一阶逻辑表达式进行验证.Z3 求解器会返回两种结果:(1) 一阶逻辑表达式通过验证;(2) Z3 返回一个反例或者提示超时.

3.1 VCC中Ghost语句语法

从 VCC 的视角,一切都被视为对象.每一个对象都是彼此独立的,它们的类型和地址都是唯一定义的,每个对象都有一组字段.这些字段也彼此独立,每个字段都有其类型.对象类别分为两种:一种是具体类型,另一种是 Ghost 类型.

具体类型对象对应运行程序中实际存在的变量或数据结构.具体类型对象拥有固定的存储区域,且任意两个具体类型对象的存储区域都不会发生重叠.具体类型对象对应的是 C 程序的一部分.Ghost 类型对象只存在于验证的过程中,该对象不是 C 程序的一部分.

Ghost 类型对象是验证工程师为了验证而手动添加的辅助代码,添加的这段代码称为 Ghost 代码,通常写为 _(ghost …)形式.其中,Ghost 代码无法直接改变具体类型对象的值,C 代码也不能直接更改 Ghost 类型对象的值.

在 VCC 中,如果定义一个 Ghost 函数,可使用_(def …)语句,这个语句被称为“纯函数”,其对整个程序的执行没有副作用.Ghost 代码和 C 代码的关系类似于汇编代码与 C 代码的关系,因此本文利用 Ghost 语言实现汇编语言的抽象模型.Ghost 语言是一种类 C 的语言,它的语法定义如下.

$$program ::= ghost\ stmt1 \mid def\ \ stmt2 \mid typedef\ \ stmt3$$
$$stmt1 ::= X = Exp\ stmt1 \mid Type * ptr\ stmt1$$
$$stmt2 ::= FunctionName(Type\ Id).Block\ stmt2$$
$$stmt3 ::= struct\ varDeclaration * Id\ stmt3 \mid Type\ X\ \ stmt3$$
$$Exp ::= n \mid X \mid e_0 + e_1 \mid e_0 - e_1 \mid \text{true} \mid NOT\ e \mid e_0 \mid e_1 \mid e_0 \ll n$$

Ghost 定义以 ghost、def、typedef 为关键字的宏注释代码中.

Exp 声明是在 Ghost 中的表达式,可以是一个立即数、一个变量、多个原子表达式的加减或者逻辑运算、移位运算等.

关键字 ghost 的宏注释代码可以定义赋值、变量申明语句.关键字为 def 的宏注释代码用于定义相关的 Ghost 辅助函数.关键字为 typedef 的宏注释用来定义一个新的 Ghost 数据类型或者结构体.

3.2 抽象模型的实现

汇编程序的抽象模型是包含状态 *S*、堆栈指针 *sp* 以及转移关系δ的三元组.程序状态 *S* 用 Ghost 结构体 *MCF_c* 表示.*MCF_c* 结构体中的 3 个元素依次对应了数据寄存器、地址寄存器和状态寄存器,抽象模型的状态 *S* 和堆栈指针 *sp* 的定义如下.

```
1. _(typedef struct MCF_Hardware_struct{
2.     MCF_B32t D[8];
3.     MCF_B32t A[8];
4.     MCF_B16t SR;
5. }MCF_c;)
6. _(ghost MCF_B32t *SP)
```

在μC/OS-II 通过只使用一个硬件指针实现了将即将运行的任务控制块(OSTCB)的内容从内存区域中加载到寄存器中,或者将当前正在运行任务的内容存储到相应的任务控制块中.

*MCF_B*16*t* 类型和 *MCF_B*32*t* 类型是我们自定义类型,它们分别对应了无符号 16 位整型和无符号 32 位整型,通过使用关键字 typedef 定义,如下:

_(typedef *unsigned short MCF_B*16*t*)

_(typedef *unsigned long MCF_B*32*t*)

抽象模型中的状态 *S* 包括数据通用寄存器、地址通用寄存器和状态寄存器,这 3 个成员在实现中分别对应于数组 *D*[8]和 *A*[8]和变量 *SR*,抽象模型的堆栈指针 *sp* 对应于实现中的*SP.在 Ghost 语句中,使用了关键字 ghost 对指针*SP 进行了定义.

抽象模型中的状态转移关系δ表示了抽象模型执行汇编语句前后状态变化.在第 2.2 节中给出了内核汇编代码中所涉及到的 7 种状态转移关系.这 7 种状态转移关系的实现见表 2 中.

