# 嵌入式操作系统的形式化验证研究

陈丽蓉 李 允 罗 蕾

(电子科技大学计算机科学与工程学院 成都 611731)

**摘 要** 描述了一个汽车电子嵌入式实时操作系统的分层形式模型:在低层,该操作系统的顺序内核承担基础设施的角色,实施任务、ISR和系统服务等并发执行体之间的切换;而在高层,该操作系统向用户提供可并发执行的系统服务。两个层次的模型具有不同的配置状态视图和操作粒度。作为最重要的安全相关特性,应用与OS之间的存储隔离保护机制在顺序内核的模型中得以体现。建立了操作系统的实现正确性定理,包括相应的仿真关系和实现不变量。根据该操作系统两个部分模型的特点及相应代码的实现语言情况,选择组合应用定理证明器Isabelle/HOL和程序验证工具VCC的方式,有效完成了该操作系统的形式化验证。

**关键词** 嵌入式操作系统,形式化验证,建模,Isabelle/HOL,VCC

**中图法分类号** TP316.2 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2015.8.043

## Research on Formal Verification of Embedded Operating System

CHEN Li-rong LI Yun LUO Lei

(School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu 611731, China)

**Abstract** A layered formal model for an automotive embedded real-time operating system was presented. At the lower layer, the sequential kernel plays the infrastructural role in executing switching between concurrent entities such as tasks, ISRs and system services, and at the higher layer the concurrent system services are provided to users. The two layers of the model have different views of configurations and operation granularities. As the most important safety related feature, the memory isolation and protection mechanism between applications and the OS is modeled in the sequential kernel. The implementation correctness theorem of the OS was established along with the corresponding simulation relation and implementation invariants. According to the features of the model and the related implementation languages, the OS was formally and effectively verified with a combined usage of the theorem prover Isabelle/HOL and the program verifier VCC.

**Keywords** Embedded operating system, Formal verification, Modeling, Isabelle/HOL, VCC

## 1 引言

目前,越来越多的科学家及软件工程师将目光聚焦在软件的建模以及形式化验证方面,尤其对于那些在安全关键系统中应用的软件。汽车控制系统是典型的安全关键或安全相关系统。道路车辆功能安全国际标准ISO26262-6[1]高度推荐在高ASIL(汽车安全完整性)等级系统软件开发中运用形式化方法。嵌入式实时操作系统由于其在相关系统中的核心地位而成为形式化方法高度关注的目标。

本文工作的现实需求来自于开发并验证一个嵌入式实时操作系统,该系统提供存储保护、定时保护、服务保护、硬件保护等安全相关功能,满足AUTOSAR OS规范[2]以及ISO26262-6中"Freedom from interference by software partitioning"的需求。该操作系统称为eAutoOS,主要由2大部分组成:内核和系统服务,提供任务管理、资源管理、事件管理、通信管理、中断管理、计数器及报警器管理、OS应用管理等功能。其旨在成为一个可信的嵌入式实时操作系统,并应用于汽车工业的ECU(电子控制单元)中。

业界已有不少关于内核、操作系统等软件的形式化验证研究工作,从文献[3,4]中可获得一个比较深入的了解。近年来在这个领域,国外比较著名的项目包括由德国萨尔大学、慕尼黑工业大学、微软欧洲研究院等机构联合开展的Verisoft及Verisoft XT项目,以及澳大利亚国家ICT实验室(NICTA)发起实施的L4.verified项目。L4.verified项目采用交互式的定理证明器Isabelle/HOL,对seL4微内核的基于ARMv6平台的版本进行了功能正确性及其访问控制等安全相关特性的证明[5-7]。Verisoft项目的目标在于对整个计算机系统从底层硬件到应用软件自底向上地进行普适化的形式证明。该项目基于Isabelle/HOL建立了一个面向顺序命令式编程语言的通用程序验证框架Isabelle/Simpl[8],并基于此开展了通信虚拟机CVM[9-12]、VAMOS[13,14]和OLOS[15]内核以及简单的用户级操作系统SOS[16]的形式化验证。Verisoft XT项目则使用面向并发C程序验证的工具链VCC+Boogie+Z3,对微软Hyper-V管理程序[17,18]、PikeOS[19]等进行了验

到稿日期:2014-07-09 返修日期:2014-11-20 本文受国家自然科学基金项目(61401067),四川省应用基础研究项目(2013JY0002)资助。

陈丽蓉(1972-),女,硕士,高级工程师,主要研究方向为嵌入式系统及软件、软件形式化验证等,E-mail:lrchen@uestc.edu.cn;李 允(1971-),男,博士,研究员,主要研究方向为嵌入式系统、高可信嵌入式软件技术;罗 蕾(1967-),女,硕士,教授,主要研究方向为嵌入式系统及软件。

证。这些验证对象中既有通用操作系统或内核(seL4、VAMOS),也包括一些面向专用应用领域比如航空电子(PikeOS)、汽车电子(OLOS)等与安全相关的操作系统或内核。

在国内,若干高校及机构也开展了有关操作系统形式化验证的研究工作。文献[20]针对 L4 微内核操作系统的内存管理机制开展了形式化验证。文献[21-23]以高阶逻辑和类型论为基础,提出了操作系统对象语义模型(OSOSM),对可信操作系统 VTOS 的安全属性、微内核架构的中断机制、VTOS 汇编语言层的正确性等进行了验证,确保了 VTOS 设计和安全需求的一致性。文献[24]针对 SELinux 的安全策略,将以 UML 描述的安全模型编译成模型检测器的规范语言,使用模型检测分析安全模型的性能,验证策略实施与安全需求之间的一致性。文献[25]对 LINUX IPC 子系统中的 SystemV 进程通信机制进行了验证。文献[26]针对访问验证保护级安全操作系统原型,采用 B 语言对安全策略模型进行形式化建模,以模型检测为基础验证了安全策略模型的正确性。文献[27]针对机载嵌入式软件,应用霍尔逻辑的相关推理规则,以定理证明的方式开展了程序验证。文献[28]针对 L4/Fiasco 微内核操作系统的改进的 IPC 机制 L4STM,对 L4STM 设计中的并行算法进行了建模和验证。文献[29]报道了一款面向汽车电子应用领域并经过了形式化验证的嵌入式操作系统 ORIENTAIS,其符合 OSEK/VDX OS 规范。文献[30]针对 AUTOSAR 操作系统的部分模块建立了模型,验证了任务间的互斥性、调度表间的互斥性、天花板优先级协议、防止优先级反转以及资源分配无死锁性等性质,但尚未报道其在保护相关属性方面的证明。上述研究工作中主要使用了模型检测工具 SPIN[20,25,28]或 PAT[30]、定理证明辅助工具 Isabelle/HOL[21-23]或 Coq[27],以及并发 C 程序验证的工具链 VCC+Boogie+Z3[29]。

软件的形式化建模及验证工作涉及大量的数学、逻辑推理等技术,因而相对于传统的软件设计及验证技术,其代价也不低。Gerwin Klein 等人提出,适合进行形式化验证的软件对象的规模基本应限制在 10000 行源代码以内[3]。嵌入式实时内核、嵌入式操作系统等由于其规模较小(通常只有几千行的 C 代码),同时又具有较高的安全或可信等级的要求,因此成为形式化验证的重要对象。

运用合理的工具对于提供可信度高的形式化验证结果很重要。比如在 Verisoft 项目中,绝大多数部件在设计实现期间都经过了手工的验证,而 75%左右的验证在 Isabelle/HOL 环境中进行了同步的机器级验证[3]。在 seL4 的验证中也主要使用了 Isabelle/HOL 环境。HOL 是经典的基于类型化 λ 演算的高阶逻辑(Higher-Order Logic),而 Isabelle/HOL 则是在通用辅助证明工具 Isabelle 中实现了 HOL,内含大量已形式化的数学逻辑,非常适合对一些重要的数学性质进行推理验证。但是,使用 Isabelle/HOL 来验证诸如 C 这样的语言的程序,代价是相对较高的,因为这一方面会引入与语言相关的工作开销(对相关语言的语法及语义进行形式化描述,针对其程序的部分或完全正确性建立相应的逻辑并证明其合理性和完备性[8]),另一方面还需要在此基础上对程序的逻辑进行转化描述[13,14]。

在 Verisoft 的二期项目即 Verisoft XT 中,为了更高效地验证操作系统的 C 程序,开发并应用了 VCC 环境,并且成熟的 VCC 工具也是该项目的一个重要成果。VCC 是面向 C 代码的自动化验证器,其通过注解式语言,能够对 C 语言程序进行"合同编程",描述各种数据结构的不变量、函数的前置条件与后置条件、断言等,并能通过其特有的 ghost 变量及代码,增强描述程序的逻辑规范(二阶逻辑),实现与其他逻辑工具如 Isabelle/HOL 的连接[31]。VCC 内嵌的 ownership 模型能够很好地满足验证并发程序的需要,因此成为验证诸如微软 Hyper-V 管理程序这样的系统核心软件的重要选择[17]。

然而,VCC 不支持其他语言如汇编语言的程序验证。Alkassar[32]、Shadrin[33]等人尝试在 VCC 中以仿真的方式建立目标处理器的指令集体系架构模型,形式化其汇编指令的操作语义,以便在 VCC 中验证一个完整的软件——BHV(Baby HyperVisor),包括其 C 代码部分和汇编代码部分。

因此,不同工具(定理证明器或代码验证工具)工作的机理不同,因而采用何种工具进行程序的验证与被验证对象的模型性质及实现语言有关。本文的目的在于,针对被验证对象(汽车电子嵌入式操作系统)的功能及特性需求,研究其形式化建模的技术,以及更重要的是,如何根据其模型的特点以及工具对相关实现语言的支持程度,选择并组合适宜的工具环境(上述相关工作基本是在单一工具环境中进行的验证),对该操作系统软件进行有效的验证,从而掌握面向安全相关领域嵌入式操作系统的形式化设计及验证的关键技术。

