软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2016,27(12):3143−3157 [doi: 10.13328/j.cnki.jos.004851]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

操作系统汇编级形式化设计和验证方法*

钱振江[1,2,3], 黄 皓[1], 宋方敏[1]

[1](南京大学 计算机科学与技术系,江苏 南京 210023)

[2](常熟理工学院 计算机科学与工程学院,江苏 苏州 215500)

[3](King's College London, London WC2R 2LS, UK)

通讯作者: 钱振江, E-mail: tony_h@sina.com

摘 要: 由于系统的巨大规模,操作系统设计和实现的正确性很难用传统的方法进行描述和验证.在汇编层形式化地对系统模块的功能语义进行建模,提出一种汇编级的系统状态模型,作为汇编语言层设计和验证的纽带.通过定义系统状态模型的合法状态和状态转换函数来建立系统状态模型的论域,并以此来描述汇编层的论域.通过验证汇编层的功能模块的正确性来保证汇编语言层设计的正确性,达到对系统功能实现的正确性验证.同时,使用定理证明工具 Isabelle/HOL 来形式化地描述这一系统状态模型,基于这一形式化模型,在 Isabelle/HOL 中验证系统模块的功能语义的正确性.以实现的安全可信 OS(verified secure operating system,简称 VSOS)为例,阐述了所提出的形式化设计和验证方法,说明了这一方法的可行性.

关键词: 操作系统;正确性验证;形式化方法;系统状态模型

中图法分类号: TP316

中文引用格式: 钱振江,黄皓,宋方敏.操作系统汇编级形式化设计和验证方法.软件学报,2016,27(12):3143−3157. http://www.jos.org.cn/1000-9825/4851.htm

英文引用格式: Qian ZJ, Huang H, Song FM. Method of formal design and verification of OS on assembly layer. Ruan Jian Xue Bao/Journal of Software, 2016,27(12):3143−3157 (in Chinese). http://www.jos.org.cn/1000-9825/4851.htm

Method of Formal Design and Verification of OS on Assembly Layer

QIAN Zhen-Jiang[1,2,3], HUANG Hao[1], SONG Fang-Min[1]

[1](Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China)

[2](School of Computer Science and Engineering, Changshu Institute of Technology, Suzhou 215500, China)

[3](King's College London, London WC2R 2LS, UK)

Abstract: The correctness of design and implementation of operating system is difficult to be described and verified with the traditional methods, due to the huge size of the system. In this paper, a model is build for the functionality semantics of the system modules on the assembly layer. A system state model is proposed as the link of the design and verification on the assembly layer. Through defining the legal states and state transition functions, the universe of discourse of the system state model and the assembly layer are constructed. The correctness of functionality modules is verified to ensure the correctness of design and functionality implementation. Within the Isabelle/HOL theorem prover, the system state model is described, and the correctness of functionality semantics of the system modules is

∗ 基金项目: 国家自然科学基金(61402057); 江苏省科技计划自然科学研究项目(BK20140418); 中国博士后科学基金(2015M571737); 江苏省"六大人才高峰"高层次人才项目(2011-DZXX-035); 江苏省高校自然科学研究项目(12KJB520001)

Foundation item: National Natural Science Foundation of China (61402057); the Natural Science Foundation of Jiangsu Province (BK20140418); China Postdoctoral Science Foundation (2015M571737); the "Six Talents Peak" High-Level Personnel Project of Jiangsu Province (2011-DZXX-035); Natural Science Foundation of the Higher Education Institutions of Jiangsu Province of China (12KJB520001)

收稿时间: 2014-06-10; 修改时间: 2014-10-20, 2015-03-27, 2015-04-03; 采用时间: 2015-05-09

verified. Meanwhile, the self-implemented secure trusted operating system (verified secure operating system, VSOS) is used as the example to illustrate the proposed formal method for design and verification, and to show the feasibility of the method.

Key words: operating system; correctness verification; formal method; system state model

操作系统(operating system,简称 OS)作为重要的系统软件,为上层的各种应用程序提供平台服务和安全保障.OS 的正确性是信息安全的重要部分.如何说明和保证 OS 的正确性,一直是学术界和工业界研究的热点.OS 由于其巨大的规模和复杂性,使得其正确性很难用定量的方法来进行描述和说明.即使经过尽可能全面的大规模测试,OS 中的错误还是随着时间的推移不断地出现,这点从目前各种主流商业 OS 不断发布更新补丁这一事实上可见端倪.

对于低安全性的应用环境,使用系统测试的方法可以在一定程度上保证 OS 的正确性.但对于高安全性的应用场合,采用测试的方法是不够的,即使尽可能完整的测试用例,也很难保证 OS 的实现是正确的.

形式化方法是目前公认的对 OS 的正确性进行验证的有效工具.对 OS 的形式化验证可以从两个方面进行,包括高级语言级(如 C 语言)形式化验证和底层汇编语言级形式化验证.很多学者和形式化项目对高级语言级的形式化验证进行了研究[1−3],然而,高级语言代码并非真正在机器上运行的机器代码,即使高级语言层的代码经过了形式化验证,也只能说明系统在高级语言层是可靠的,但无法保证系统在物理机器上实际运行情况的正确性.原因在于高级语言层和机器代码层之间隔了汇编语言层,从高级语言代码到汇编语言代码的转化是通过编译器来实现的.要说明高级语言代码和汇编语言代码的一致性,主要有两个途径:第一是通过验证编译器的正确性来说明;第二是通过对汇编语言代码层进行建模,并对其语义的正确性进行验证来说明.对于第 1 种途径,由于编译器的巨大规模和复杂性,往往很难验证其正确性,很多学者对此进行了尝试[4,5],从所耗费的人力和物力上来说,使得很多的 OS 开发项目望而却步;而第 2 种方法,即直接对汇编语言层进行建模和验证,由于汇编语言过于底层,对其进行形式化验证的难度较大,如何有效地对汇编语言代码进行建模,便于对其语义和功效的正确性进行验证,成为 OS 形式化领域的一大挑战.

同时我们认为,不仅需要使用形式化的方法对系统实现进行验证,对于系统设计的过程,就需要使用严格的形式逻辑来保证设计的正确性,从而最大程度上保证系统的正确性.本文阐述我们对于 OS 形式化设计和验证的方法.本文的创新点在于汇编层形式化地对系统模块的功能语义进行建模,提出一种汇编级的系统状态模型,该模型作为汇编语言层设计和验证的纽带,通过定义系统状态模型的合法状态和状态转换函数来建立系统状态模型的论域,并以此来描述汇编层的论域.通过验证汇编层的功能模块的正确性来保证汇编语言层设计的正确性,从而达到对系统功能实现的正确性验证.我们使用定理证明工具 Isabelle/HOL[6]来形式化地描述该系统状态模型.基于这一形式化模型,我们在 Isabelle/HOL 中验证系统模块的功能语义的正确性.本文以我们实现的安全可信 OS(verified secure operating system,简称 VSOS)为例阐述我们的方法,说明这一形式化方法在对 OS 系统汇编层的建模和语义功效的正确性验证方面的可行性.