Table 2 Code implementation of state transition

表 2 状态转移的代码实现

编号	转移关系的代码实现
1	*des=src*;
2	*des=*(SP+val)*;
3	*(SP+val)=src*;
4	*des=*SP,SP=\old(SP)+1*;
5	*SP=\old(SP)−1,*SP=src*;
6	*des=des|val*
7	*SP=\old(SP)+val*;

表 2 中,*src* 和 *des* 对应 *MCF_c* 结构体里的成员,即数组 *D* 和数组 *A*.表 2 中,名为 *val* 的变量代表了一个立即数.变量 *SP* 表示了一个全局的堆栈指针,定义的堆栈指针 *SP* 指向单位长度为 4 字节的存储区域.Ghost 变量对应着不同的数据类型,不同的数据类型的长度也是不同的,且只能对符合类型的数据进行操作.因此,抽象模

型的实现存在着数据类型的转换过程.在 Ghost 代码实现中,堆栈指针 *SP* 指向的是一个 4 字节的数据,数组成员 *D*[1]也是存储 4 字节的数据.如果要实现汇编指令取出 2 字节的操作,需要进行两次数据转换:第 1 步,先将 *SP* 指向的 4 字节数据转换为 2 节的数据;然后,再将 2 字节的数据转换成 4 字节数据,并将它传递给 *MCF_c* 抽象模型的结构成员 *D*[1]中.另外,*old*(*E*)在 VCC 的环境中用于记录表达式 *E* 在函数入口处的值.在验证工作中,通常会利用*old*(*E*)来计算循环次数,或者用来证明在程序执行之前和之后的状态转移关系的正确性.

下面代码中给出了这种转换的过程.第 2 行、第 3 行定义了两个 Ghost 临时变量,16 位的 *Temp* 整型变量和 32 位的 *Temp*32 整型变量.首先,将堆栈中指定地方的值取出来放到 *Temp*32 变量中;其次,将 Ghost 变量 *Temp*32 传递给 Ghost 辅助变量类型转换函数 *Conv_to_B*16(*MCF_B*32*t*),该函数的输入为一个 32 位的整型数值,输出的结果为一个 16 位的整型数值.在 Ghost 语句中,将 *Conv_to_B*16(*MCF_B*32*t*)的输出值赋给变量 *Temp*;最后,利用 Ghost 辅助变量类型转换函数 *Conv_to_B*32(*MCF_B*32*t*)将 *Temp* 变量的值转化为 32 位整型数值并将该数值赋给 MCF 的 *D*[1]成员.

1. _(**ghost** *MCF_c* **MCF*)
2. _(**ghost** *MCF_B*16*t Temp*)
3. _(**ghost** *MCF_B*16*t Temp*32)
4. ...
5. _(**ghost** *Temp*32=*(*SP*+10))
6. _(**ghost** *Temp*=*Conv_to_B*16(*Temp*32))
7. _(**ghost** *MCF*→*D*[1]=*Conv_to_B*32(*Temp*))
8. ...

3.3 C代码和抽象模型的粘合

在μC/OS-II 的内核代码中,汇编程序和 C 程序分别定义在两种不同类型的文件中.C 语言定义的程序具有高移植性,汇编语言定义的程序可以对内核运行的硬件平台进行访问控制,内核的正常运转离不开这两种语言程序的协作运行.这两种不同语言程序的协作,是通过在各自程序中调用另一语言定义的函数完成的.

在 VCC 设计理念中,Ghost 语言只存在于验证过程中,不能直接影响原程序的执行.本文采用了 Ghost 代码模拟了汇编程序的执行.但在 OS 实际运行过程中,汇编语言程序与 C 语言程序之间存在数据的交换.为了解决抽象模型 Ghost 代码与 C 代码数据交换的问题,提出了在纯函数中添加 VCC 合约语句,见下面的代码.

1. _(**def INT32U** *Assignment*(*MCF_B*32*t p*))
2. _(**ensures** *result*==*p*)
3. ...)

第 1 行的关键字 def 定义了一个 Ghost 纯函数 *Assignment*(·),其返回值为无符号 32 位整数.第 2 行 Ghost 纯函数的入口处添加了后置条件_(*ensures**result*==*p*),该后置条件表述了 Ghost 纯函数 *Assignment*(·)的返回值应为 *p*.

通常在 VCC 中,函数入口处的前置条件和后置条件是函数应该满足的性质.但是在函数体不为空时,直接在验证函数的入口处添加前置条件或后置条件,VCC 认为该性质描述语句是重言式,然后可以通过 Ghost 语句将 *Assignment*(·)的返回值赋给一个具体类型对象.例如,在 C 语言程序中的有一个类型为无符号 32 位的 *StoreValue* 变量,需要将抽象模型中 *D*0 的值赋值给 C 语言的变量 *StoreValue*,此时使用下列语句就可以实现汇编指令与 C 语言代码的通信.

1. _(ghost *StoreValue*=*Assignment*(*MCF*→*D*[0]))

同理,也可以通过 Ghost 纯函数 *Assignment*(·)将具体变量的值传递给 Ghost 类型变量.借助于在定义的 Ghost 纯函数中添加断言的形式,成功地模拟出 C 代码和抽象模型之间的数据通信.这样,抽象模型的实现模拟了汇编指令的执行,并可以与 C 代码一起在 VCC 上运行.

这部分给出了高层实现语言 Ghost 代码的语法定义,通过该 Ghost 语言对抽象模型中三元组的各个元素进

行具体实现,最后介绍了如何将抽象模型的实现,以及抽象模型与内核中 C 代码的粘合.

4 验证μC/OS-II 及其分析

我们运用前面提出的验证框架验证了基于μC/OS-II 的商用实时操作系统内核,包括近 8 000 行的 C 代码和 100 多行的汇编代码(去除了空格和注释),分为系统调用 8 个模块的验证和混合语言实现核心服务程序的验证.