## 2 OS 特性与需求

eAutoOS 是一款典型的、应用于汽车电子实时控制系统的嵌入式操作系统,满足以下特征:

- 采用基于任务优先级的可抢占式调度策略。
- 支持可抢占的系统服务,即系统服务在执行某些非临界区代码时可以被中断打断,进而可被其他任务抢占。
- 提供存储保护、定时保护、服务保护、硬件保护等安全相关功能,使得具有不同可信属性或 ASIL 等级的应用可以在一个 ECU 中集成。
- 支持优先级天花板协议[34],以便有效地解决优先级反转以及死锁的问题。支持 4 种资源类型——标准资源、内部资源、中断资源和调度器资源,使得应用可以更为灵活地定义各个任务排斥其他任务抢占其 CPU 使用权利的范围和时间长度。
- 支持扩展任务,支持同优先级多个任务以及基本任务的多次激活。
- 提供符合 OSEK/AUTOSAR OS 规范[2,34]的 API 接口。

与 AUTOSAR OS 规范相一致,eAutoOS 将系统中的应用分为可信的与不可信的两大类,它们分别运行于处理器的特权或非特权模式下。可信与不可信的区分取决于应用本身,嵌入式系统的集成者可根据多种因素(应用具有的 ASIL 等级、其供应商的可信度、应用的成熟程度等)的综合赋予应用不同的可信属性[35]。可信的应用可以在关闭监视器及保护功能的情况下运行,而不可信的应用不允许在关闭监视器及保护功能的情况下运行。因此操作系统必须提供有关的机制,实现可信应用与不可信应用之间的隔离和保护。每个应用可能包含若干任务和/或 ISR,它们具有与所属应用相同的可信/不可信属性。

图 1 所示为基于 eAutoOS 的软件系统结构。在单处理器的系统中，任何时刻只能有一个任务在执行，并且可被一系列的中断打断，而中断也是可以嵌套的。任务和中断服务程序(简称 ISR)均可调用 OS 的系统服务，而系统服务也是可以在执行其非临界区代码时被中断进而被抢占的。任务的切换只能在所有嵌套的中断都执行完成时才能进行。

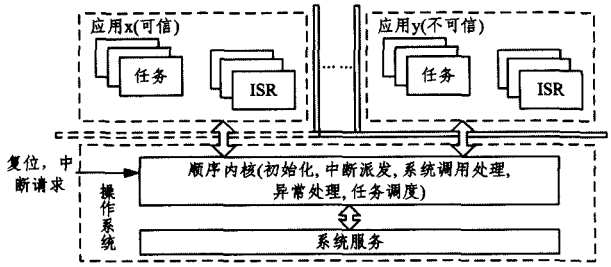


图 1 基于 eAutoOS 的软件系统结构

因此在基于 eAutoOS 的应用系统中，定义 3 种类型的并发执行体：任务、ISR 和可被抢占的系统服务。建模中断处理程序的一个自然途径是将其作为并发的线程[17]。操作系统在系统服务的非临界代码区可被中断，然而在执行其一些关键的底层机制时，中断是被严格禁止的。本文把包含初始化、中断派发、系统调用处理、异常处理、任务调度(及切换)等功能在内的操作系统底层机制划分到所谓的“顺序内核”中，这些机制实现任务、ISR 与系统服务之间的原子性切换，其执行过程不能被打断。因此顺序内核的关键作用是建立一个框架，使得其上层的执行体(应用中的任务和 ISR)可以安全地进行彼此间以及和系统服务之间的交互。顺序内核中底层机制的实现一方面与硬件相关(涉及汇编语言程序)，另一方面也部署了核心的安全相关机制如存储隔离与保护。

## 3 OS 建模

如上所述，本文将整个操作系统分解为 2 个大的部分进行建模：顺序内核和系统服务。基于它们的功能特性及操作对象，从不同的关注角度定义它们的配置状态以及变迁函数。在此之前，需要首先明确系统的内存部署策略。

### 3.1 系统内存部署策略

为了满足 AUTOSAR OS 的 3 个层面的存储保护要求即操作系统的保护、应用的隔离和执行体的隔离，整个系统的内存被分为若干个互不相交的分区，每个分区位于一个物理内存页中，通过 MMU 进行地址转换和访问权限控制[36]。操作系统以及每个不可信应用的数据段和代码段被部署到不同的分区中，所有可信的应用和操作系统共享相同的分区。为了能够从任务和 ISR 的粒度区分不同的栈空间，本文定义了 4 种类型的栈：

- *TUS*：任务用户栈的集合，其中每个栈对应一个不可信的任务；
- *IUS*：中断用户栈的集合，其中每个栈对应一个不可信的 ISR；
- *TSS*：任务系统栈的集合，其中每个栈对应一个任务；
- *iss*：中断系统栈，被所有的 ISR 共享。

*TUS* 和 *IUS* 中的每个栈位于相应不可信应用的数据分区中，它们只被所对应的任务或 ISR 用于执行其自身代码或其所属应用提供的公共函数。*TSS* 中的系统栈和 *iss* 均位于操作系统的数据分区中，它们被相应的任务或 ISR 用于保存中断上下文或执行系统服务。可信的任务和 ISR 还能在相应的系统栈中执行其自身代码或有关的公共函数。图 2 为基于此策略的内存部署图。

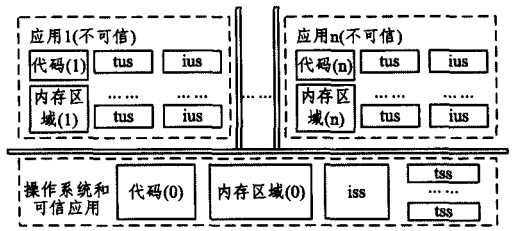


图 2 系统内存部署

为了管理分离的内存区域，顺序内核的配置状态视图中包含这些分离的内存区域以及栈空间。顺序内核的主要功能就是计算当前的执行体，在不同的执行体之间实施转换即内存区域及栈的转换。在转换过程中需要满足内存保护的要求。而操作系统的系统服务则有不同的配置状态视图。该视图所关联的数据结构位于操作系统的专用内存区域中。该部分的执行不会导致当前工作的内存区域及栈的变更。

### 3.2 顺序内核建模

#### 3.2.1 系统配置状态

配置状态是系统受程序执行的影响而动态变化的状态。面向顺序内核的系统抽象配置状态定义为如式(1)所示的记录类型：

$$c_{eas} \equiv (mr, tus, ius, tss, iss, sr, ct, aisrList, asv) \in C_{eas} \quad (1)$$

具有这样配置状态的系统为 *EAS* 系统，$C_{eas}$ 是其所有配置状态的集合。其中：

$$c_{eas}.mr :: [0; nna] \rightarrow (B^{32} \rightarrow B^{8}) \quad (2)$$

如式(2)所示，函数 $c_{eas}.mr$ 实现从应用 ID 号 $aid \in [0; nna]$ 到可按字节编址的内存区域的映射，其中 $nna \in N$ 是不可信应用的数量，而 0 号区域被操作系统和可信应用使用。这些内存区域用于全局变量的空间分配。$B$ 表示二进制数字的集合，即 $=\{0,1\}$。

$$c_{eas}.tus :: NTID \rightarrow frame^{*}_{C\_IL} \quad (3)$$

如式(3)所示，函数 $c_{eas}.tus$ 实现从不可信任务 ID 号到其用户栈上的栈帧序列的一个部分映射，栈帧的类型只能是 $frame_{C\_IL}$。$NTID$ 是所有不可信任务 ID 号的集合。

$$c_{eas}.ius :: NIID \rightarrow frame^{*}_{C\_IL} \quad (4)$$

如式(4)所示，函数 $c_{eas}.ius$ 实现从不可信 ISR ID 号到其用户栈上的栈帧序列的一个部分映射，栈帧的类型只能是 $frame_{C\_IL}$。$NIID$ 是所有不可信 ISR ID 号的集合。

$$c_{eas}.tss :: TID \rightarrow (frame_{C\_IL} \cup frame_{INT} \cup frame_{SCI})^{*} \quad (5)$$

如式(5)所示，函数 $c_{eas}.tss$ 实现从任务 ID 号到其系统栈上的栈帧序列的一个部分映射，栈帧的类型可以是 $frame_{C\_IL}$，$frame_{INT}$ 或 $frame_{SCI}$。$TID$ 是所有任务 ID 号的集合。

$c_{eas}.iss \in (frame_{C\_IL} \cup frame_{INT} \cup frame_{SCI} \{\perp\})^{*}$ 表示中断系统栈是一个栈帧的序列，栈帧的类型可以是 $frame_{C\_IL}$、$frame_{INT}$ 或 $frame_{SCI}$。

$c_{eas}.sr \in \{INT\_ENABLED, INT\_DISABLED\}$ 表示系统的中断使能状态。

$c_{eas}.ct \in TID$ 表示当前任务的 ID 号。

$c_{eas}.aisrList \in IID^*$ 是活动 ISR ID 的列表(模拟嵌套中断服务的 LIFO 队列),$IID$ 是所有 ISR ID 号的集合。

$$c_{eas}.asv :: TID \cup IID \mapsto B \tag{6}$$

如式(6)所示,函数 $c_{eas}.asv$ 用于计算一个任务或 ISR 是否已调用了一个系统服务且该系统服务尚未返回。

文献[18,33]中定义了 C-IL 及 CC-IL(并发 C-IL)配置模型。C-IL 配置中,只定义了 1 个内存区域和 1 个系统栈空间,旨在描述 C-IL 语言的语义,以及定义和证明其配套编译器的正确性。CC-IL 配置模型则定义了 1 个内存区域和若干个栈空间,每个栈空间对应一个线程,但它们共享同一个内存区域,没有隔离,线程之间的切换只涉及栈空间的切换。这两种配置都不足以表达顺序内核的模型,因为顺序内核管理的各个执行体(任务、ISR 和系统服务)使用的是不同的内存区域和栈空间。在执行体之间进行切换时,涉及内存区域和栈空间的切换,这也是顺序内核需要正确处理好的核心问题。因此本文进行了如下扩展:(1)存储全局变量的内存区域被分为 $1+nna$ 个互不相交的部分,与应用的数量相对应;(2)栈空间分为 4 种类型;(3)除了普通 C 函数调用涉及的 C-IL 帧外,还定义了 2 种由顺序内核产生的特殊类型的抽象帧:

$frame_{INT}$:保存被中断实体的中断上下文的特殊帧。中断上下文的内容取决于处理器的体系结构以及编译规则,内核的实现必须遵守此规则以保证被中断实体的正确恢复。不失一般性,该抽象帧定义为如式(7)所示的记录类型:

$$frame_{INT} \equiv (p :: P_{name}, loc :: \mathbb{N}, spr :: \mathbb{Z}^*, gpr :: \mathbb{Z}^*) \tag{7}$$

其中,$p$ 是中断派发程序的名字(简称 $IDP$),$loc$ 记录 $IDP$ 中调用用户 ISR 函数的指令的位置。$gpr$ 和 $spr$ 分别为存储通用寄存器和特殊寄存器内容的整数列表,这些寄存器的内容应该在调用用户 ISR 函数之前保存在该特殊帧中。N 表示包括 0 在内的自然数的集合,Z 表示整数的集合。

$frame_{SCI}$:该特殊帧用于从非可信的任务或 ISR 中调用系统服务时保存某些重要信息,其定义为如式(8)所示的记录类型:

$$frame_{SCI} \equiv (p :: P_{name}, loc :: \mathbb{N}, sra :: \mathbb{Z}, sms :: \mathbb{Z}, sp :: \mathbb{Z}) \tag{8}$$

其中,$p$ 是系统调用处理程序的名字,$sra$ 是系统调用执行完成之后的返回地址,$sms$ 是执行系统调用指令之前的机器状态,$sp$ 是保存的栈指针。

普通 C-IL 帧由并发执行体的 C 函数产生,其定义如式(9)所示:

$$frame_{C\_IL} \equiv (f :: F_{name}, loc :: \mathbb{N}, \mathcal{M}_\xi :: \mathbb{V} \rightharpoonup (B^8)^*, rds :: val_{ptr} \cup val_{bref} \cup \{\perp\}) \tag{9}$$

其中,$F_{name}$ 表示 $C$ 函数名称的集合,$f$ 是该帧所关联的 C 函数名称;位置计数器 $loc$ 是下一个将被执行的语句在 $f$ 函数体中的序号;$\mathcal{M}_\xi$ 是存储局部变量和参数的内存空间,$rds$ 表示返回值的地址。

C-IL 配置被定义为如式(10)所示的记录类型 $C_{C\_IL}$:

$$C_{C\_IL} \equiv (\mathcal{M} :: B^{32} \rightharpoonup B^8, s :: frame^*_{C\_IL}) \in C_{C\_IL} \tag{10}$$

### 3.2.2 变迁函数

EAS 系统的顶级变迁函数 $\delta_{eas}$ 以一个 EAS 的配置状态 $c_{eas}$ 和外部事件 $eev \equiv (eint :: B^{32}, reset :: B)$(包括外部中断请求 $eint$ 和复位信号 $reset$)作为输入,完成一个执行步骤后系统更新到一个新的配置状态 $c'_{eas}$:

$$\delta_{eas} :: C_{eas} \times (B^{32} \times B) \mapsto C_{eas} \tag{11}$$

该变迁函数根据对该系统被触发的具体的原因判定,调用相应的内核二级变迁函数,其定义如式(12)所示:

$$c'_{eas} = \delta_{eas}(c_{eas}, eev) = \begin{cases} \delta_{eas\_init}, & reset?(eev) \\ \delta_{eas\_IDP1}(c_{eas}, eev), & ir?(c_{eas}, eev) \\ \delta_{eas\_IDP2}(c_{eas}), & rfuisr?(c_{eas}, eev, \pi) \\ \delta_{eas\_SCI1}(c_{eas}), & apiCall?(c_{eas}, eev, \pi) \\ \delta_{eas\_SCI2}(c_{eas}), & rfsc?(c_{eas}, eev, \pi) \\ \delta_{eas\_SCHD}(c_{eas}), & scheduler?(c_{eas}, eev, \pi) \\ \delta_{eas\_me}(c_{eas}), & me?(c_{eas}, eev) \\ \delta_{eas\_internal}(c_{eas}), & internalStep?(c_{eas}, eev, \pi) \end{cases} \tag{12}$$

$\pi \in Prog_{EAS}$ 是 EAS 系统的程序,由 C 语言部分 $\pi_{C\_IL}$ 和汇编语言部分 $\pi_{ASM}$ 组成,$Prog_{EAS}$ 的定义如式(13)所示:

$$Prog_{EAS} \equiv (\pi_{ASM} :: Prog_{ASM}, \pi_{C\_IL} :: Prog_{C\_IL}) \tag{13}$$

$reset?(eev)$ 判定系统发生了复位;$ir?(c_{eas}, eev)$ 判定对一个外部中断请求的响应;$rfuisr?(c_{eas}, eev, \pi)$ 判定从一个用户 ISR 中返回;$apiCall?(c_{eas}, eev, \pi)$ 判定对用户(任务或 ISR)进行系统调用的响应;$rfsc?(c_{eas}, eev, \pi)$ 判定从某个系统服务中返回;$scheduler?(c_{eas}, eev, \pi)$ 判定内核执行任务调度及切换过程;$me?(c_{eas}, eev)$ 判定对访存相关异常的响应;$internalStep?(c_{eas}, eev, \pi)$ 判定上述各种情况都没有发生,则当前执行体执行其自身代码的一个步骤。式(14)和式(15)是其中 2 个判定的定义示例。

$$ir?(c_{eas}, eev) \equiv \neg reset?(eev) \wedge c_{eas}.sr = INT\_ENABLED \wedge eev.eint \neq 0 \tag{14}$$

判定 $ir?(c_{eas}, eev)$ 要求:为了能够响应一个外部中断请求,首先系统没有被复位,其次系统必须处于中断使能状态,且外部中断请求不为 0。

$$rfuisr?(c_{eas}, eev, \pi) \equiv \neg(reset?(eev) \vee ir?(c_{eas}, eev) \vee me?(c_{eas}, eev)) \wedge stmt_{curr}(c_{eas}, \pi) = return \wedge frame_{curr}(c_{eas}).f = fisr(ci(c_{eas})) \tag{15}$$

判定 $rfuisr?(c_{eas}, eev, \pi)$ 要求在复位、中断请求、访存异常等事件都未发生的前提下($\neg(reset?(eev) \vee ir?(c_{eas}, eev) \vee me?(c_{eas}, eev))$),当前即将执行的是 $return$ 语句($stmt_{curr}(c_{eas}, \pi) = return$),并且系统的当前栈帧所对应的函数(即正在执行的函数)是当前正在处理的用户 ISR 函数($frame_{curr}(c_{eas}).f = fisr(ci(c_{eas}))$)。辅助函数 $stmt_{curr}(c_{eas}, \pi)$ 用于计算 $c_{eas}$ 的当前 C 语句;$frame_{curr}(c_{eas})$ 属于 $frame_{C\_IL}$ 类型,是 $c_{eas}$ 当前工作栈的当前(顶级)帧;$fisr :: IID \mapsto FN$ 实现从 ISR ID 号到其 C 函数名之间的映射;$ci(c_{eas}) \in IID \cup \{\perp\}$ 计算当前 ISR 的 ID 号,在其为 $\perp$ 的情况下,系统没有进行中断处理,则当前任务 $c_{eas}.ct$ 正在执行。

EAS 系统可被看作是顺序内核和并发执行体之间的交替执行。所有并发执行体的执行被封装在一个单一的变迁函数 $\delta_{eas\_internal}(c_{eas})$ 中,它不涉及当前执行体的转换。在 $\delta_{eas\_internal}(c_{eas})$ 中当前执行体执行一个 C-IL 变迁步骤[18,33],该变迁仅仅作用于同一个 C-IL 机器(由一个特定的内存区域和一个栈组成)。除 $\delta_{eas\_internal}(c_{eas})$ 外,$\delta_{eas}$ 其他所有的子变迁函数都是顺序内核的原子性变迁。$\delta_{eas\_init}$ 使系统到达初始状态 $c^0_{eas}$,此时当前执行体是系统的初始化任务;根据访存异常的具体原因,$\delta_{eas\_me}$ 执行相应的异常处理程序。$\delta_{eas\_IDP1}$,$\delta_{eas\_IDP2}$,$\delta_{eas\_SCI1}$,

$\delta_{eas\_SCI2}$ 和 $\delta_{eas\_SCHD}$ 均涉及到当前执行体的转换并因此在不同的 C-IL 机器间切换。C-IL 机器间的切换通过内核特殊帧粘合起来，因而操作系统能够通过多个栈跟踪不同的并发线程。

$\delta_{eas\_IDP1}$ 和 $\delta_{eas\_IDP2}$ 是与操作系统的中断派发过程有关的一对变迁函数，分别对应中断派发程序的前半段和后半段。$\delta_{eas\_IDP1}$ 将系统的当前执行体转换为一个新的用户 ISR，$\delta_{eas\_IDP2}$ 则退出当前的用户 ISR，回到之前被该 ISR 打断的执行体(此时也可能切换到另一个任务)。$\delta_{eas\_SCI1}$ 和 $\delta_{eas\_SCI2}$ 是一对与系统调用接口有关的变迁函数，前者从当前任务或 ISR 切换到一个系统服务中，而后者完成相反的操作。$\delta_{eas\_SCHD}$ 实施任务调度及切换。在顺序内核执行过程中中断被禁止，因此这些变迁函数都能够在不被打断的情况下执行至完成，它们都可被看作是 EAS 系统执行的“一个步骤”。下面以 $\delta_{eas\_IDP1}$ 为例来说明内核变迁函数的建模情况。