本文第 1 节介绍操作系统形式化验证领域的相关工作,并与我们的工作进行比较.第 2 节介绍 VSOS 的整体框架.第 3 节以 VSOS 为例描述我们提出的系统状态模型.第 4 节描述系统状态模型在 Isabelle/HOL 定理证明器环境中的构建方法.第 5 节阐述系统初始化过程汇编级验证方法.第 6 节阐述系统功能模块的汇编级验证方法.第 7 节对我们的工作进行总结,并对未来的工作进行展望.

1 相关工作

1978 年,UCLA 为 PDP-11 机器开发了 UCLA Secure Unix[7].在该系统中,给出了多层规格说明.顶层规格说明描述了内核的权限访问控制模型,抽象层规格说明主要是一些抽象的数据结构,低层规格说明包含了在内核调用接口中要用到的所有变量对象,最低层是内核的 Pascal 代码.UCLA 证明了部分的抽象层次规约的一致性,但是没有证明所有内核层次之间以及与实现的一致性.

Provably Secure Operating System(PSOS)由 SRI 国际组织[8]于 1973 年~1980 年设计,其目的在于提供一个

可证明安全性的通用 OS.PSOS 提出了多级分层抽象的方法,其采用的规约和断言语言 SPECIAL[9]用于精确定义各个层次的模块以及层与层之间的抽象映射.PSOS 只提供了一些简单的实例来说明实现和规约是一致的,没有做到真正的形式化设计和验证.

1992 年,美国支持完成的 LOCK 项目致力于开发达到或超过 A1 级的安全 OS,其整个开发周期的 58%的工作都被用来进行形式化安全策略模型和顶层规范的设计,以及策略与设计之间的一致性证明.1992 年~1993 年间,美国国家安全局(NSA)和安全计算公司(SCC)的研究人员在 LOCK 项目的基础上共同设计和实现了分布式可信 Mach 系统(DTMach).DTMach 项目的后继项目——分布式可信 OS(DTOS)[10]最终形成了 Z 语言版的形式化安全策略模型(FSPM)和形式化顶层规范(FTLS)技术文档.

1995 年,Charlie Landau 组织了 EROS[11]项目,它的形式化验证工作主要考虑地址转换的正确性和内核的内存使用的安全性部分.EROS 的后继项目 Coyotos 由 Hopkins 在 2006 年组织,设计了一种低层次的编程语言 BitC,同时给出了相应的形式语义[12].

VFiasco 项目[13]由德累斯顿技术大学组织,对兼容 L4 的 OS 微内核 Fiasco 进行形式化验证,采用了 SPIN 模型检测的方法来验证 IPC 机制;同时,采用定理辅助证明工具 PVS[14]进行建模和代码验证,对 C++语言进行了部分改进加强编程语言的安全性.2008 年,文献[15]报告了 VFiasco 的工作在 Nova Hypervisor 项目中的延续,提出了 IA32 处理器的模型和内存模型,实现了将 C++代码直接转换为相应 PVS 语意代码的工具.

Robin[16]项目开始于 2006 年,使用 PVS 进行了形式化描述,验证了简单版本的 x86 硬件模型以及 C++的子集语义的正确性.

seL4 项目由澳大利亚国家 ICT 实验室(NICTA)在 2004 年~2006 年实施发起,该项目重点在形式化验证 L4 微内核的 ARM 体系结构版本,其正确性证明采用 Isabelle/HOL 形式化证明辅助工具.Tuch 和 Klein 在文献[1]中介绍了验证的工作,验证的目标是使用 Isabelle/HOL 证明最终的实现符合抽象模型的预期定义.seL4 致力于采用形式化方法验证 OS 的可靠性[17]和完整性[18],并提高实际系统的运行效率[19].

从 20 世纪 90 年代开始,Shao 等人带领的 Flint 项目组在形式化验证方面做了大量的工作[2,20].在安全语言方面,Flint 开发了一种新的编程语言 VeriML[21],其采用的逻辑系统λHOL^{ind}在λHOL逻辑[22]的基础上加入了对数据类型的归纳定义,提供了丰富的形式化描述能力以及类型安全特性.同时,Feng 等人提出的 OCAP[23]开放逻辑框架首次成功将 OS 不同模块的验证逻辑结合起来,形成一个完整的验证系统,并保证验证模型的可扩展性.同时,Flint 项目组研究了并发管理的形式化验证方法[24],并使用分层抽象的方法验证 OS 中的功能模块[25].Flint 项目组和中国科技大学陈意云教授带领的小组组建的可信软件实验室,致力于高可信软件中的形式程序验证技术,研究如何有效地集成形式程序验证和领域专用语言这两种软件技术,形成提高编写健壮软件生产力和提高对它们正确性和安全性的软件开发新方法,并构建开发基于携带证明(PCC)的大型系统软件的基础结构[26−30].

Verisoft 是一个大型的计算机系统设计开发项目,目标在于对整个计算机系统自底向上从硬件层到应用软件层进行普适形式化证明(pervasive formal verification)[3,31].在 2007 年,其后继项目 Verisoft XT 正式启动.在 Verisoft 项目中,普适形式化证明意味着整个项目重点关注的问题并不仅仅是编译器和机器指令模型的正确性,而是整个系统的设计层次都要经过严格的形式化验证,从而组成一个完整的软硬件的验证链[32−34].

目前尚无一款通用 OS 或者 OS 内核可以认为是完全经过形式化验证的,尽管部分小型系统接近 Common Criteria 的 Verified 指标,但仍然需要人工地将系统的实现代码转化为定理证明器的验证代码,或者直接在定理证明器中编写针对系统实现的验证代码,或多或少地存在着非形式化的描述和验证.Verisoft 项目是目前看来比较成功的例子,但目前已终止,其中,VAMOS 内核的验证工作也只是在很小的范围进行.seL4 项目致力于对系统代码进行完全的验证,由于内核结构的复杂性,其验证工作尚未完成.

从 20 世纪 70 年代到现在,OS 的代码从几百行发展到现在的上万行,验证的复杂度也上升了多个数量级,即使是在很大程度上经过形式化设计和验证的项目,如何将系统的实现转化为验证工具的输入,仍然是困扰很多形式化学者的重要问题.同时,系统运行环境和形式化验证之间存在着现实世界的非形式化和逻辑世界的形

式化之间的一道鸿沟.OS 的形式化设计和验证仍然是困难和费时的研究工作,系统实现代码和验证代码量 1:40 以上的工作量,导致形式化验证技术对很多的系统项目来说不得不考虑人力和物力的问题.

同时,具有代表性的 Verisoft,seL4 等形式化工作在对系统进行验证的过程中都是从非常高的层次去验证系统中的功能函数的正确性,并通过保证编译过程的正确性来说明系统整体的正确性.

VSOS 强调在前期设计过程中就采用形式化的方法,以此来保证系统在设计过程的可验证性,同时也是为了力争做到形式化方法和理论对整个 OS 开发过程指导的全程化以及整个 OS 代码的可维、可控,做到系统的完全自主.本文尝试从汇编级对系统建立形式化状态模型,将该模型作为汇编语言层设计和验证的纽带,通过定义系统状态模型的合法状态和状态转换函数来建立系统状态模型的论域.在对系统状态和指令语义的形式化定义的基础上,对系统的安全性(正确性)条件以命题公式的形式来表达,并使用交互式的机械化证明的方法来验证系统的设计和实现是否满足预期的安全性(正确性)条件.验证工作的结果表明,我们的方法是可行的和高效的.如果从系统实现的代码量和验证的代码量的比例的标准来看,我们达到了平均情况下 1:10 的量;同时,我们的验证工作覆盖了整个微内核以及文件系统、进程管理、内存管理等微内核外围模块.