4.1 VCC的基础知识

前面已经提到,VCC 是基于 C 语言代码的验证,其所要验证的性质是以注释的形式给出的,如下面的代码,在函数体内只含有 1 条注释语句.

```
1.  #include ⟨vcc.h⟩
2.  int main(void){
3.      int a,b,c;
4.      if (a≥b)
5.          c=a;
6.      else c=b;
7.      _(assert c≥a)
8.      return 0;
9.  }
```

注意到,在 VCC 验证环境中,每个程序都包含 *vcc.h* 头文件,该头文件给出关于 VCC 相关宏、函数的定义.该程序中第 7 行使用了语句**_(assert** *E*),并通过注释定义的,其中,assert 是标记;*E* 是一个表达式,表示尝试在程序的当前点证明表达式是否成立.如果验证程序成功,则表示该表达式在程序的每次执行中都成立.这和 C 语言中的断言函数不一样,因为 C 语言中断言函数运行成功,表示这次的运行是满足该断言的,并不意味着它将在下次执行时能再次成功.

4.2 系统调用的验证

μC/OS-II 内核中一共 74 个系统调用,分别为任务管理、内存管理、消息队列、信号量、邮箱、互斥量信号量、时间管理和事件标志等 8 大模块.在验证过程中,根据需求提取出每个系统调用需要满足的性质,性质是基于 Hoare 逻辑的形式给出的,并采用 VCC 提供的合约或者断言的形式,以注释的方式插入到源代码中.在系统调用的验证中,所要验证的性质大体可分为下列 3 个大类:类型检查、安全性和边界检查.系统调用的验证性质见表 3.系统调用的 8 个模块列于表 3 的第 1 列中,相对应的每个模块中所验证的系统调用个数列于表的第 2 列,每一个模块所提取的性质列在了表的第 3 列.在 74 个系统调用中添加了共 653 条性质,并完成了验证.

Table 3 Property extraction of system call

表 3 系统调用性质提取

已验证的模块	系统调用个数	性质个数
Task management	12	127
Memory management	6	42
Message queue	10	103
Semaphore	7	63
Mailbox	7	59
Mutex	7	86
Time management	14	101
Flags	11	95
Total	74	653

4.2.1 性质的提取

VCC 通过验证 C 语言程序与其性质的一致性,来证明程序的功能正确性.在μC/OS-II 的需求规范中,面临着

以下3个问题:(1) 自然语言易产生二义性,造成因不同人的理解而导致代码功能实现的差异;(2) 自然语言规范的书写没有一个严格的标准,导致不同的人对需求规范的表述存在着较大的差异;(3) 还没有一种工具能够直接完全的将自然语言规范转换为工具能够识别的性质.因此,在性质的提取过程中,还是人手工抽象的.

VCC验证时,需要从自然语言规范中提取性质并转换,并将其插入到C语言程序的适当位置.这里以创建任务的系统调用 *OSTaskCreate* 的验证为例,给出如何提取性质,并将性质添加到源代码中.

系统调用 *OSTaskCreate*(*task*,*p_arg*,*OS_STK*,*ptos*,*prio*),其接口的自然语言规范表述如下.

(1) *task* 是一个指向任务代码的指针.

(2) *p_arg* 是指向一个可选择数据区域的指针,用于给相应任务初次运行时传递所需的参数.

(3) *ptos* 是指向任务栈栈顶的指针,这个栈是用于在中断时存储本地变量、函数参数、返回地址以及CPU寄存器.当 *OS_STK_GROWTH* 设置为1时,认为指针是从高地址往低地址移动,为0的时候则反向.

(4) *prio* 是任务的优先级,一个优先级最多只能对应一个任务且数字越低表示越高的优先级.

由于 *task* 是指向任务代码的指针,对于该规范这里抽象为指向代码段的指针,即若该指针是指向代码段的某个地址,则认为其是正确的.针对上述4个参数的自然语言规范,转化后的性质描述见表4.

对于所有输入参数的约束,都是属于前置条件,对于它们的验证语句都是放置在_(**requires** …)的前置条件语句中,前置条件语句应插入至函数的入口处.这里的 *code* 是抽象地表示了指当前验证的指针所应指向的代码段区域,*zone* 也是表示了相应的数据区域块.针对不同的硬件架构,代码段区域和数据段区域是不同的,因此在验证不同架构的函数过程中,*code* 和 *zone* 是需要验证人员根据该架构的存储地址的分配进行手动配置的.

Table 4 Parameter property description of API *OSTaskCreate*( )

表4 *OSTaskCreate*(·)参数性质描述

编号	条件	性质描述
1	前置条件	_(logic \bool *is_pointer*(*void*p*,*void*q*)=*p*==*q*) _(requires *is_pointer*(*task*,*code*))
2	前置条件	_(requires *is_pointer*(*p_arg*,*zone*))
3	前置条件	_(logic \bool *grow_dir*(*int x*,*OS_STK*p*)=((*x*==0)⇒*addr*(*p*+1)>*addr*(*p*))‖ ((*x*==1)⇒*addr*(*p*+1)<*addr*(*p*))) _(requires *grow_dir*(*OS_STK_GROWTH*,*ptos*))
4	前置条件	_(logic \bool *pri_uni*(*int pri*,*int*ps*,*unsigned len*)=\forall unsigned *i*; *i*<*len*⇒*pri*!=*ps*[*i*]) _(requires *prio*≥0 && *prio*≤63 && *pri_uni*(*prio*,*ps*,*len*))