中断请求可以在任务、ISR 或系统服务的执行期间被响应。$\delta_{eas\_IDP1}$ 定义了从一个中断请求被处理器响应的时刻到目标 ISR 函数被中断派发程序调用之后一刻的系统配置变化。在此期间：

• 根据系统当前嵌套的中断层次(通过辅助函数 $id(c_{eas})$ 计算)，中断上下文被保存到被中断执行体的系统栈(任务系统栈或中断系统栈)上。

• 根据目标 ISR 的可信/不可信属性，其对应的函数调用帧在其用户栈或中断系统栈上被建立。该属性通过辅助函数 $imode(iid)$ 计算，$iid \in IID$ 表示目标 ISR 的 ID 号。

令 $c'_{eas}=\delta_{eas\_IDP1}(c_{eas}, eev)$，有如式(16)的定义：

$$\frac{\begin{array}{l} ir?(c_{eas}, eev) \quad iid=il(eev) \\ c_{C\_IL}=(mregion_{curr}(c_{eas}), stack_{curr}(c_{eas})) \\ ctx\_int_{frame}(c_{eas}, frame_{ctx\_int}) \\ isr_{frame}(\pi.\pi_{C\_IL}, \theta, fisr(iid), frame_{newisr}) \\ s_1=\begin{cases} tss(c_{eas}.ct) & id(c_{eas})=0 \\ iss & id(c_{eas})>0 \end{cases} \\ s_2=\begin{cases} ius(iid) & imode(iid)=M\_USR \\ iss & imode(iid)=M\_SYS \end{cases} \end{array}}{\pi, \theta \vdash c_{eas} \rightarrow_{eas} c''\begin{bmatrix} aisrList=iid \circ c''.aisrList \\ s_2=frame_{newisr} \circ c''.s_2 \end{bmatrix}} (\delta_{eas\_IDP1})$$

$$c''=c_{eas}[s_1=frame_{ctx\_int} \circ c_{eas}.s_1] \tag{16}$$

辅助函数 $mregion_{curr}(c_{eas})$ 用于计算 $c_{eas}$ 的当前内存区域，$stack_{curr}(c_{eas})$ 计算当前的工作栈。$l_1 \circ l_2$ 表示列表 $l_1$ 和 $l_2$ 的连结。具体的外部中断请求号通过 $il(eev)=\min\{j \mid j \in [0:31] \wedge eev.eint[j]=1\}$ 计算。帧 $frame_{ctx\_int}$ 的内容由判定 $ctx\_int_{frame}(c_{eas}, frame_{ctx\_int})$ 中如式(17)所示的约束条件所限制：

$$\begin{array}{l} frame_{ctx\_int} \in frame_{INT} \quad frame_{ctx\_int} \cdot p=IDP \\ frame_{ctx\_int} \cdot loc=\text{调用用户 ISR 之后的指令序号} \end{array} \tag{17}$$

$frame_{ctx\_int}$ 中 $spr$ 和 $gpr$ 的内容因与目标处理器的体系架构相关而被定义在内核的实现不变量中，即该变迁函数的实现必须保证通过此特殊帧能够正确地保存和恢复这些寄存器的内容。

帧 $frame_{newisr} \in frame_{C\_IL}$ 的内容由判定 $isr_{frame}(\pi.\pi_{C\_IL}, \theta, fisr(iid), frame_{newisr})$ 中如式(18)所示的约束条件所限制：

$$\begin{array}{l} isr_{frame} :: Prog_{C\_IL} \times Params_{C\_IL} \times F_{name} \times frame_{C\_IL} \mapsto B \\ isr_{frame}(\pi, \theta, f, frame) \equiv frame.loc=0 \wedge frame.f=f \wedge \\ frame.rds=\perp \wedge \forall 0 \leqslant i < len(V(f)): len(frame.\mathcal{M}_c(v_i))= \\ size_\theta(t_i) \end{array} \tag{18}$$

其中，$Prog_{C\_IL}$ 表示 C-IL 程序的集合，而 $\pi.\pi_{C\_IL}$ 即是具体的 EAS 程序的 C-IL 部分。$Params_{C\_IL}$ 表示 C-IL 程序环境参数的集合，而 $\theta$ 则表示一个具体环境参数记录。$V(f)$ 是函数 $f$ 的局部变量及其类型的集合，$v_i$ 是其中具有类型 $t_i$ 的第 $i$ 个变量。上述公式的最后一个合取项要求被调用函数的所有局部变量都在新的栈帧中被分配了适宜的空间。

### 3.3 系统服务建模

将操作系统的系统服务部分建模为一个自动机 $A_{ck}$，其定义如式(19)：

$$A_{ck} \equiv (C_{ck}, C^0_{ck}, \Sigma_{ck}, \Omega_{ck}, \delta_{ck}) \tag{19}$$

其中，$C_{ck}$ 是该自动机的状态集合，$C^0_{ck}$ 是其初始状态的集合且有 $C^0_{ck} \subset C_{ck}$。$\Sigma_{ck}$ 是输入符号表，$\Omega_{ck}$ 是输出符号表，而 $\delta_{ck}$ 是其变迁函数。输入主要来自应用程序通过顺序内核传递过来的系统服务调用参数，输出则依赖于每个系统服务的返回值。$A_{ck}$ 部件之间的关系如式(20)所示：

$$(c'_{ck}, o_{ck})=\delta_{ck}(c_{ck}, i_{ck}) \tag{20}$$

其中，变迁函数 $\delta_{ck}$ 以源状态 $c_{ck}$ 和输入 $i_{ck}$ 为参数，产生一个目标状态 $c'_{ck}$ 和输出 $o_{ck}$。

#### 3.3.1 静态配置信息与动态配置状态

与顺序内核不同，除了动态变化的配置状态外，系统服务还包括一些相关的静态配置信息，其内容包含：

• 任务 ID 号的集合：$TID=\{1..TASK\_MAX\}$

• ISR ID 号的集合：$IID=\{TASK\_MAX+1..TASK\_MAX+ISR\_MAX\}$

• 资源 ID 号的集合：$RID=\{1..RESOURCE\_MAX\}$

• 标准资源 ID 号的集合：$RES\_STANDARD \subseteq RID$

• 内部资源 ID 号的集合：$RES\_INTERNAL \subseteq RID$

• 中断资源 ID 号的集合：$RES\_INTERRUPT \subseteq RID$

• 调度器资源的 ID：$schedRes \in RID$

本文要求 $RES\_STANDARD$，$RES\_INTERNAL$，$RES\_INTERRUPT$ 和 $\{schedRes\}$ 是互不相交的。

• 任务优先级的集合：$TP=\{1..PRIO\_TASK\_MAX\}$

• 中断优先级的集合：

$IP=\{PRIO\_TASK\_MAX+1..PRIO\_TASK\_MAX+PRIO\_INT\_MAX\}$

• 对于每个任务，函数 $taskInfo$ 实现从任务 ID 到其静态配置信息的映射，相关定义如式(21)所示：

$$\begin{array}{l} taskInfo: TID \rightarrow \textbf{taskInfoT} \\ \textbf{taskInfoT} \equiv (originPriTask, maxActive, non\text{-}preemptable, extended, inRes) \end{array} \tag{21}$$

$originPriTask \in TP$ 是任务的初始(静态分配的)优先级，$maxActive \in N$ 是最大的激活次数，$inRes \in RES\_INTERNAL \cup \{0\}$ 是其内部资源的 ID 号。在 $inRes=0$ 的情况下，该任务没有内部资源。

• 对于每个 ISR，函数 $originPriISR$ 实现从 ISR ID 到其初始优先级的映射，如式(22)所示：

$$originPriISR: IID \rightarrow IP \tag{22}$$

• 对每个资源，函数 $resOwners$ 实现从资源 ID 到共享该资源的任务或 ISR 集合的映射，如式(23)所示：

$$resOwners: RID \rightarrow (TID \cup IID)^* \tag{23}$$

关于动态配置状态，本文将自动机 $A_{ck}$ 的一个具体动态配置状态表示为 $c_{ck}$，其形式定义如式(24)所示：

$c_{ck} \equiv (ct, id, schedata, schedulable, taskConfig, resConfig, resourceList\_ISR, interruptMaskGlobal)$ (24)

其中：

- $ct \in TID$ 是当前执行任务的 ID 号；
- $id \in N$ 是嵌套中断服务的深度；
- $schedata \in$ **schedataT**：根据系统中任务的不同状态（运行态、就绪态、等待态或挂起态），它们被组织到 *schedata* 的各个队列或集合中，包括就绪队列 *rqueue* 和等待任务的集合 *waitingtasks*。**schedataT** 的定义如式(25)所示：

$$\textbf{schedataT} \equiv (rqueue \in (TP \rightarrow (TID \cup \{\perp\}))^*, waitingtasks \subseteq TID) \quad (25)$$

就绪任务被组织到不同优先级的就绪队列中，每个队列中的任务按 FIFO 顺序排列。当前运行任务位于其当前优先级的就绪队列。

- $schedulable \in \{0,1\}$ 用于记录在某个 ISR 执行期间调度条件已被满足，即在退出最外层的中断服务时操作系统会进行任务调度。
- *taskConfig*：从任务 ID 号到其动态信息的映射，相关定义如式(26)所示：

$$taskConfig: TID \rightarrow \textbf{taskConfigT}$$
$$\textbf{taskConfigT} \equiv (everExecuted, priCurrent, resourceList, curActiveNum, setEvent, waitEvent) \quad (26)$$