2 VSOS 框架

VSOS 是我们实现的安全操作系统原型,VSOS 内核部分提供基本的系统服务,如中断处理、消息处理和简单的 I/O 服务.其他的功能服务,如文件管理、进程管理、内存管理等,都采用用户态进程的方式来提供.VSOS 的整体框架如图 1 所示.

Fig.1 VSOS architecture
图 1 VSOS 框架

VSOS 采用微内核架构,只有微内核运行于特权级模式,其他的系统功能模块和第三方的驱动程序都运行于用户级,有效地保证了微内核运行环境的隔离性.为此,微内核作为整个系统的可信基.如果对微内核部分进行了严格的验证,就能够保证整个系统的核心的正确性.VSOS 的微内核代码量(C 语言和汇编实现)控制在 10k SLOC(source lines of code)左右,便于对其正确性进行形式化验证.

在 VSOS 的框架中,进程间通信(IPC)采用消息机制实现.从功效来说,消息处理机制是微内核将消息发送进程的消息缓冲区中的消息拷贝到消息接收进程的消息缓冲区中.消息处理需要检查目标进程的合法性,查找内存中的发送和接收缓存.同时,微内核将硬件中断和软件中断转换成消息,硬件中断由硬件产生,软件中断是用户层程序为请求系统服务而向内核传递的唯一途径.

微内核进行进程调度行为,负责进程在就绪态、运行态和阻塞态之间的转换.

由于隔离性,位于微内核外部的进程被禁止执行实际的 I/O 操作,也不能改动系统表以及完成其他操作系统一般都有的功能.为此,需要让微内核为驱动程序和服务器提供一组服务,这些专门的服务是由系统任务进行处理的.系统任务作为独立的进程并被调度,负责接收所有来自驱动程序和服务器的调用请求消息并进行转发.

微内核还负责对系统的硬件中断和系统调用(软件中断)进行处理,包括保存进程上下文(寄存器信息)、中

断处理例程以及恢复并重启进程等行为.

3 VSOS 系统状态模型

本节首先分析构成 VSOS 系统状态模型的元素,如软件/硬件计算单元、系统对象状态和事件.在这些元素的基础上,我们阐述 VSOS 的系统状态模型.

3.1 硬件和软件计算单元

一台安装了 OS 和服务软件的网络服务器,在启动后可以持续地提供约定的服务,这是软硬件系统协同工作的结果.在描述 OS 的抽象模型之前,首先分析系统软/硬件计算单元的基本要素.

现代计算机系统主要的构成部分包括存储器、算术单元(arithmetic unit)和控制单元(control unit)等.这里所讲的存储器部分包括了 CPU 中寄存器、缓存(cache)、传统意义上的主存、设备控制器中寄存器以及硬盘的交换区域.系统执行过程中,算术单元所需的指令序列和数据以及计算的结果都存放在存储器中.我们这里所涉及的存储器概念与传统意义上的计算机组织结构存在一定的差别,这主要是为了便于后续对 OS 抽象模型的描述.能够改变存储器中数据的主体包括 CPU 中的运算器(或 CPU 处理核中的运算器)、采用直接内存访问(direct memory access,简称 DMA)机制的系统中的设备控制器,它们并行地改变存储器中的数据.我们将这些对象称为硬件计算单元.

CPU 中,控制器根据存储器中的数据控制着 CPU 中运算器的运行方式,例如在 Intel 处理器上,当 EFLAG 寄存器的 CPL 位不等于 0 时,运算器不能执行 SETGDT 指令.同时,由于存储器中数据(如 Intel 处理器中的 EFLAGS 寄存器)包含了对于系统中断的掩码信息,为此,控制器根据这些信息控制着系统从到达计算机的中断事件中选择出一组恰当的事件接下来进行处理,从而影响着对下一组执行指令的选择.对于不同的指令,执行所需要的时钟周期数是不同的.在每个时钟周期结束时,有些指令可能完全执行完成,但有些指令则需要在下一个时钟周期继续执行.同时,在每个时钟周期结束时,可能有多个事件到达系统.CPU 中的控制器根据存储器中的当前数据以及恰好执行完指令的硬件计算单元来选择多个事件,在下一个时钟周期进行处理.在这样一个场景下,我们将可以在一个硬件计算单元上运行的一个指令序列称为软件计算单元,例如进程、线程和函数对象等.

假设 U 是一个软件计算单元,U 的只读数据对象集合表示为 $R(U)$,对应地,U 的可修改数据对象集合表示为 $W(U)$.我们将 $R(U)\cup W(U)$称为 U 的一个工作对象集合,表示为 $RW(U)$.由于系统中存在中断机制,软件计算单元 U 在执行过程中可能会被打断,从而进入等待队列进行等待.在 VSOS 的设计过程中,我们利用隔离机制保证了在 U 的等待过程中,其他的软件计算单元不能访问其工作对象集合.为此,我们可以将当前的所有软件计算单元分成两类:一类是正在运行的软件计算单元集合,记为 A_r,这些软件计算单元都在各自对应的硬件计算单元上运行;另一类是正在等待的软件计算单元集合,记为 A_w,这些软件计算单元并不占有硬件计算单元,即,在等待队列中等待运行.所有的软件计算单元的集合表示为 A,即,$A=A_r\cup A_w$.

3.2 系统对象状态

我们使用李未院士的 P 过程记法[35]来描述和定义系统状态.

假设运行计算单元集合为 $A_r=\{a_1,a_2,\ldots,a_t\}$,等待计算单元集合为 $A_w=\{a_{t+1},a_{t+2},\ldots,a_n\}$,$a_i$ 的工作对象集合为 $RW(a_i)=\{x_{ij}|j=1,2,\ldots,t_i\}$,其中,$i=1,2,\ldots,n$,$t_i$ 表示 a_i 的工作对象的个数.对象 x_{ij} 的取值范围表示 V_{ij}.x_{ij} 的初始值为 s_{ij}^0.假设 a_i 有 m_i 条指令,在执行完 k 条指令后,集合 $RW(a_i)$中的对象 x_{ij} 的值为 s_{ij}^k,$k=1,2,\ldots,m_i$.

因此,工作对象集合 $RW(a_i)$的值空间为

$$\prod_{j=1}^{t_i} V_{ij} \tag{1}$$

表示为 $VRW(a_i)$.软件计算单元 a_i 的语义可以表示为映射: $\prod_{j=1}^{t_i} V_{ij} \to \prod_{j=1}^{t_i} V_{ij}$.

我们将笛卡尔积:

$$\prod_{i=1}^{n}\prod_{j=1}^{t_i} V_{ij} \tag{2}$$

称为系统对象状态集合,记为 S_D.

除了公式(2)中的对象外,系统存储器中还存在着多个空闲存储空间片段,这些片段的位置与数量是不断变化的.为了便于分析系统对象状态,我们对包括寄存器、缓存、内存等在内的所有存储单元在表达方式上统一进行编号,这些编号全体属于自然数集合 N,我们将其标记为 N',$N'\subset N$.