除了对入口参数进行性质的提取,还需要在验证中添加相应的性质.如在系统调用 *xQuery*(*arg*1,*arg*2)(查询互斥信号量的当前状态)中,*arg*1 参数是要查询的信号量,*arg*2 参数是用于存放查询到的状态信息,性质添加见下面的代码.

```
1.  INT8U OSMutexQuery(OS_EVENT*pev,OS_MUT_DATA*p_mut_data)
2.  _(requires is_pointer2ECB(pev,Ecb_mut))
3.  _(maintains \arrays_disjoint(p_mut_data→OSEventTbl,
4.  OS_EVENT_TBL_SIZE,pev→OSEventTbl,OS_EVENT_TBL_SIZE))
5.  _(writes \span(p_mut_data))
6.  _(ensures \result==OS_NO_ERR⇒\forall unsigned m;
7.     m<OS_EVENT_TBL_SIZE⇒
8.     p_mut_data→OSEventTbl[m]==pev→OSEventTbl[m])
9.  {
10. ...
11.    for (i=0; i<OS_EVENT_TBL_SIZE; i++)
12.    _(invariant psrc==&pev→OSEventTbl[0]+i)
13.    _(invariant pdest==&p_mut_data→OSEventTbl[0]+i)
```

```
14.   _(invariant \forall unsigned n; n<i⇒
15.     p_mut_data→OSEventTbl[n]==pev→OSEventTbl[n])
16.   {
17.       *pdest++=*psrc++;
18.   }
19.   ...
20. }
```

第 2 行的前置条件要求 *pev* 是一个指向某个已有的互斥量的指针.第 3 行、第 4 行的表达式描述了所要查询的信号量当前的存储区域与用于存放信号量的存储区域在函数执行前后都没有发生重叠.第 5 行表达了存放信号量的区域在本函数中是可写的.第 6 行~第 8 行的式子是一个后置条件,该表达式描述了当函数的返回值为 *OS_NO_ERR* 时,要查询互斥量的相关信息已经成功复制到所要储存的区域中.第 12 行~第 15 行是我们从规范中提取的 3 条循环不变式语句,它们表述了每执行一次循环,当前的源地址和目的地址始终是初始地址加上当前的循环控制量 *i*.对于已经赋过值的元素,其值应该是相等的.*OSMutexQuery*(*arg*1,*arg*2)函数所需要验证的性质位于第 2 行~第 8 行、第 12 行~第 15 行,添加完所有必要的注释代码后,则可以将注释 C 程序放置于 VCC 环境中进行验证.

4.3 核心服务程序的验证

μC/OS-II 内核的核心服务程序是以混合语言(即 C 语言和汇编语言)实现,其中,汇编语言完成有关中断控制、上下文切换以及寄存器读写的操作.为了实现对混合语言程序的验证,将汇编程序转换为抽象建模,并在 VCC 中实现.而对性质的提取、性质的形式化描述与系统调用的方法是相同的.我们在验证中是针对程序是否严格满足所要求的需求规范进行分析验证.如果满足,则表示功能正确;反之,表示软件存在缺陷.在本文中,需求规范亦是需要证明是否完全充分和正确,关于需求规范的验证则不是本文所研究的范围.

这里以函数 *OSIntExit*(·)为例介绍混合语言程序的验证过程,该函数同时包含了第 2.4 节中介绍的两种形式的混合语言程序.

OSIntExit(·)函数用于通知操作系统中断服务程序已经完成了执行,需要退出中断服务程序.它通常与进入中断服务程序 *OSIntEnter*(·)函数成对出现.每次调用 *OSIntEnter*(·)都会对中断嵌套深度变量 *IntNesting* 加一,而 *OSIntExit*(·)是对该变量进行减一操作,其中,中断嵌套的深度 *IntNesting* 是一个全局变量.当最后一层的中断服务执行完后,如果当前就绪队列中有一个任务的优先级高于原被中断任务的优先级,则启动任务调度功能.此时,中断执行完成后将返回到就绪队列中优先级最高的任务,而不一定是返回到最初被中断的任务.

在执行 *OSIntExit*(·)程序的临界区代码段之前,应禁用外部中断以保证其临界区代码的执行不会被打断.这里将状态寄存器作为抽象模型中状态 *S* 的一个成员,它是 *MCF_c* 结构体类型的 *SR* 变量成员.函数 *OSIntExit*(·)的验证代码如下.

```
1.  void OSIntExit(void)
2.  _(writes OSIntNesting && ...)
3.  _(requires OSIntNesting>0)
4.  _(maintains \arrays_disjoint(SP,16,MCF→D,8)
5.     && \wrapped(MCF))
6.  _(ensures OSIntNesting==0 && OSLockNesting==0
7.        ⇒OSCtxSwCtr==\old(OSCtxSwCtr)+1
8.        && OSPrioCur==OSPrioHighRdy)
9.  {
10.     _(unwrap MCF)
11.     _(ghost SaveSR_DisableInt(MCF))
```

```
12.      _(ghost cpu_sr=Assignment(MCF→D[0]))
13.    ...//Dec. 1 of OSIntNesting
14.    ...//If OSIntNesting==0 && OSLockNesting==0
15.    ...//Switch to highest priority task
16.    OSTaskSwHook(·);
17.    _(ghost MCF→D[0]=OSPrioHighRdy)
18.    _(ghost OSPrioCur=Assignment(MCF→D[0]))
19.    _(ghost SP=OSTCBHighRdy→OSTCBStkPtr)
20.          _(ghost MCF→D[0]=*SP++)
21.          ...//Restore registers from stack
22.    ...//Restore SR from D0
23.    _(wrap MCF)
24. }
```