其中，$everExecuted \in \{0,1\}$ 表示任务是否曾经执行过，$priCurrent \in (TP \cup IP)$ 是任务的当前优先级，$curActiveNum \in N$ 是任务当前被激活实体的个数，$setEvent \in B^{32}$ 是任务当前被设置的事件向量，$waitEvent \in B^{32}$ 是任务正在等待的事件向量，$resourceList \in$ **resourceNodeT**$^*$ 是任务当前占有的标准资源栈，**resourceNodeT** 的定义如式(27)所示：

$$\textbf{resourceNodeT} \equiv (resId \in RES\_STANDARD, curPriority \in TP) \quad (27)$$

*curPriority* 是任务获得该资源后的新的“当前优先级”。

- *resConfig*：从资源 ID 号到其动态信息的映射，相关定义如式(28)所示：

$$resConfig: RID \rightarrow \textbf{resConfigT}$$
$$\textbf{resConfigT} \equiv (isUsed \in \{0,1\}) \quad (28)$$

*isUsed* 表示该资源是否被占用。

- $resourceList\_ISR \in$ **resIntNodeT**$^*$：被占用的中断资源的栈，**resIntNodeT** 的定义如式(29)所示：

$$\textbf{resIntNodeT} \equiv (resId \in RES\_INTERRUPT, interruptMask \in B^{32}) \quad (29)$$

*interruptMask* 是当该中断资源被占用后新的全局中断屏蔽码。

- $interruptMaskGlobal \in B^{32}$：系统当前的全局中断掩码。

### 3.3.2 变迁函数

系统服务的变迁函数表示为 $\delta_{API\_zzz}$ 的形式。以 API *ActivateTask*(*tid*) 为例，其变迁函数为 $\delta_{API\_ActivateTask}$。该函数将 ID 号为 *tid* 的任务以其初始优先级从挂起态迁移到就绪态。基本任务在其当前激活次数没有超过最大配置值时可被再次激活；当激活一个扩展任务时，其所有的事件均被清空。如果满足相应的调度条件，则操作系统可马上进行任务调度或者记录调度标志。$\delta_{API\_ActivateTask}$ 的形式定义如表 1 所列。

表 1 $\delta_{API\_ActivateTask}$ 的形式定义

$$(c'_{ck}, o_{ck}) = \delta_{API\_ActivateTask}(c_{ck}, tid) \equiv$$
if
  extended?(tid) $\wedge$ curActiveNum($c_{ck}$, tid) = 1
  $\vee \neg$extended?(tid) $\wedge$ curActiveNum($c_{ck}$, tid) = maxActive(tid) $\rightarrow$
    $o_{ck}$ = E_OS_LIMIT
  tid $\notin$ TID $\rightarrow$ $o_{ck}$ = E_OS_ID
  otherwise $\rightarrow$
    $c^1_{ck}$. taskConfig(tid). curActiveNum = curActiveNum($c_{ck}$, tid) + 1
    curActiveNum($c_{ck}$, tid) = 0 $\rightarrow$ $c^1_{ck}$. taskConfig(tid). everExecuted = 0
    extended?(tid) $\rightarrow$ $c^1_{ck}$. taskConfig(tid). setEvent = $0^{32}$
    $c^2_{ck} = \delta_{meta\_Activate}(c^1_{ck}, tid)$

$$c'_{ck} \begin{cases} \delta_{internal\_taskDispatch}(c^2_{ck}), & needSchedule?(c^2_{ck}) \\ c^2_{ck}[schedulable = 1], & needSchedule^*?(c^2_{ck}) \\ c^2_{ck}, & otherwise \end{cases}$$

    $o_{ck}$ = E_OK
fi

其中：

- *maxActive*(*tid*)，*curActiveNum*($c_{ck}$, *tid*) 分别用于访问任务 *tid* 的 *maxActive*，*curActiveNum* 部件，而 *extended*?(*tid*) 用于判定该任务是否是扩展任务。
- *needSchedule*$^*$?和 *needSchedule*?用于判定是否满足相应的调度条件。前者判定是否在中断处理过程中产生了调度需求，因而当退出最外层的中断处理时操作系统能够实施调度；而后者判定是否马上可以进行调度。两者均要求调度器没有被上锁、当前任务可被抢占、没有中断资源被占用、当前运行任务不是最高优先级的就绪任务。*needSchedule*$^*$?和 *needSchedule*?的定义分别如式(30)和式(31)所示。

$$needSchedule^*?(c_{ck}) \equiv \neg SchedulerLocked?(c_{ck}) \wedge preemptable?(c_{ck}.ct) \wedge noInterruptResouce?(c_{ck}) \wedge Tc(c_{ck}) \neq c_{ck}.ct \wedge InISR?(c_{ck}) \quad (30)$$

$$needSchedule?(c_{ck}) \equiv \neg SchedulerLocked?(c_{ck}) \wedge preemptable?(c_{ck}.ct) \wedge noInterruptResouce?(c_{ck}) \wedge Tc(c_{ck}) \neq c_{ck}.ct \wedge \neg InISR?(c_{ck}) \quad (31)$$

- $\delta_{meta\_Activate}$：形如 $\delta_{meta\_zzz}$ 的“元变迁”函数之一，其操作对象是操作系统进行任务管理及调度的核心：就绪任务队列 $c_{ck}.schedata.rqueue$ 和等待任务的集合 $c_{ck}.schedata.waitingtasks$。元变迁函数包括 $\delta_{meta\_Activate}$（激活任务），$\delta_{meta\_Schedule}$（调度任务），$\delta_{meta\_Terminate}$（终止任务），$\delta_{meta\_Preempt}$（抢占任务），$\delta_{meta\_Waiting}$（任务等待），$\delta_{meta\_Waken}$（唤醒任务），$\delta_{meta\_Lift}$（优先级提升）和 $\delta_{meta\_Decrease}$（优先级降低），它们会在不同的系统服务变迁函数中被调用。以 $\delta_{meta\_Activate}$ 元变迁为例，其定义为式(32)：

$$c'_{ck} = \delta_{meta\_Activate}(c_{ck}, tid) \equiv c'_{ck}.schedata.rqueue(originPri(tid)) = c_{ck}.schedata.rqueue(originPri(tid)) \circ tid \quad (32)$$

- $\delta_{internal\_taskDispatch}$：内部函数，即调用顺序内核的任务调度函数 *scheduler*。

## 4 OS 的实现正确性

操作系统的实现正确性与具体采用的证明方法及工具有关。本文使用不同的策略和工具验证操作系统的不同部分，即顺序内核和系统服务。在 VCC 中验证系统服务是因为其自动机的状态结构可直接映射至实现代码的相关数据结构，

并且其变迁函数都被实现为 C 函数。VCC 的注解用来描述数据结构的实现不变量，以及相关 C 函数的前置/后置条件等。相比在 Isabelle/HOL 中描述并推理这些程序的逻辑而言，其工作量将小很多，因为如果采用后者，则不得不对 C 中类型及内存的公理化过程(步骤)进行彻底展开[31]。

然而对于顺序内核，一方面其抽象模型的配置状态以不同的内存区域、栈空间以及栈帧为描述对象，且并未建立在 C 数据结构之上；另一方面其变迁函数牵涉到了目标处理器的 ISA 语义(包括汇编指令的操作语义和硬件响应异常时的动作)，而 VCC 是天生不支持这部分内容的。如果要在 VCC 中对顺序内核进行验证，则需要建立相应的仿真数据结构和仿真函数，这也将导致产生附加的工作量并引入可能的错误。因而考虑在 Isabelle/HOL 这一更为通用的逻辑工具中描述顺序内核的模型并定义更为低层的 C-IL 语义和汇编语义(涉及目标处理器的指令集体系架构)，以便对顺序内核实施完整的证明。

使用定理证明器 Isabelle/HOL 这一通用的逻辑工具证明顺序内核的正确性，需要建立一个证明的结构，即顺序内核的实现正确性定理，然后在该定理的框架下，通过不断地“精化”(refinement)，把对定理的归纳证明过程分解为对汇编语义或 C-IL 语义的推导过程。而对于系统服务的实现正确性，需要以 VCC 的注解式语言(annotation)描述每个系统服务变迁函数的规范，以及相关的不变量。因此需要重点研究系统服务的实现不变量。

### 4.1 顺序内核的实现正确性

#### 4.1.1 顺序内核的实现正确性定理

本文需要证明内核的实现相对于其规范的正确性。从所涉及语言的层次来看，可将 EAS 系统的计算过程看作是在 C 程序和汇编程序之间不断进行转换的过程，顺序内核的实现状态随着其 C 或汇编程序的执行而发生变化。然而最终，编译及汇编之后的实现代码是在目标体系架构的物理机器上执行，且 C 和汇编程序之间的切换必须保证在物理机器级配置状态的一致性。因此，顺序内核的实现正确性定理将围绕 EAS 系统的配置状态、顺序内核的实现状态、物理机器的配置状态等 3 个层次进行定义，如图 3 所示。

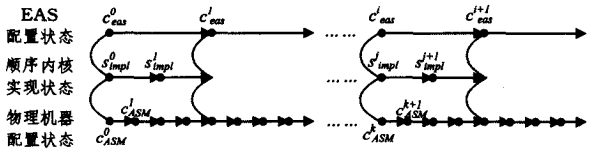


图 3 顺序内核正确性定理覆盖的状态层次

在定义具体的顺序内核实现正确性定理之前，先进行有关于物理机器(即汇编级机器)配置状态及其变迁函数的形式定义。

物理机器配置状态 $c_{ASM}$ 的定义如式(33)所示：

$$c_{ASM} \equiv (pc::\mathbb{N}, gpr::\mathbb{Z}^*, spr::\mathbb{Z}^*, m::\mathbb{N} \mapsto \mathbb{B}^8) \in C_{ASM} \tag{33}$$

$C_{ASM}$ 是物理机器所有配置状态的集合，作用在该状态集合上的变迁函数如式(34)所示：

$$\delta_{ASM}::C_{ASM} \times (\mathbb{B}^{32} \times \mathbb{B}) \mapsto C_{ASM} \tag{34}$$