每个数据对象都有自己的类型,都对应于一个或者若干个连续的存储单元.例如,32 位的整型变量的存储单元为$\{\&y,\&y+1,\&y+2,\&y+3\}$,一共 4 个编号连续的存储单元,我们用区间$[\&y,\&y+3]$表示这个存储单元的集合,下面称其为 N'的子区间.如果当前系统具有 n 个进程$\{p_i|i=1,2,\dots,n\}$,其中,进程 p_i 的工作对象集合为$\{x_{ij}|j=1,2,\dots,t_i\}$.这些对象对应于 N'的 t_i 个互不相交的子区间$\{N_{ij}|j=1,2,\dots,t_i;i=1,2,\dots,n\}$.除了对象$\{x_{ij}|j=1,2,\dots,t_i\}$所占据的 N'中的这些互不相交的子区间,我们假设仍然剩余 t_0 个互不相交的子区间未被占用,这些子区间表示为$\{N_{0j}|j=1,2,\dots,t_0\}$,对应的数据对象表示为$\{x_{0j}|j=1,2,\dots,t_0\}$.显然,这两部分的子区间,即$\{N_{ij}|j=1,2,\dots,t_i;i=0,1,\dots,n\}$仍然是互不相交的,并且 $\bigcup_{i=0}^{n}\bigcup_{j=1}^{t_i}N_{ij}=N'$.不失一般性,我们假设未被占用的这部分子区间$\{N_{0j}|j=1,2,\dots,t_0\}$的取值为 0,原因在于新分配的数据对象的值都会被初始化成 0.

3.3 事 件

在每一个时钟周期,每个设备都有可能产生硬件中断,这些中断会改变正在运行的软件计算单元集合 A_r;同时,CPU 的异常也会改变 A_r.我们将在时钟周期中产生的软/硬件中断和异常称为当前到达事件,记为 E_a.对应地,之前到达的事件或者等待被处理的事件称为等待事件,记为 E_w.正在被处理的事件称为运行事件,记为 E_r.显然,系统对象状态 S_D、当前到达事件 E_a、等待事件 E_w、正在运行的软件计算单元 A_r 和正在等待的软件计算单元集合 A_w 共同决定了接下来的 E_r,E_w,A_r 和 A_w.

3.4 VSOS系统状态模型的框架

在以上分析的基础上,我们认为,计算机系统的运行过程可以被描述成一系列的计算单元并行地操作着存储器中的数据对象,如图 2 所示.

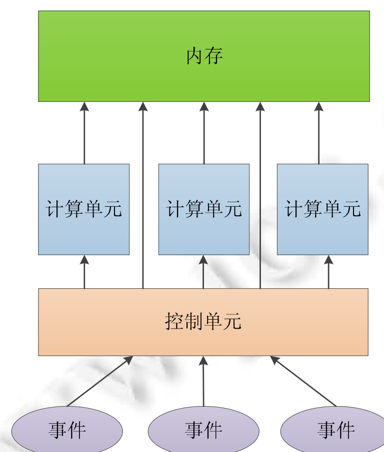

Fig.2 Schematic diagram of operations of computing units on the data objects

图 2 计算单元操作数据对象示意图

从软件的角度来说,我们可以假设硬件计算单元以互斥的方式进行工作.因此,影响系统运行的因素包括数据对象和存储器中的指令序列以及达到事件集合.综上所述,在一定程度上,计算系统和一个状态自动机(state automaton)模型是相符的[36],主要有以下几个理由:接受的事件对象集合对应于状态自动机的字母表(alphabet);系统运行过程中处理的所有事件构成状态自动机接受的一个句子(sentence);在每个时钟周期结束时,存储器的

状态和达到的事件构成状态自动机的当前状态(current state);硬件计算单元对存储器单元的修改动作对应于状态自动机的状态转化(state transition);CPU 执行 halt 指令的状态对应于状态自动机的终止状态(termination state).在此基础上,我们来构建 VSOS 的系统状态模型.

定义 1(VSOS 的系统状态模型). VSOS 的系统状态模型是一个状态自动机,即 $A_{VSOS}=(S,\Sigma,\delta,s_0,\Gamma)$,$S,\Sigma,\delta,s_0$ 和 Γ的定义如下:

(1) 系统状态:$S=(S_D,A_r,A_w,E_r,E_w)$:

a) 对象状态 S_D:系统对象状态;

b) 正在运行的软件计算单元集合 A_r:占有硬件计算单元并正在运行的软件计算单元集合;

c) 等待运行的软件计算单元集合 A_w:正在等待占有硬件计算单元的软件计算单元集合;

d) 运行事件 E_r:被系统选中并正在处理的事件集合;

e) 等待事件 E_w:等待被处理的事件集合;

(2) Σ是系统所接受的各种事件;

(3) 状态转化函数δ:$\delta(s,E_a)=s'$;

(4) s_0 是系统的初始状态;

(5) Γ是系统的终止状态集合,$\Gamma\subseteq S$.当系统到达Γ中的状态时,系统就终止.

4 VSOS 系统状态模型在 Isabelle/HOL 中的构建

上一节分析了构成 VSOS 系统状态模型的基本要素——硬件/软件计算单元、系统对象状态、事件,并在这些基本要素的基础上阐述了 VSOS 系统状态模型.本节首先对我们采用的 Isabelle/HOL 环境做介绍,并在此基础上阐述 VSOS 论域以及在 Isabelle/HOL 中形式化地构建 VSOS 系统状态模型的方法.

4.1 Isabelle/HOL定理证明器环境

Isabelle/HOL 是一款可以对系统规格说明进行描述和验证的定理证明器.Isabelle 是针对逻辑形式化理论的通用系统,Isabelle/HOL 是 Isabelle 对于高阶逻辑(higher-order logic,简称 HOL)的规格说明和实现.在 Isabelle/HOL 环境中,提供了强大的对象描述和规格说明表达的能力,以及对于命题定理的证明方法.

Isabelle/HOL 是一个类型系统,具有多种预定的基本类型描述,如自然数 *nat*、整型 *int*、列表 *list* 等.用户可以采用函数式编程(functional programming)的方式来构建新的类型,如记录类型 *record* 和自构造数据类型 *datatype*,并可以构造各种函数,如原始递归函数 *primrec*、全函数 *fun*,以及各种引理 *lemma* 和定理 *theorem*.在 Isabelle/HOL 环境中,特定域(special domain)以称为理论的方式进行构造,如 *theory*,为此,每个特定域可以认为是类型、定义、函数和定理证明的集合.

在这一节,针对下面的章节将使用到的 Isabelle/HOL 符号和操作进行说明.

自构造数据类型 *datatype* 可以用于构造复合的数据类型,例如,对于系统的运行状态级别在 Isabelle/HOL 中可以定义为:*datatype ring=user_level|root_level*,新数据类型 *ring* 由构造子 *user_level* 和 *root_level* 组成,分别表示用户级模式和内核级模式.

记录类型 *record* 用于构造包含多个成员的复合类型,例如,对于消息类型在 Isabelle/HOL 中可以定义为:

record *message=sender::pid*
receiver::pid
content::data

新类型 *message* 包含 3 个成员:发送者标识 *sender* 和接收者标识 *receiver*,类型为 *pid*;以及消息内容 *content*,类型为 *data*.而类型 *pid* 和 *data* 可以是新的自定义类型.对于 *record* 类型的更新操作,例如,假设 *message* 类型的一个实例为 *msg*,值为(|*sender=p*1,*receiver=p*2,*content=msg_c*1|),对于更新操作形如 *msg*(|*content:=msg_c*2|),表示将 *msg* 的 *content* 成员更新为 *msg_c*2,而其他成员不变.