源程序在执行临界区代码之前需要关掉中断,即在第 11 行,通过调用纯函数 *SaveSR_DisableInt*(·)来实现. *SaveSR_DisableInt*(·)函数先将当前状态寄存器的值存储到数据寄存器 *D*0 中,然后更新状态寄存器中的中断控制位以屏蔽中断请求.*OSIntExit*(·)函数从此处开始执行临界区代码.由于抽象模型 MCF 成员的值会被 *SaveSR_DisableInt*(·)修改,使得抽象模型的对象不变性无法保持,因此需要将 MCF 切换到 open 状态才能进行有关成员的修改,即我们在第 10 行中添加了一个 *unwrap* 的子句.在再次包裹 MCF 之前(第 23 行),应该恢复状态寄存器的中断控制位(第 22 行),从而 MCF 的对象不变性又得以保持(第 23 行).

每次执行 *OSIntExit*(·)函数,都会使得 *OSIntNesting* 的值减 1.当中断嵌套深度为 0(即 *OSIntNesting*==0)、调度程序已解锁(即 *OSLockNesting*==0),并且被中断的任务不是就绪队列中优先级最高的任务(第 11 行~第 14 行)时,则发生任务切换.然后将当前任务优先级参数设置为最高可运行优先级(*OSPrioCur=OSPrioHighRdy*),并从新任务的任务栈中的信息恢复至 CPU 寄存器中(第 16 行~第 20 行).第 2 行~第 6 行中列出了性质规范.

4.4 验证结果与分析

验证包括了 13 个 C 语言文件、2 个头文件以及 1 个汇编语言文件,共计 6 446 行 C 语言程序和 100 行的汇编语言程序(除去了代码中所有的注释和空行),添加了 936 行性质验证代码和 205 行的抽象模型的代码实现,抽象模型实现与汇编代码的比例约为 2:1.

μC/OS-II 内核总共验证出 10 个缺陷,分布于 7 个功能函数中.验证出的缺陷可分为两类.

- 类型不匹配.在程序代码中难免会存在一些数据类型的转换,不同数据类型之间的赋值操作需要进行强制转换.例如,在某段程序中的计算赋值语句 *spoke=Match*%*SIZE*,经过验证发现两边数据类型不一致.我们定位到源代码中发现赋值语句左侧的类型是 INT16U,而右侧是 INT32U,这条语句计算的结果可能会导致 *spoke* 变量发生类型溢出.
- 数据溢出.该类缺陷出现的原因在于程序的需求规范缺少对相关的数据的详细约束说明.比如,调度函数 *OS_Sched*(·)中的全局变量 *OSCtxCtr* 是一个上下文切换计数器.*OS_Sched*(·)每执行一次, *OSCtxCtr* 变量加 1,但是整个内核系统代码中没有一处可以对这个全局变量 *OSCtxCtr* 进行重置的操作.因此,该变量会一直累加下去,程序运行到某个时刻,此变量会发生溢出.

我们将以上验证结果反馈给企业的开发人员,得到了他们的认可.

5 验证框架的评估

针对μC/OS-II 内核,上述工作是基于 MCF 平台上的内核代码进行的.由于内核可能运行在不同的硬件平台上,本文提出的验证框架具有通用性,可以运用于不同平台上的操作系统内核的验证.为此,我们验证了 ARM Cortex-M3 平台上的μC/OS-II 内核代码.由于硬件平台的不同,内核的汇编代码部分的不同,发现在 CM3 版本的

内核代码中存在优先级反转问题.

5.1 ARM平台下的验证

为了证明该验证框架的可行性和可扩展性,我们还尝试将其应用于 CM3 平台.由于μC/OS-II 内核具有高可移植性,因而大多数代码与硬件无关,只有少量汇编代码与硬件平台相关.在验证 CM3 平台时,需要抽象一个新的汇编程序的抽象模型.有关 CM3 硬件结构的详细说明,请参考文献[35].同样地,CM3 平台的抽象模型仍是一个三元组,但是三元组的每个成员发生了变化.其中,状态 S 变成了 16 个寄存器;状态寄存器 SR 也发生了变化,不是一个 16 位的状态寄存器,而是一个中断控制相关的特殊寄存器的组合;状态转移关系δ也因汇编语句的变化而发生了变化.尽管抽象模型发生了变化,但是建立抽象模型的方法和 MCF 平台上的方法一样,只是外在的表现形式不同而已.