$c'_{ASM} = \delta_{ASM}(c_{ASM}, eev)$ 是物理机器 $c_{ASM}$ 执行一个步骤后所到达的新的配置状态，它可能是一条指令的执行结果，或是硬件响应某个异常(包括外部中断请求)的动作。

另外，在定理中还用到了记录 EAS 程序编译信息的 $info_{eas}$，它主要包含各个代码段的基地址、各个栈的基地址和大小、每个编译/汇编后的 C 语句/指令的基地址等。

在此基础上，顺序内核实现正确性见定理 1。

**定理 1**(顺序内核实现正确性) 令 $\pi$ 为一个 EAS 系统的程序，$info_{eas}$ 是其编译信息，$c^0_{ASM}$ 和 $c^0_{eas}$ 分别为物理(汇编)机器和 EAS 系统的初始配置状态，$s^0_{impl}$ 为内核的初始实现状态。那么对于 EAS 系统及其外部事件的一个任意序列 $(c^0_{eas}, eev^0), (c^1_{eas}, eev^1), \cdots$ 且 $\forall i \in \mathbb{N}: c^{i+1}_{eas} = \delta_{eas}(c^i_{eas}, eev^i)$，均存在一个物理机器及相应外部事件的序列 $(c^0_{ASM}, eev^0), (c^1_{ASM}, eev^1), \cdots$ 且 $\forall j \in \mathbb{N}: c^{j+1}_{ASM} = \delta_{ASM}(c^j_{ASM}, eev^j)$，和一个内核实现状态的序列 $s^0_{impl}, s^1_{impl}, \cdots$ 且 $\forall k \in \mathbb{N}: \pi_{Kn}, \theta \vdash s^k_{impl} \rightarrow_{Kn} s^{k+1}_{impl}$。并且，对于 EAS 系统的任何序号为 $i$ 的步骤及其对应的配置状态 $c^i_{eas}$，存在映射函数 $t::\mathbb{N} \mapsto \mathbb{N}$ 和 $s::\mathbb{N} \mapsto \mathbb{N}$，使得有一个相应的物理机器配置状态 $c^{t(i)}_{ASM}$ 和一个内核实现状态 $s^{s(i)}_{impl}$，并使得仿真关系 $\sim_{EAS}(c^i_{eas}, c^{t(i)}_{ASM}, s^{s(i)}_{impl}, eev^j, \pi)$ 和实现不变量 $impl\_inv?(c^i_{eas}, info_{eas}, c^{t(i)}_{ASM}, s^{s(i)}_{impl})$ 成立。

定理 1 的形式描述如式(35)所示。

$$\begin{aligned}
&\exists info_{eas}.\ \forall (c^i_{eas})_i, \exists (c^j_{ASM})_j, \exists (s^k_{impl})_k.\\
&(\forall i,j,k \in \mathbb{N}.\ c^{j+1}_{ASM} = \delta_{ASM}(c^j_{ASM}, eev^j)\\
&\quad \wedge \pi_{Kn}, \theta \vdash s^k_{impl} \rightarrow_{Kn} s^{k+1}_{impl}\\
&\quad \wedge c^{i+1}_{eas} = \delta_{eas}(c^i_{eas}, eev^j))\\
&\wedge \exists s::\mathbb{N} \mapsto \mathbb{N}.\ \exists t::\mathbb{N} \mapsto \mathbb{N}.\\
&\quad (\cdot\ \forall i \in \mathbb{N}.\ \sim_{EAS}(c^i_{eas}, c^{t(i)}_{ASM}, s^{s(i)}_{impl}, eev^j, \pi)\\
&\quad \wedge impl_inv?(c^i_{eas}, info_{eas}, c^{t(i)}_{ASM}, s^{s(i)}_{impl}))
\end{aligned} \tag{35}$$

其中，$\pi_{Kn}, \theta \vdash s^k_{impl} \rightarrow_{Kn} s^{k+1}_{impl}$ 表示内核的一个执行步骤，它可能执行完成了 1 条指令或 C 语句，或者在用户执行时它是一个“空步骤”($s^{k+1}_{impl} = s^k_{impl}$)。

仿真关系 $\sim_{EAS}$ 和实现不变量 $impl\_inv?$ 是定理 1 的重要组成内容，下面将对它们的定义作进一步展开。

#### 4.1.2 仿真关系 $\sim_{EAS}$

在一个 EAS 系统的运行过程中，内核的实现状态随着 C 程序或汇编程序的执行而发生改变。仿真关系用于将上层的 EAS 配置状态 $c_{eas}$ 与低层的 C-IL 配置状态 $c_{C\_IL}$ 或汇编级配置状态 $c_{ASM}$ 通过内核的实现状态 $s_{impl}$ 关联起来。当计算在不同的程序语言之间转换时，需要构造相应语言层的配置，使得计算可以进展下去。总体的仿真关系 $\sim_{EAS}$ 是子关系 $\sim_{sr}$，$\sim_{ct}$，$\sim_{id}$，$\sim_{user}$ 和 $\sim_{kern}$ 的合取，如式(36)所示：

$$\begin{aligned}
&\sim_{EAS}::C_{EAS} \times C_{ASM} \times S_{IMPL} \times (\mathbb{B}^{32} \times \mathbb{B}) \times Prog_{EAS} \mapsto \mathbb{B}\\
&\sim_{EAS}(c_{eas}, c_{ASM}, s_{impl}, eev, \pi) \equiv \sim_{sr}(c_{eas}, c_{ASM}) \wedge \sim_{ct}(c_{eas}, c_{ASM}, s_{impl}) \wedge \sim_{id}(c_{eas}, c_{ASM}, s_{impl}) \wedge \sim_{user}(c_{eas}, eev, \pi) \wedge \sim_{kern}(c_{eas}, eev, \pi)
\end{aligned} \tag{36}$$

$\sim_{sr}::C_{EAS} \times C_{ASM} \mapsto \mathbb{B}$ 将抽象配置状态的部件 $c_{eas}.sr$ 与机器的外部中断使能状态关联起来，在典型的汽车电子微处理器 MPC563X 上有式(37)：

$$\begin{aligned}
&\sim_{sr}(c_{eas}, c_{ASM}) \equiv (c_{eas}.sr = INT_ENABLED \rightarrow ee?(c_{ASM}.spr[msr])) \wedge (c_{eas}.sr = INT_DISABLED \rightarrow \neg ee?(c_{ASM}.spr[msr]))
\end{aligned} \tag{37}$$

这里通过判定 $ee?$ 指示外部中断是否使能。

同样，$\sim_{ct}(c_{eas}, c_{ASM}, s_{impl})$将当前任务 $c_{eas}.ct$ 与存储在 $c_{ASM}$ 的某个内存地址中的值关联起来，该地址对应内核实现的全局变量 $s_{impl}.ct$，$\sim_{id}(c_{eas}, c_{ASM}, s_{impl})$将抽象模型的中断嵌套深度 $id(c_{eas})$ 与存储在 $c_{ASM}$ 的某个内存地址中的值关联起来，该地址对应内核实现的全局变量 $s_{impl}.id$。

子关系 $\sim_{user}$ 和 $\sim_{kern}$ 分别被定义来构造 EAS 系统中 2 类主要的执行体（并发执行体（任务/ISR/系统服务）和顺序内核）的 C-IL 配置或汇编配置。$\sim_{user}$ 关系的定义如式（38）所示，其中 $c^{e}_{C\_IL}$ 表示当前并发执行体执行时的 C-IL 配置，需要计算其关联的特定内存区域及栈：

$$\sim_{user}(c_{eas}, eev, \pi) \equiv internalStep?(c_{eas}, eev, \pi) \rightarrow \exists\, c^{e}_{C\_IL}.\ c^{e}_{C\_IL} \in C_{C\_IL} \wedge c^{e}_{C\_IL}.\mathcal{M} = mregion_{curr}(c_{eas}) \wedge c^{e}_{C\_IL}.s = stack_{curr}(c_{eas}) \tag{38}$$

顺序内核的配置的构造关系如式（39）所示：

$$\sim_{kern}(c_{eas}, eev, \pi) \equiv (isKernCIL?(c_{eas}, eev, \pi) \rightarrow \exists\, c_{kstart}.\ c_{kstart} \in C_{C\_IL} \wedge c_{kstart}.\mathcal{M} = c_{eas}.mr(0) \wedge c_{kstart}.s = stack_{curr}(c_{eas})) \wedge (isKernASM?(c_{eas}, eev, \pi) \rightarrow \exists\, c_{kstart}.\ c_{kstart} \in C_{ASM} \wedge c_{kstart} = \delta_{hard}(\mathcal{R}_{cc}(c_{eas}), eev)) \tag{39}$$

每当顺序内核被触发，都需要为它构造一个新的初始配置状态，因为根据具体的触发原因，它总是从某个变迁函数的开始而执行的。内核的程序包括 C-IL 程序和汇编程序两个部分。$\sim_{kern}$ 关系的第一个子句表明，如果内核从一个 C-IL 函数开始执行（$isKernCIL?(c_{eas}, eev, \pi)$），其 C-IL 配置中的全局内存区域始终是 $c_{eas}.mr(0)$，而其起始时刻的工作栈取决于当前的并发执行体。第二个子句表明，如果内核始于某个汇编程序（$isKernASM?(c_{eas}, eev, \pi)$），其初始配置状态为汇编配置（$c_{kstart} \in C_{ASM}$），其中的寄存器内容受 2 个因素影响：1）有关寄存器使用的编译规则：当把一个 C-IL 配置转换为汇编配置时，它将决定寄存器的值。本文把与编译规则一致的寄存器值的选择定义为函数 $\mathcal{R}_{cc}::C_{EAS} \rightarrow C_{ASM}$，其中相关的 C-IL 配置可以通过当前的 EAS 配置 $c_{eas}$ 计算而获得；2）当处理器响应异常时，某些寄存器如程序计数器、机器状态寄存器、某些用于保存返回地址及机器状态的特殊寄存器等因为硬件的响应动作而发生变化。函数 $\delta_{hard}(c_{ASM}, eev)$ 用于描述这些寄存器的变化。