4.2 VSOS论域

如第 3 节所述,系统对象状态以及状态转化关系可以用数学系统——论域来表示[35,37],记为 M.M 包含 3 个部分:第 1 部分是非空元素集合 m,它可以用来描述各个进程的工作对象的所有可能的取值的集合,显然,$\prod_{i=1}^{n}\prod_{j=1}^{t_i}V_{ij}$ 是 m 的一个子集;第 2 部分是 m 上的非空函数的集合 F,从软件的角度来看,F 中的函数的语义可以表达成这个函数的工作对象集合的取值空间上的一个映射,即,表示一个行为体如何通过工作对象集合的初始值计算出工作对象集合结束时的值;第 3 部分是关于 m 的非空命题的集合 P,用于描述系统的性质或者 F 中函数的条件,如当前的进程的地址空间不相交等性质,或者前置/后置条件等.

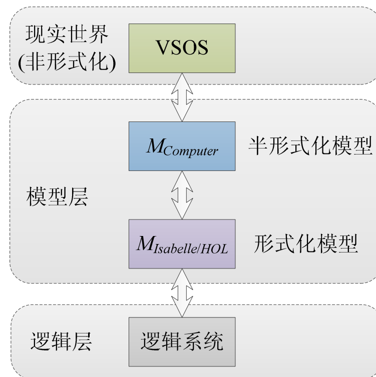

Fig.3 Relationship among VSOS, the universe of discourse and the logic system

图 3 VSOS、论域和逻辑系统的关系

系统状态、软/硬件计算单元的操作以及以命题方式表达的系统的属性组成VSOS的论域,记为 $M_{Computer}$.相应地,在 Isabelle/HOL 中,对应的 VSOS 论域表示为 $M_{Isabelle/HOL}$.根据 VSOS 的设计和实现,我们构造论域 $M_{Isabelle/HOL}$.$M_{Computer}$ 中的数据对象和行为操作语义与 $M_{Isabelle/HOL}$ 中描述的对象和行为定义是一一对应的,为此,我们可认为 $M_{Computer}$ 和 $M_{Isabelle/HOL}$ 是同构的[37],同时,$M_{Computer}$ 中的命题成立当且仅当 $M_{Isabelle/HOL}$ 中的命题成立.

如图 3 所示,我们通过构造 $M_{Isabelle/HOL}$ 和 $M_{Computer}$ 来建立 VSOS 和逻辑系统之间的联系.我们通过构造 VSOS 系统状态模型来设计和实现 VSOS.在 VSOS 实现的基础上,我们描述域 $M_{Computer}$ 以及和 $M_{Computer}$ 同构的 $M_{Isabelle/HOL}$.因此,对于 VSOS 的功能性和安全性的属性,可以映射为 $M_{Isabelle/HOL}$ 中的逻辑命题和公式.我们通过在 Isabelle/HOL 逻辑系统中以推理的方式来验证 VSOS 设计和实现的正确性和一致性.

4.3 在Isabelle/HOL中构建VSOS系统状态模型

VSOS 系统状态模型的基本元素包括系统状态集合 S 和状态转化函数 δ.S 中的每个元素是由所有当前进程的数据对象的值构成的向量.所有事件处理函数和功能性函数构成状态转化函数 δ.每个状态转化函数的语义是根据功能性语义将数据对象的初始值转化为结果值.

下面阐述在 Isabelle/HOL 中构建 VSOS 系统状态模型的方法.

我们从汇编级对 VSOS 系统状态模型进行描述,根据第 3.4 节和第 4.2 节的分析,以定义 1 所描述的 VSOS 系统状态模型为基础,我们在 Isabelle/HOL 中对 VSOS 系统状态模型从汇编级建立抽象模型,我们对该抽象模型定义如下:

state=(*M*,*R*,*IF*,*ZF*,*SF*,*ring*,*GDT*,*IDT*)

M={*nat*→*mem_cell*}*

mem_cell=*null*|*SomeInt int*|*SomeInstr instr*

R={*register*→*int*}*

register=*ds*|*es*|*fs*|*gs*|*di*|*si*|*fp*|*st*|

ax|*bx*|*cx*|*dx*|*bp*|*sp*|*pc*

ring=*user_level*|*root_level*

GDT={*nat*→*int*}*

IDT={*nat*→*int*}*

instr=instr_nop|instr_fault|instr_excp
|movrr register register|movrm register nat
|movmr nat register|movrd register int
|movar register register|movra register register
|add register register|sub register register
|jmp int|jmpPC int|branch register register int
|call int|ret|iret|cli|sti
|pushr register|pushv int
|pushra register|pop register
|swint int|hwint int
|Or register register
|And register register
…

其中,*state* 为系统状态,包括内存状态 *M*、寄存器状态 *R*、条件标志(中断允许标志 *IF*、零标志 *ZF*、符号标志 *SF*)、运行级别 *ring*、全局描述符表状态 *GDT* 和中断描述符表状态 *IDT*.指令类型 *instr* 包括对 Intel X86 架构下的常用指令的描述以及空指令(*instr_nop*)、指令错误(*instr_fault*)和指令异常(*instr_excp*)等.

类型 *instr* 中,各个指令的操作语义见表 1,其中,[*M*(*n*)]表示内存单元 *M*(*n*)中的数据.

Table 1 Schematic table of operation semantics of instructions

表 1 指令的操作语义示例

Instructions	NextState
movrr r_s, r_d	$(M,R\{r_d\sim>R(r_s),pc\sim>R(pc)+1\},IF,ZF,SF,ring,GDT,IDT)$
movrm r_s, n	$(M\{n\sim>SomeInt\ R(r_s)\},R\{pc\sim>R(pc)+1\},IF,ZF,SF,ring,GDT,IDT)$
movmr n, r_d	$(M,R\{r_d\sim>[M(n)],pc\sim>R(pc)+1\},IF,ZF,SF,ring,GDT,IDT)$
movir d, r_d	$(M,R\{r_d\sim>d,pc\sim>R(pc)+1\},IF,ZF,SF,ring,GDT,IDT)$
addrr r_s, r_d	$(M,R\{r_d\sim>R(r_s)+R(r_d),pc\sim>R(pc)+1\},IF,ZF_{check},SF_{check},ring,GDT,IDT)$
addmr n, r_d	$(M,R\{r_d\sim>[M(n)]+R(r_d),pc\sim>R(pc)+1\},IF,ZF_{check},SF_{check},ring,GDT,IDT)$
addrm r_s, n	$(M\{n\sim>SomeInt(R(r_s)+[M(n)])\},R\{pc\sim>R(pc)+1\},IF,ZF_{check},SF_{check},ring,GDT,IDT)$
addir d, r_d	$(M,R\{r_d\sim>d+R(r_d),pc\sim>R(pc)+1\},IF,ZF_{check},SF_{check},ring,GDT,IDT)$
addim d, n	$(M\{n\sim>SomeInt(d+[M(n)])\},R\{pc\sim>R(pc)+1\},IF,ZF_{check},SF_{check},ring,GDT,IDT)$
andrr r_s, r_d	$(M,R\{r_d\sim>R(r_s)\ \text{AND}\ R(r_d),pc\sim>R(pc)+1\},IF,ZF_{check},SF_{check},ring,GDT,IDT)$
andmr n, r_d	$(M,R\{r_d\sim>[M(n)]\ \text{AND}\ R(r_d),pc\sim>R(pc)+1\},IF,ZF_{check},SF_{check},ring,GDT,IDT)$
andrm r_s, n	$(M\{n\sim>SomeInt(R(rs)\ \text{AND}\ [M(n)])\},R\{pc\sim>R(pc)+1\},IF,ZF_{check},SF_{check},ring,GDT,IDT)$
andir d, r_d	$(M,R\{r_d\sim>d\ \text{AND}\ R(r_d),pc\sim>R(pc)+1\},IF,ZF_{check},SF_{check},ring,GDT,IDT)$
andim d, n	$(M\{n\sim>SomeInt(d\ \text{AND}\ [M(n)])\},R\{pc\sim>R(pc)+1\},IF,ZF_{check},SF_{check},ring,GDT,IDT)$
ret	$(M,R\{pc\sim>M(R(sp))\},IF,ZF,SF,ring,GDT,IDT)$