表 5 将两个平台上的验证结果进行了对比.由于系统调用代码与硬件无关,因此系统调用验证的可重用性达到 100%.核心服务程序的可重用率达到 67.6%,操作系统的总可重用性为 83.8%.根据表 5 中的内容显示,本验证框架具有验证其他任意平台上的μC/OS-II 内核的能力.

Table 5 Verification and comparison of different platforms

表 5 不同平台的验证比较

硬件平台	系统调用	核心服务程序	验证出的缺陷个数
MCF	有缺陷	无缺陷	10
CM3	有缺陷	有缺陷	12
可重用率(%)	100	67.6	–

注:形式化规范的总重用率为 83.8%

5.2 缺陷分析

在系统调用中检测到的缺陷,在不同的平台验证都是一样的,因为 C 语言编写的代码是与平台无关的.但是 CM3 平台发现了两个与优先级反转相关的缺陷,而出现优先级反转的原因是与 CM3 平台的硬件架构有关.

相比较之下,在 MCF 平台上没有出现优先级反转的原因在于,其与 CM3 平台采用了不同方式实现上下文切换.CM3 平台通过 PendSV 实现上下文切换,即可悬挂请求.PendSV 是一个中断驱动程序,为系统级服务的最低优先级,因此它也受硬件中断控制位控制.而 MCF 平台使用陷阱指令实现上下文切换.MCF 陷阱指令不受中断控制位控制,即使中断被禁止也可立即执行上下文切换.

在对任务队列的建模过程中,可以将所有的任务抢占过程分解为为两个任务或者 3 个任务抢占.所以本文采用了两步走的方法来建模任务切换过程:第 1 步,假设当前有一个任务 A 正处于运行的状态,并且任务队列中只有另外一个任务 B 处于就绪状态,就绪任务 B 会去抢占任务 A 的运行,通过验证抢占过程来验证程序的调度性问题;第 2 步,假设当前有一个任务 A 正处于运行的状态,并且任务队列中存在两个任务处于就绪状态,即任务 B 和任务 C,通过验证这 3 个任务同时抢占的过程来验证抢占调度机制的正确性.

图 9 阐述了在 CM3 上发生优先级反转的情况:一开始,高优先级任务 A 处于运行状态,低优先级任务 B 处于就绪状态;然后任务 A 被延时,此时立即调用调度器进行任务切换.在执行任务调度之前,需要禁用中断控制位,并更新相关信息,使得任务 B 在上下文切换后顺利进入到运行状态.

此时运行的状态是 $OSPrioCur=H,OSTCBCur=A,OSPrioHighRdy=H,OSTCBHighRdy=B$,其中,$OSPrioCur$ 为当前运行任务的优先级,$OSTCBCur$ 是当前运行任务的任务控制块,$OSPrioHighRdy$ 是就绪队列中可运行的最高优先级任务的优先级,$OSTCBHighRdy$ 是就绪队列中可运行的最高优先级任务的任务控制块.

但是在禁用中断的情况下,不能立即执行上下文切换.如果在中断禁用期间有新的中断使任务 A 进入到就绪状态,该中断会调用调度器,再次更新信息.此时的运行的状态是 $OSPrioCur=H,OSTCBCur=A,OSPrioHighRdy=L,OSTCBHighRdy=B$.这样,就绪队列中最高优先级任务为 A,但是最高优先级任务的任务控制块为任务 B 的.因此,中断位使能后,并且没有除 $PendSV$ 之外的其他的中断等待执行,则 $PendSV$ 立即执行.执行

的代码是 *OSPrioCur=OSPrioHighRdy,OSTCBCur=OSTCBHighRdy*,也就是 *OSPrioCur=H,OSTCBCur=B*.发现系统运行的是 *B* 任务,但它的优先级变成了 *A* 的,导致 *A* 任务从此以后一直无法得到执行.

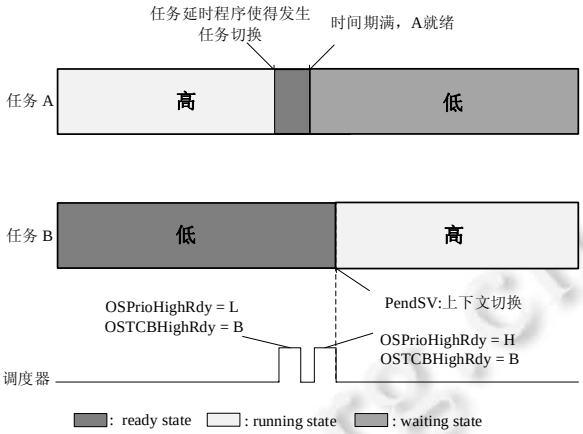

Fig.9 Priority inversion of platform CM3

图 9 CM3 平台优先级反转示例

VCC 的验证原则是将验证合同转换为 BoogiePL 中间语言,最后转换为一阶谓词逻辑,可使用 Z3 求解器进行求解.Z3 求解器是基于 SMT,具有强大而有效的推理和分析功能.有许多公式可以使用 VCC 的推理功能进行验证,例如等式、不等式、加法、减法、常数乘法和布尔连接词.但是涉及非线性的计算、位矢量证明.例如位相关的验证,VCC 不能进行验证.MCF 平台上验证μC/OS-II 时遇到了无法直接对位进行验证的问题,为此我们提出了比较有技巧性地验证位操作的方法,详细见下面代码.

```
1. _(ghost MCF→SR=OS_INITIAL_SR)
2. ...
3. _(ghost MCF→SR=MCF→SR|0x0700)   //Disable interrupt
4. _(assert \forall int x,y;Bv_lemma(x|(y&MCF→SR),8)
5.      ==(Bv_lemma(x,8)|Bv_lemma(y,8)))
```