### 4.1.3 顺序内核的实现不变量 *impl_inv*?

1）顺序内核的实现不变量的组织

实现不变量要求在内核或并发执行体执行期间，给定的 EAS 配置状态及内核实现满足一定的合规性（well-formedness）要求。图 4 展示了内核实现不变量的组织结构，其中：

- $consis^{control}$ 表示 EAS 系统的控制一致性，由程序流控制一致性关系 $consis^{progCtrl}$、数据控制一致性关系 $consis^{dataCtrl}$、PID 控制一致性关系 $consis^{pidCtrl}$ 和模式控制一致性关系 $consis^{modeCtrl}$ 等 4 个子不变量合取而成。$consis^{progCtrl}$ 用于判定 pc 寄存器的当前值以及保存的返回地址的合规性，类似地，$consis^{dataCtrl}$ 用于判定当前栈指针寄存器以及保存的栈指针内容的合规性。$consis^{modeCtrl}$ 用于保证 OS（系统调用接口函数除外）及可信应用执行于特权模式而非可信应用执行于用户模式。为了分离不同的非可信应用，内核的实现利用了目标处理器（e200z3 Power Architecture）MMU 的特殊寄存器 pid0 来控制对每个应用内存页面的访问，因此需要 $consis^{pidCtrl}$。
- 代码一致性 $consis^{code}$ 要求编译后的 EAS 程序代码位于从相应的起始地址开始的物理机器内存区域中。
- 对于存放常量及全局变量的内存空间，存储一致性 $consis^{mem}$ 要求除了分配给栈的空间外，每个数据内存页的其余地址空间的内容针对 C-IL 机器和汇编机器而言是相同的。
- 一致性关系 $consis^{stack}$ 用于判定在普通的函数调用帧中存储的参数、局部变量以及返回值地址的正确性；一致性关系 $consis^{INT}$ 和 $consis^{SCI}$ 分别用于判定类型为 $frame_{INT}$ 和 $frame_{SCI}$ 的特殊内核帧所存储内容的正确性。

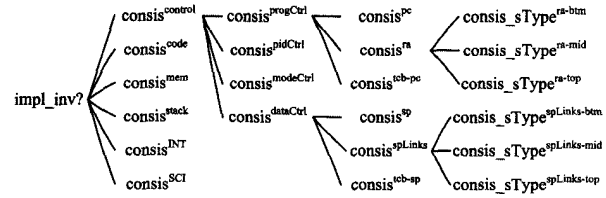


图 4 顺序内核的实现不变量的组织

2）实现不变量 *impl_inv*? 的顶层定义

顺序内核的实现不变量 *impl_inv*? 的顶层定义是图 4 中各个第一级子不变量的合取，它定义了在并发执行体及内核的执行过程中需要被满足的各项一致性关系，如式（40）所示：

$$impl\_inv?(c_{eas}, info_{eas}, c_{ASM}, s_{impl}) \equiv consis^{code}(info_{eas}, c_{ASM}) \wedge consis^{mem}(c_{eas}, info_{eas}, c_{ASM}) \wedge consis^{stack}(c_{eas}, info_{eas}, c_{ASM}) \wedge (user\text{-}step?(c_{ASM}, info_{eas}) \rightarrow consis^{control}(c_{eas}, info_{eas}, c_{ASM}, s_{impl}) \wedge consis^{INT}(c_{eas}, c_{ASM}) \wedge consis^{SCI}(c_{eas}, c_{ASM})) \wedge (kernel\text{-}step?(c_{ASM}, info_{eas}) \rightarrow consis^{control*}(c_{eas}, info_{eas}, c_{ASM}, s_{impl}) \wedge consis^{INT*}(c_{eas}, c_{ASM}) \wedge consis^{SCI*}(c_{eas}, c_{ASM})) \tag{40}$$

并发执行体可被看作是顺序内核的用户，用户步骤和内核步骤分别通过判定 *user-step*? 和 *kernel-step*? 进行区分，它们将 pc 寄存器的当前值与编译后的用户或内核程序的地址范围相比较。式（40）中前 3 项的合取表明无论是用户还是内核在执行，一致性关系 $consis^{code}$、$consis^{mem}$ 和 $consis^{stack}$ 都是要被满足的。第 4 个大的合取项表明当用户执行时（满足判定 *user-step*?），$consis^{control}$、$consis^{INT}$ 和 $consis^{SCI}$ 需要被满足。第 5 个大的合取项表明当内核执行时（满足判定 *kernel-step*?），$consis^{control*}$、$consis^{INT*}$ 和 $consis^{SCI*}$ 需要被满足，它们是分别对应 $consis^{control}$、$consis^{INT}$ 和 $consis^{SCI}$ 的一组弱一致性关系，因为当内核执行时，$consis^{control}$、$consis^{INT}$ 和 $consis^{SCI}$ 的条件太强了，它们所涉及的一些内容正处于被构造的过程当中。$consis^{control*}$、$consis^{INT*}$ 和 $consis^{SCI*}$ 表示除了那些正在被当前内核函数构造的帧以及正在被改变的全局变量外，对于系统配置的其余部分，相关判定仍然是被满足的。

3）子不变量的形式定义

对于 $consis^{control}$ 的每个子不变量，有关的信息可能被维护于 CPU 核心、栈帧或 TCB 中，因此每个子不变量又都是 3 个子关系的合取。例如图 4 中所示，程序流控制一致性关系 $consis^{progCtrl}$ 是 $consis^{pc}$、$consis^{ra}$ 和 $consis^{tcb\text{-}pc}$ 的合取，它们分别关注 CPU 核心中的 pc 寄存器、保存在栈帧中的返回地址，以及保存在 TCB 中的返回地址。对于保存在栈帧中的信息，进一步区分每个栈的底帧、中间帧以及顶帧的不同情况，并分别表示为一致性关系 $consis\_sType^{xx\text{-}btm}$、$consis\_sType^{xx\text{-}mid}$ 和 $consis\_sType^{xx\text{-}top}$。$sType \in \{S\_IUS, S\_TUS, S\_ISS, S\_$

$TSS\}$表示 $c_{eas}$ 中 4 种栈类型的集合。以中断用户栈($sType=S\_IUS$)的顶帧为例,其一致性关系 $consis\_S\_IUS^{ra\text{-}top}$ 的定义如式(41)所示:

$$consis\_S\_IUS^{ra\text{-}top}(c_{eas},info_{eas},c_{ASM})\equiv\forall i\in NIID\wedge c_{eas}.ius(i)\neq[]\wedge((topisr?(c_{eas},i)\wedge c_{eas}.asv(i)=0.\rightarrow c_{ASM}.pc=info_{eas}.code_{ia}(\mathbf{hd}(c_{eas}.ius(i)).f,\mathbf{hd}(c_{eas}.ius(i)).loc))\vee(c_{eas}.asv(i)=1.\rightarrow\exists!j,0\leqslant j<|c_{eas}.iss|-1\wedge c_{eas}.iss[j]=fr_{SCI}\in frame_{SCI}.fr_{SCI}.sra=info_{eas}.code_{ia}(\mathbf{hd}(c_{eas}.ius(i)).f,\mathbf{hd}(c_{eas}.ius(i)).loc)\wedge\mathbf{hd}(c_{eas}.ius(i)).f\in OSAPI\wedge c_{eas}.iss[j+1].f\in SS)\vee(\neg topisr?(c_{eas},i)\wedge c_{eas}.asv(i)=0.\rightarrow\exists!j,0\leqslant j<|c_{eas}.iss|\wedge c_{eas}.iss[j]=fr_{INT}\in frame_{INT}.fr_{INT}.spr[srr0]\in info_{eas}.code_{ar}(\mathbf{hd}(c_{eas}.ius(i)).f,\mathbf{hd}(c_{eas}.ius(i)).loc))) \quad (41)$$

式(41)的含义如图 5 所示,即 $consis\_S\_IUS^{ra\text{-}top}$ 要求一个非空的 $c_{eas}.ius(i)$栈的顶帧是如下 3 种情况之一:

a)ISR $i$ 是当前的执行体,因为它处于嵌套 ISR 的最内层且没有调用系统服务 API($topisr?(c_{eas},i)\wedge c_{eas}.asv(i)=0$),因此 $c_{eas}.ius(i)$的顶帧是当前的工作帧。

b)ISR $i$ 有调用系统服务 API($c_{eas}.asv(i)=1$),因此其顶帧对应的是那个被调用的 API 函数($\mathbf{hd}(c_{eas}.ius(i)).f\in OSAPI$,$OSAPI$ 是提供给应用的 OS API 函数名称的集合)。$c_{eas}.iss$ 栈上的一个相关的 $frame_{SCI}$ 帧 $fr_{SCI}$ 的 $sra$ 部件中保存了被调用 API 函数的返回地址,而 $fr_{SCI}$ 的后继帧则与其相应的系统服务函数相关联($c_{eas}.iss[j+1].f\in SS$,$SS$ 是被内核的接口函数调用的系统服务函数名称的集合)。

c)ISR $i$ 已被中断且没有调用系统服务 API($\neg topisr?(c_{eas},i)\wedge c_{eas}.asv(i)=0$),$c_{eas}.iss$ 上一个相应的 $frame_{INT}$ 帧 $fr_{INT}$ 保存了返回地址(中断被响应时的后一条指令)。

式(41)中的 $info_{eas}.code_{ia}:(F_{name}\times N)\mapsto N$ 用于映射函数名和位置到 EAS 程序相应语句/指令被编译/汇编之后的基地址,而 $info_{eas}.code_{ar}:(F_{name}\times N)\mapsto[N:N]$用于映射函数名和位置到相应语句被编译之后的地址范围。