对于表 1 中各个指令的操作语义,在 Isabelle/HOL 中定义如下:

fun *NextS*::“*state*⇒*instr*⇒*state*” **where**

“*NextS s*(*movrr y x*)=*PCinc*(*s*(|*R*:=((*R s*)(*x*:=((*R s*) *y*)))|))”|

…

其中,函数 *PCinc* 表示对 *PC* 寄存器进行递增操作,定义如下:

fun *PCinc*::“*state*⇒*state*” **where**

“*PCinc s*=*s*(|*R*:=((*R s*)(*pc*:=((*R s*) *pc*)+1))|)”

函数 *NextS* 的语义是对指令的单步执行操作,对于指令序列的操作,可以递归地调用 *NextS* 来实现.对于空指令(*instr_nop*),函数 *NextS* 将修改 *PC* 寄存器(递增);对于指令错误(*instr_fault*)和指令异常(*instr_excp*)等情况,函数 *NextS* 分别转入指令出错处理和指令异常处理过程.

www.jos.org.cn

5 系统初始化过程汇编级验证

限于篇幅,在这一节中,以中断描述符表 *IDT* 为例,阐述对系统初始化过程汇编级验证的方法.

在 VSOS 中,建立 *IDT* 如下:

```
struct GateDescriptor {
  uint_32 offset_15_0: 16;
  uint_32 segment: 16;
  uint_32 pad0: 8;
  uint_32 type: 4;
  uint_32 system: 1;
  uint_32 privilege_level: 2;
  uint_32 present: 1;
  uint_32 offset_31_16: 16;
};
struct GateDescriptor idt[];
```

VSOS 使用 *init_segment*()函数以及 *set_segment*()函数来初始化全局描述符表 *GDT*,并使用 *init_idt*(), *set_trap*()和 *set_intr*()来初始化 *IDT*.在初始化后,*IDTR* 寄存器指向 *IDT* 表对象.在系统镜像刚加载到内存后,*IDT* 表刚被定义并且尚未被赋予正确的值,这时,由初始化的模块对 *IDT* 表进行赋值.

IDT 表的初始化代码(C 语言、汇编语言部分)如下所示:

```
void set_trap (
  struct GateDescriptor *ptr,
  uint_32 selector,
  uint_32 offset,uint_32 dpl)
{
  ptr→offset_15_0=offset;
  ptr→segment=selector;
  ptr→pad0=0;
  ptr→type=TRAP_GATE_32;
  ptr→system=FALSE;
  ptr→privilege_level=dpl;
  ptr→present=TRUE;
  ptr→offset_31_16=
    (offset>>16) & 0xFFFF;
}
push %ebp
mov %esp, %ebp
…
mov 0x10(%ebp), %eax
mov %eax, %edx
mov 0x8(%ebp), %eax
mov %eax, %edx
…
```

在汇编级抽象模型的基础上,*IDT* 表的初始化过程对应的 Isabelle/HOL 定义如下:

```
definition set_trap::"Code" where
  "set_trap==
  pushr ebp;
  movrr esp ebp;
  …
  movirr 4 ebp eax;
  movrr eax edx;
  movirr 2 ebp eax;
  movrir edx;
  …"
```

在上述对 *IDT* 表的初始化过程的定义中,指令 *movirr n reg*1 *reg*2 的语义是将内存地址 *reg*1+*n* 处的值赋给寄存器 *reg*2.在 *set_trap* 的定义中,*ebp*+4 指向参数的偏移地址.因此,*movirr* 4 *ebp eax* 将参数的偏移地址赋给 *eax* 寄存器,在执行完 *set_trap* 后,其语义为对 *IDT* 表进行了预期的赋值.

在完成对系统初始化过程的功能语义的定义后,为了验证这些功能语义的正确性,我们需要在 Isabelle/HOL 中给出这些正确性的命题公式,以此作为验证的目标.针对 *IDT* 初始化过程的正确性命题公式是:

```
…
s.M(idt_base+offset_low)=addr_low
s.M(idt_base+idt_segment)=segment
…
s.M(idt_base+idt_flag)=flag
s.M(idt_base+offset_high)=addr_high
…
```

其中,*s* 为状态对象,*s.M* 表示内存状态,*s.M*(*idt_base*+*offset_low*)表示内存位置 *idt_base*+*offset_low* 处的值.公式 *s.M*(*idt_base*+*offset_low*)=*addr_low* 说明在内存位置 *idt_base*+*offset_low* 处被正确地赋值了 *addr_low*.

6 系统功能模块的汇编级验证

限于篇幅,本节以 VSOS 微内核消息处理为例,阐述在 Isabelle/HOL 中如何在第 4.3 节中阐述的汇编级抽象模型的基础上对 VSOS 的功能模块在汇编级进行验证的方法.

现以 VSOS 微内核消息处理模块的发送消息函数 *sys_send* 在汇编级的正确性证明为例,阐述对 VSOS 的功能模块在汇编级进行验证的方法.*sys_send* 功能函数的工作对象集合主要包括发送进程和接收进程的进程控制块(process control block,简称 PCB).

sys_send 的 C 代码和汇编代码:

```
int sys_send (
  struct proc *caller_ptr,
  int dst, message *m_ptr,
  unsigned flags)
{ struct proc *dst_ptr=
    get_proc_from_pid(dst);
  …
  copy_mess(m_ptr,
  dst_ptr→p_messbuf,sizeof(message));
  …}
```

```
⟨sys_send⟩:
   push %ebp
   movl %esp, %ebp
   subl $0x28, %esp
   movl 0xc(%ebp), %eax
   movl %eax, (%esp)
   call c0101128
   …
   movl %eax, 0x4(%esp)
   movl 0x10(%ebp), %eax
   movl %eax,(%esp)
   call c010116d ⟨copy_mess⟩
   …
```

在第 4.3 节汇编级抽象模型的基础上,*sys_send* 功能函数的功效在 Isabelle/HOL 中定义如下:

```
definition sys_send::"Code" where
"sys_send==
pushr ebp;
movrr esp ebp;
subir 10 esp;
movirr 3 ebp eax;
movrir eax 0 esp;
call get_proc_from_pid;
…
movrir eax 1 esp;
movirr 4 ebp eax;
movrir eax 0 esp;
call copy_mess;
…"
```

在对 *sys_send* 语义定义的基础上,下面从汇编级对其正确性命题进行分析.