原汇编程序的运行中,机器状态(SR)会经常需要进行置位和复位的操作,例如,通过修改机器状态的相应控制位来使能和禁止外部中断.而在程序验证过程中,由于 VCC 引擎对相应的位矢量验证存在着缺陷,在验证的过程中需要验证人员采用一定的手段告知引擎相应的位已被修改,如此才能确保 Ghost 代码准确地模拟了汇编程序的执行过程.我们在验证过程中使用 Ghost 语句的宏定义定义了 *Bv_lemma* 的纯函数,用于取出数据中相应的位,然后采用一阶逻辑公式表述位矢量应满足的关系.

6 总 结

本文工作主要的目的是将形式化方法应用于工业界,也就是在项目开发的过程中,采用了一边开发、一边采用形式化方法进行验证的方法.在业界,软件的开发是根据需求规范而来的,最终开发的软件如果严格满足需求,则认为开发的软件不存在问题.因而,我们的验证团队根据判断开发的程序是否严格满足所要求的需求规范:如果满足,则表示功能正确;反之,则表示软件存在缺陷.在项目开发中,需求规范亦需要证明是否完全充分和正确,关于需求规范的验证则不是本文所研究的范围.我们所使用的规范,对程序中每一个变量进行了详细的说明,如果程序中多定义,或者存在规范未描述的状态,我们亦认为代码功能不正确.

本文提出了一个通用的嵌入式操作系统内核自动化验证框架,该验证框架支持对 C 语言程序和 C 语言与汇编语言混合程序的验证.为了检验本框架的可行性,以商用实时操作系统μC/OS-II 的内核作为研究对象,运用

本验证框架,通过验证工具 VCC,完成了该内核的系统调用及混合代码的验证.本框架分别应用于μC/OS-II 的两个不同硬件平台上,以证明本验证框架的有效性和通用性.并且通过验证发现了一些系统缺陷,包括系统调用的缺陷和与硬件平台不同而出现的缺陷.

操作系统内核的高可靠性是非常重要的,今后将把重点放在以下 3 个方面的分析与验证上.

- 规范层面上的形式化验证.规范是用于描述一个系统应该具备的功能,然而对于规范的书写大多采用的是自然语言.一个程序的正确性与其规范息息相关,规范的正确性是程序正确性的前提.
- 内存保护机制和信息流安全.当前的工作涉及对调度机制、中断处理、操作系统内核的资源管理的功能验证.为了实现μC/OS-II 的全面验证,内存保护机制和信息流安全是内核的重要组成部分,对其进行验证分析也是非常重要的.
- 时间属性的分析.由于μC/OS-II 是一个实时操作系统,时间特性也是需要关注的方面.

References:

[1] Ehrenfeld JM. Wannacry, cybersecurity and health information technology: A time to act. Journal of Medical Systems, 2017,41(7): Article No.104.

[2] Mohurle S, Patil M. A brief study of wannacry threat: Ransomware attack 2017. Int'l Journal of Advanced Research in Computer Science, 2017,8(5):1938−1940.

[3] Lipp M, Schwarz M, Gruss D, *et al*. Meltdown. arXiv preprint arXiv:1801.01207, 2018.

[4] Kocher P, Genkin D, Gruss D, *et al*. Spectre attacks: Exploiting speculative execution. arXiv preprint arXiv:1801.01203, 2018.

[5] Simakov NA, Innus MD, Jones MD, *et al*. Effect of meltdown and spectre patches on the performance of HPC applications. arXiv preprint arXiv:1801.04329, 2018.

[6] Klein G, Elphinstone K, Heiser G, *et al*. seL4: Formal verification of an OS kernel. In: Proc. of the Symp. on Operating Systems Principles. ACM, 2009. 207−220.

[7] Klein G, Andronick J, Elphinstone K, *et al*. Comprehensive formal verification of an OS microkernel. ACM Trans. on Computer Systems (TOCS), 2014,32(1):1−70.

[8] Paulson LC. Isabelle: A Generic Theorem Prover. Springer Science & Business Media, 1994.

[9] der Rieden TI, Tsyban A. CVM—A verified framework for microkernel programmers. Electronic Notes in Theoretical Computer Science, 2008,217:151−168.

[10] Baumann C, Beckert B, Blasum H, *et al*. Formal verification of a microkernel used in dependable software systems. In: Proc. of the Int'l Conf. on Computer Safety, Reliability, and Security. Berlin, Heidelberg: Springer-Verlag, 2009. 187−200.

[11] Daum M, Schirmer NW, Schmidt M. Implementation correctness of a real-time operating system. In: Proc. of the 7th IEEE Int'l Conf. on Software Engineering and Formal Methods. IEEE, 2009. 23−32.

[12] Baumann C, Beckert B, Blasum H, *et al*. Lessons learned from microkernel verification—Specification is the new bottleneck. arXiv preprint arXiv:1211.6186, 2012.

[13] Baumann C, Beckert B, Blasum H, *et al*. Better avionics software reliability by code verification. In: Proc. of the Embedded World Conf. Nuremberg, 2009.

[14] Cohen E, Dahlweid M, Hillebrand M, *et al*. VCC: A practical system for verifying concurrent C. In: Proc. of the TPHOLs. 2009. 23−42.

[15] Kaiser R, Wagner S. Evolution of the PikeOS microkernel. In: Proc. of the 1st Int'l Workshop on Microkernels for Embedded Systems. 2007. 50−57.

[16] Chlipala A. Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant. MIT Press, 2013.

[17] Gu R, Koenig J, Ramananandro T, *et al*. Deep specifications and certified abstraction layers. ACM SIGPLAN Notices, 2015,50(1): 595−608.

[18] Costanzo D, Shao Z, Gu R. End-to-end verification of information-flow security for C and assembly programs. ACM SIGPLAN Notices, 2016,51(6):648−664.

[19] Gu R, Shao Z, Chen H, *et al*. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In: Proc. of the 10th USENIX Symp. on Operating Systems Design and Implementation (OSDI 2016). 2016. 653−669.

[20] Chen H, Wu XN, Shao Z, *et al*. Toward compositional verification of interruptible OS kernels and device drivers. ACM SIGPLAN Notices, 2016,51(6):431−447.

[21] Nelson L, Sigurbjarnarson H, Zhang K, *et al*. Hyperkernel: Push-button verification of an OS kernel. In: Proc. of the 26th Symp. on Operating Systems Principles. ACM, 2017. 252−269.

[22] Xu F, Fu M, Feng X, *et al*. A practical verification framework for preemptive OS kernels. In: Proc. of the Int'l Conf. on Computer Aided Verification. Cham: Springer-Verlag, 2016. 59−79.

[23] Xu FW. Design and implementation of a verification framework for preemptive OS kernels [Ph.D. Thesis]. Hefei: University of Science and Technology of China, 2016 (in Chinese with English abstract).

[24] Ma JB, Fu M, Feng XY. Formal verification of the message queue communication mechanism in μC/OS-II. Journal of Chinese Computer Systems, 2016,37(6):1179−1184 (in Chinese with English abstract).

[25] Labrosse JJ. C/OS-II The Real-Time Kernel User's Manual. Rev.2.92.17, 2019. https://doc.micrium.com/pages/viewpage.action?pageId=10753158&preview=/10753158/20644170/100-uC-OS-II-003.pdf

[26] Zhao Y, Sanan D, Zhang F, *et al*. Refinement-based specification and security analysis of separation kernels. IEEE Trans. on Dependable and Secure Computing, 2017,16(1):127−141.

[27] Zou L, Lv JD, Wang SL, *et al*. Verifying Chinese train control system under a combined scenario by theorem proving. In: Proc. of the Working Conf. on Verified Software: Theories, Tools, and Experiments. Berlin, Heidelberg: Springer-Verlag, 2013. 262−280.

[28] Ding JZ. Formal verification and analysis of RTOS kernel [MS. Thesis]. Shanghai: East China Normal University, 2019 (in Chinese with English abstract).

[29] Freescale Semiconductor. Mcf5441x Reference Manual. Rev.4, 2012.

[30] Freescale Semiconductor. Coldfire Family Programmer's Reference Manual. Rev.3 03, 2005.

[31] Yiu J. The Definitive Guide to ARM® Cortex®-M3 and Cortex®-M4 Processors. Newnes, 2013.

[32] Moskal M, Schulte W, Cohen E, Hillebrand MA, *et al*. Verifying C programs: A VCC tutorial. 2012. https://archive.codeplex.com/?p=vcc

[33] DeLine R, Leino KRM. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report, 2005. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2005-70.pdf

[34] Barnett M, Chang BYE, DeLine R, *et al*. Boogie: A modular reusable verifier for object-oriented programs. In: Proc. of the Int'l Symp. on Formal Methods for Components and Objects. Berlin, Heidelberg: Springer-Verlag, 2005. 364−387.

[35] ARM® Cortex® -M3 Processor Technical Reference Manual. Rev.r2p1, 2015. https://static.docs.arm.com/100165/0201/arm_cortexm3_processor_trm_100165_0201_01_en.pdf?_ga=2.256657411.904144181.1588151964-699243147.1588151964

附中文参考文献:

[23] 许峰唯.抢占式操作系统内核验证框架的设计和实现[博士学位论文].合肥:中国科学技术大学,2016.

[24] 马杰波,付明,冯新宇.μC/OS-II 中消息队列通信机制的形式化验证.小型微型计算机系统,2016,37(6):1179−1184.

[28] 丁继政.实时操作系统内核的形式化验证与分析[硕士学位论文].上海:华东师范大学,2019.

郭建(1969－),女,陕西西安人,博士,副教授,CCF 专业会员,主要研究领域为嵌入式实时操作系统,形式化建模与验证,模型检验.

丁继政(1991－),男,硕士,主要研究领域为嵌入式系统建模与验证.

朱晓冉(1991－),女,博士,主要研究领域为形式化验证.