我们尝试从汇编级对系统功能模块的语义正确性进行验证,定义的正确性命题需要保证在任何可信状态下,功能模块运行的语义都能满足这些正确性命题.对于 *sys_send* 功能语义的正确性命题可以定义如下:

$$\forall s, Q(s)\wedge s'=NextnS(sys_send,s)\rightarrow P(s,s') \tag{3}$$

其中,$NextnS(sys_send,s)$表示在执行完 *sys_send* 的功能语义后的状态,即,多步执行后的效果(状态).为此,上述公式表示当开始状态 s 满足条件谓词 Q 时,*sys_send* 能正确地将消息发送给目标进程,即,开始状态 s 和结束状态 s'满足条件谓词 $P(s,s')$.值得一提的是,并不是所有的状态都可以满足谓词 Q,对于不满足 Q 的状态可以不做考虑.在公式(3)中,条件谓词 P 可以作为函数 *sys_send* 的功能语义表述,而 Q 作为初始条件.为此,针对 *sys_send*,谓词 Q 定义如下:

$$Q(s)\equiv(s.regs.sp+1=caller_ptr)\wedge(s.regs.sp+2=dst)\wedge(s.regs.sp+3=m_ptr) \tag{4}$$

公式(4)表明,*sys_send* 函数所需的参数存放在栈中合适的位置,并且拥有正确的值.

当发送进程将消息发给接收进程,并且接收进程也在等待发送进程时,*sys_send* 将发送进程消息缓冲区中预期长度的字节数复制到接收进程的缓冲区中.因此,*sys_send* 的正确性条件是这两个内存区间的值应该相同,也就是说,这两个消息缓冲区中的消息体相同.该正确性条件的描述如下:

$$P(s,s')\equiv CmpM(s',s.M(s.regs.sp+3),s'.M(proc+sizeof_proc*s.M(s.regs.sp+2)+p_messbuf),sizeof_msg) \quad (5)$$

其中,辅助函数 *CmpM* 的定义如下:

$$CmpM(s,p,q,n)\equiv(i\geqslant 0\wedge i\leqslant(n-1))\rightarrow(s.M(p+i)=s.M(q+i)) \quad (6)$$

对于上述公式(3)的证明,我们通过展开 *NextnS*(*sys_send*,*s*)操作,即多步执行后的效果,并根据第 4.3 节所描述的指令的操作语义,从而验证执行完 *sys_send* 之后的状态是否满足后置条件 *P*.

VSOS 验证环境配置见表 2.VSOS 汇编级的 Isabelle/HOL 验证工程量见表 3,完整的验证耗时 18min 左右.

Table 2 Configuration of verification system

表 2 验证系统配置信息

名称	版本	配置
Hardware	Dell Studio XPS 9100	Standard installation
CPU	Intel i7 930	2.8GHz
Memory	DDR3 SDRAM	3G
OS	Ubuntu 14.04 LTS	Standard installation
Isabelle	Isabelle2013-2_linux	Standard installation

Table 3 VSOS proof effort

表 3 VSOS 验证工作量

	被证明的模块		被证明的属性	被验证的代码量(汇编)	验证代码量	验证工作量 (人/年)
VSOS 验证	微内核	中断处理	完整性(integrity) 可靠性(soundness)	1.2k SLOC	15.6k SLOC	3.6
		系统任务	完整性(integrity) 可靠性(soundness)	1.1k SLOC	12.2k SLOC	2.8
		消息管理	完整性(integrity) 可靠性(soundness)	0.9k SLOC	13.7k SLOC	3.1
	进程管理		完整性(integrity)	1.6k SLOC	15.8k SLOC	3.6
	内存管理		完整性(integrity)	1.7k SLOC	14k SLOC	3.2
	文件系统		完整性(integrity)	0.9k SLOC	6.7k SLOC	1.7

VSOS 的完整性证明包括验证在系统运行过程中保持系统代码完整性和数据完整性以及系统功效的完整性.系统代码完整性和数据完整性是指代码和数据不可被恶意地修改,这些可以通过访问控制策略实现.系统功效的完整性是指系统的功能行为始终能完成系统设计所期望的效果.VSOS 微内核的可靠性证明是指验证微内核在运行过程中,不存在非预期的行为和恶意行为,即,系统行为导致的系统安全状态可达性问题.VSOS 的验证结果如图 4 所示,No subgoals 表明通过交互式的验证策略证明了命题目标.

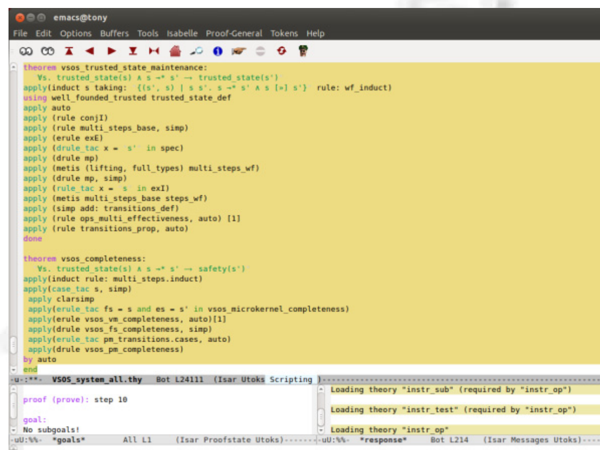

Fig.4 Isabelle/HOL verification result

图 4 Isabelle/HOL 验证结果

7 结 论

本文提出了一种OS系统状态模型,作为VSOS汇编语言层设计和验证的纽带,通过定义系统状态模型的合法状态和状态转换函数来建立系统状态模型的论域,并以此来描述VSOS汇编层的论域.在此基础上,通过给出汇编层的功能模块的正确性命题并借助Isabelle/HOL定理证明器来验证汇编层的功能模块的正确性,以此保证VSOS汇编语言层模块功效的正确性.

对于VSOS的完整验证,从系统实现的代码量和验证的代码量的比例的标准来看,我们达到了平均情况下1:10的量,可以认为我们的方法是可行和高效的.不可否认的是,整个验证过程共耗费了18人年的工作量,这是一项非常耗时的研究工作.为此,如何高效地采用模块化的验证策略,以及验证的复用问题,是我们接下来工作的主要方向.

值得一提的是,我们通过构造论域$M_{Isabelle/HOL}$和$M_{Computer}$来建立VSOS和逻辑系统之间的联系.$M_{Isabelle/HOL}$是完全形式化的模型,而$M_{Computer}$是半形式化的模型,例如对于硬件和客观环境等信息的描述是非形式化的,从严格意义上来讲,$M_{Isabelle/HOL}$和$M_{Computer}$之间的同构性问题是需要经过严格的形式化验证的,我们计划从模型论(model theory)和类型论(type theory)的角度来严格地形式化描述和验证$M_{Isabelle/HOL}$和$M_{Computer}$之间的同构性问题,这也是我们未来工作的重点.

致谢 本文作者感谢所有匿名审稿者,感谢您们对本文提出宝贵的意见.

References:

[1] Klein G, Andronick J, Elphinstone K, Murray T, Sewell T, Kolanski R, Heiser G. Comprehensive formal verification of an OS microkernel. ACM Trans. on Computer Systems, 2014,32(1):1−70. [doi: 10.1145/2560537]
[2] Shao Z. Certified software. Communications of the ACM, 2010,53(12):56−66. [doi: 10.1145/1859204.1859226]
[3] Alkassar E, Hillebrand MA, Leinenbach D, Schirmer NW, Starostin A. The verisoft approach to systems verification. In: Proc. of the 2nd IFIP Working Conf. on Verified Software: Theories, Tools, and Experiments. Springer-Verlag, 2008. 209−224. [doi: 10.1007/978-3-540-87873-5_18]
[4] Xavier L. Formal verification of a realistic compiler. Communications of the ACM, 2009,52(7):107−115. [doi: 10.1145/1538788.1538814]
[5] Dirk L, Elena P. Pervasive compiler verification—From verified programs to verified systems. Electronic Notes in Theoretical Computer Science (ENTCS), 2008,217:23−40. [doi: 10.1016/j.entcs.2008.06.040]
[6] Nipkow T, Paulson L, Wenzel M. Isabelle/HOL—A proof assistant for higher-order logic. In: Proc. of the LNCS 2283, Heidelberg: Springer- Verlag, 2002. [doi: 10.1007/3-540-45949-9]
[7] Walker BJ, Kemmerer RA, Popek GJ. Specification and verification of the UCLA Unix security kernel. Communications of the ACM, 1980,23(2):118−131. [doi: 10.1145/358818.358825]
[8] SRI Int'l. SRI project. http://www.sri.com
[9] Robinson L, Roubine O. Special: A specification and assertion language. Technical Report. Stanford Research Institute, 1977.
[10] Secure Computing Corp. DTOS formal security policy model. Technical Report, DTOS CDRL, 1996.
[11] Shapiro JS, Smith JM, Farber DJ. EROS: A fast capability system. In: Proc. of the SOSP'99. 1999. 170−185. [doi: 10.1145/319151.319163]
[12] Shapiro JS, Sridhar S, Doerrie MS. BitC language specification. Technical Report. http://coyotos.org
[13] Hohmuth M, Tews H, Stephens SG. Applying source-code verification to a microkernel: The VFiasco project. Technical Report, NewYork: ACM Press, 2002.
[14] Owre S, Rushby JM, Shankar N. PVS: A prototype verification system. In: Proc. of the CADE'92. 1992. 748−752. [doi: 10.1007/3-540-55602-8_217]
[15] Tews H, Weber T, Völp M, Poll E, Eekelen M, Rossum P. Nova micro-hypervisor verification formal, machine-checked verification of one module of the kernel source code. Technical Report, Nijmegen: Radboud University, 2008.
[16] Robin. Robin project. http://robin.tudos.org
[17] Heiser G, Murray T, Klein G. It's time for trustworthy systems. IEEE Security & Privacy, 2012,10(2):67−70. [doi: 10.1109/MSP.2012.41]
[18] Elphinstone K, Heiser G. From L3 to seL4—What have we learnt in 20 years of L4 microkernels? In: Proc. of the SOSP 2013. 2013. 133−150. [doi: 10.1145/2517349.2522720]

[19] Blackham B, Shi Y, Chattopadhyay S, Roychoudhury A, Heiser G. Timing analysis of a protected operating system kernel. In: Proc. of the RTSS 2011. 2011. 339–348. [doi: 10.1109/RTSS.2011.38]
[20] Stampoulis A, Shao Z. Static and user-extensible proof checking. In: Proc. of the POPL 2012. 2012. 273–284.
[21] Stampoulis A, Shao Z. VeriML: Typed computation of logical terms inside a language with effects. In: Proc. of the ICFP 2010. 2010. 333–344. [doi: 10.1145/1863543.1863591]
[22] Barendregt HP, Geuvers H. Proof-Assistants Using Dependent Type Systems. Amsterdam: Elsevier, 1999.
[23] Feng X. An open framework for certified system software [Ph.D. Thesis]. New Haven: Yale University, 2007.
[24] Liang H, Feng X, Shao Z. Compositional verification of termination-preserving refinement of concurrent programs. In: Proc. of the 23rd EACSL Annual Conf. on Computer Science Logic and 29th Annual IEEE Symp. on Logic in Computer Science (CSL-LICS 2014). 2014. [doi: 10.1145/2603088.2603123]
[25] Carbonneaux Q, Hoffmann J, Ramananandro T, Shao Z. End-to-End verification of stack-space bounds for C programs. In: Proc. of the PLDI 2014. 2014. [doi: 10.1145/2666356.2594301]
[26] Liang H, Hoffmann J, Feng X, Shao Z. Characterizing progress properties of concurrent objects via contextual refinements. In: Proc. of the CONCUR 2013. LNCS 8052, Heidelberg: Springer-Verlag, 2013. 227–241. [doi: 10.1007/978-3-642-40184-8_17]
[27] Liang H, Feng X, Fu M. A rely-guarantee-based simulation for verifying concurrent program transformations. In: Proc. of the POPL 2012. 2012. 455–468. [doi: 10.1145/2103656.2103711]
[28] Tan G, Shao Z, Feng X, Cai H. Weak updates and separation logic. New Generation Comput, 2011,29(1):3–29. [doi: 10.1007/s00354-010-0097-5]
[29] Fu M, Li Y, Feng X, Shao Z, Zhang Y. Reasoning about optimistic concurrency using a program logic for history. In: Proc. of the CONCUR 2010. 2010. 388–402. [doi: 10.1007/978-3-642-15375-4_27]
[30] Ferreira R, Feng X, Shao Z. Parameterized memory models and concurrent separation logic. In: Proc. of the ESOP 2010. 2010. 267–286. [doi: 10.1007/978-3-642-11957-6_15]
[31] Daum M, Dorrenbacher J, Bogan S. Model stack for the pervasive verification of a microkernel-based operating system. In: Proc. of the 5th Int'l Verification Workshop. 2008. 56–70.
[32] Alkassar E, Cohen E, Hillebrand MA, Kovalev M, Paul WJ. Verifying shadow page table algorithms. In: Proc. of the FMCAD 2010. 2010. 267–270.
[33] Alkassar E, Cohen E, Hillebrand MA, Pentchev H. Modular specification and verification of interprocess communication. In: Proc. of the FMCAD 2010. 2010. 167–174.
[34] Baumann C, Beckert B, Blasum H, Bormer T. Ingredients of operating system correctness. In: Proc. of the Embedded World 2010 Conf. 2010.
[35] Li W. Mathematical Logic: Basic Principles and Formal Calculus. 2nd ed., Beijing: Science China Press, 2007 (in Chinese).
[36] Linz P. An Introduction to Formal Languages and Automata. 3rd ed., Sudbury: Jones and Bartlett Publisher, 2004.
[37] Marker D. Model Theory: An Introduction. New York: Springer-Verlag, 2002. [doi: 10.1007/b98860]

附中文参考文献:

[35] 李未.数理逻辑:基本原理与形式演算.第 2 版,北京:科学出版社,2007.

钱振江(1982－),男,江苏苏州人,博士,讲师,CCF 高级会员,主要研究领域为操作系统安全,形式化验证,嵌入式系统.

宋方敏(1961－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为数理逻辑,量子计算机.

黄皓(1957－),男,博士,教授,博士生导师,主要研究领域为系统软件,信息安全.