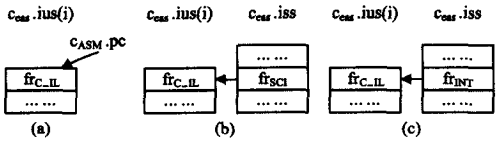


图 5 中断用户栈顶帧的一致性关系 $consis\_S\_IUS^{ra\text{-}top}$ 的含义

### 4.2 系统服务的实现正确性

如前所述,对于系统服务的实现正确性,主要关注其实现不变量的定义。系统服务的实现不变量也是一组子不变量的合取,如式(42)所示:

$$Inv\_concOS(c_{ck})\equiv inv\_ct(c_{ck})\wedge inv\_ActiveNum(c_{ck})\wedge inv\_taskStates(c_{ck})\wedge inv\_resOwners\wedge inv\_readyQueues(c_{ck})\wedge inv\_curPriTask(c_{ck})\wedge inv\_ceilPri\wedge inv\_intMask(c_{ck}) \quad (42)$$

其中,$inv_ct(c_{ck})$为判定当前任务合规性的不变量,$inv_ActiveNum(c_{ck})$为判定任务激活数量合规性的不变量,$inv_taskStates(c_{ck})$为判定任务状态合规性的不变量,$inv_resOwners$为判定资源优先级范围及其共享者合规性的不变量,$inv_readyQueues(c_{ck})$为判定就绪队列合规性的不变量,$inv_curPriTask(c_{ck})$为判定任务当前优先级合规性的不变量,$inv\_ceilPri$ 为判定资源的天花板优先级合规性的不变量,$inv_intMask(c_{ck})$为判定中断掩码合规性的不变量。

以 $inv_readyQueues(c_{ck})$为例,它要求:任何位于优先级为 $pri$ 的就绪队列中的任务,或者其初始优先级等于 $pri$,或者它已获取了一个天花板优先级为 $pri$ 的资源 $rid$。对于后者,该任务一定位于某个就绪队列的头节点,该队列具有该任务已获得资源中最高的天花板优先级。$inv_readyQueues(c_{ck})$的形式定义如式(43)所示:

$$inv\_readyQueues(c_{ck})\equiv\forall pri\in TP,\forall tid\in c_{ck}.schedata.rqueue(pri).(originPri(tid)=pri)\vee((tid=\mathbf{hd}(c_{ck}.schedata.rqueue(pri)))\wedge((\exists rid\in RID,\exists i\in[1:nOccupiedRes(c_{ck},tid)].(ceilPri(rid)=pri)\wedge(ceilPri(rid)=cPeakPri(c_{ck},tid))\wedge(rid=((c_{ck}.taskConfig(tid).resourceList)!i).resId))\vee(inresPriority(tid)=pri))) \quad (43)$$

其中,$nOccupiedRes(c_{ck},tid)$用于计算已被任务 $tid$ 占有的标准资源的数量;$cPeakPri(c_{ck},tid)$返回任务 $tid$ 的标准资源链的当前头节点所记录的峰值优先级,$inresPriority(tid)$返回该任务可能享有的内部资源的天花板优先级。

## 5 形式化验证

### 5.1 验证工作的总体流程

图 6 展示了采用 Isabelle/HOL 和 VCC 这 2 种工具环境对 eAutoOS 实施形式化验证工作的总体流程。

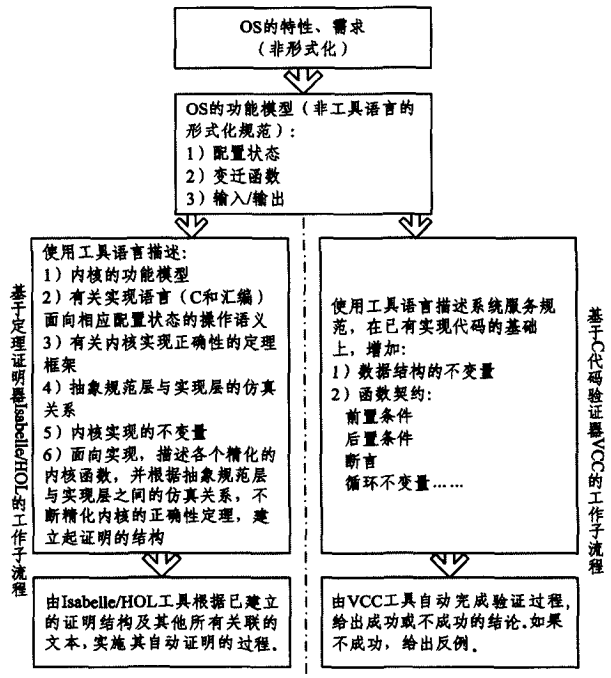


图 6 eAutoOS 形式化验证的总体流程

在 Isabelle/HOL 中,对定理 1 的归纳证明从步骤 0 开始,此时有一组相关的初始配置 $c_{eas}^{0}$,$c_{ASM}^{t(0)}$ 和 $s_{impl}^{s(0)}$,它们是由复位信号触发的初始变迁所建立的,并被定义为归纳过程的"基本情况"。在从步骤 $i$ 到 $i+1$ 的归纳过程中,对 EAS 系统的具体变迁函数进行情况区分。对内核的每个函数,归纳的步骤针对每一条即将执行的 C-IL 语句或汇编指令,根据它们作用于相应配置状态上的操作语义进行推导。

作为示例,此处给出 EAS 的顶级变迁函数及 $\delta_{eas\_IDP1}$ 变迁函数在 Isabelle 中的形式描述,分别如表 2 和表 3 所列。

表 2　顶级 EAS 变迁函数的 Isabelle 描述

```
δeas ceas eev≡
  if reset eev then δeas_init
  else if memException ceas eev then δeas_me ceas
      else if intRequest ceas eev then δeas_IDP1 ceas eev
        else
            if rfuisr ceas eev π then δeas_IDP2 ceas
            if apiCall ceas eev π then δeas_SCI1 ceas
            if rfsc ceas eev π then δeas_SCI2 ceas
            if scheduler ceas eev π then δeas_SCHD ceas
            if ¬(rfuisr ceas eev π∨apiCall ceas eev π∨rfsc ceas eev π
                ∨scheduler ceas eev π) then δeas_internal ceas
```

表 3　$\delta_{eas\_IDP1}$ 变迁函数的 Isabelle 描述

```
δeas_IDP1 ceas eev ≡
  if intRequest ceas eev
  then let iid=il eev;cC-IL=(|M=mregion_curr ceas,stack=stack_curr ceas
  |);
       ceas_2=ceas(| tss :=if (id ceas)=0
                then λf. if ctx_int_frame ceas f
                     then ceas.tss(ceas.ct ↦ f @ ceas.tss ceas.ct)
                     else ceas.tss
                else ceas.tss,
            iss := if (id ceas) ≠ 0
                then λf. if ctx_int_frame ceas f
                     then f @ ceas.iss
                     else ceas.iss
                else ceas.iss |)
    in ceas_2(| aisrList := iid@ ceas_2.aisrList,
            ius := if (imode iid)=M_USR
                then λ f. if isr_frame π. πC-IL θ (fisr iid) f
                     then ceas_2.ius(iid ↦f @ ceas_2.ius iid)
                     else ceas_2.ius
                else ceas_2.ius,
            iss:=if (imode iid)=M_SYS
                then λf. if isr_frame π. πC-IL θ (fisr iid) f
                     then f @ ceas_2.iss
                     else ceas_2.iss
                else ceas_2.iss |)
  else ceas
```

表 4 为在 VCC 中系统服务函数 *ActivateTask* 的注解代码示例。

表 4　*ActivateTask* 函数的 VCC 注解代码

```
_(requires\thread_local_array(taskInfo,TASK_MAX))
_(requires\thread_local_array(taskConfig,TASK_MAX))
_(requires\thread_local(&schedulable))
_(ensures\result==E_OS_ID==>(tid>=TASK_MAX &&. osAPIPa-
ram1==tid))
_(ensures\result==E_OS_LIMIT==>(
        ((taskInfo[tid].extended==1==>
            \old(taskConfig[tid].curActiveNum)==1)||
        (taskInfo[tid].extended==0==>\old(taskConfig[tid].curAc-
        tiveNum)==taskInfo[tid].maxActive))&&.
            osAPIParam1==tid))
_(ensures\result==E_OK==>(
        (taskConfig[tid].curActiveNum==
            \old(taskConfig[tid].curActiveNum)+1)&&.
        (\old(taskConfig[tid].curActiveNum)==0
            ==>taskConfig[tid].everExecuted==0)&&.
        (taskInfo[tid].extended==1==>
            taskConfig[tid].setEvent==(u32)0)&&.
        (AddaReadyBlock==\true)&&.
        ((needSchedule==\true==>task_Dispatch==\true)||
        (needScheduleOnExitingISRs==\true==>schedulable==1)
)))
```

在验证工具 VCC 和 Isabelle/HOL 的混合运用方面，Alkassar 等人的工作[31,37]是一个典型示范。他们在 VCC 中验证程序的功能，而在 Isabelle/HOL 中进行关键数学属性的逻辑推理。VCC 和 Isabelle/HOL 中的规范是针对同一个验证对象(即“检查程序”)的，只是它们具有不同的抽象(或精化)程度，因而 Alkassar 等人通过一个统一的二阶逻辑来连接在这两个工具环境中的工作内容。然而对于 eAutoOS，VCC 和 Isabelle/HOL 中的规范针对的是该 OS 的不同部分，涉及不同的配置状态视图，彼此更加独立。对于两者的逻辑连接只需做很少的工作，因为 OS 的这 2 个部分仅仅使用了少量相同的配置部件，如当前任务 *ct*、嵌套中断的深度 *id*，以及一些实现数据结构如调度相关的 *schedata*、记录可调度状态的 *schedulable* 等，只要保证在 2 个工具环境中对于这些配置部件和数据结构的论域是一致的。

### 5.2　验证结果分析

eAutoOS 系统服务相关 C 代码共计约 5KLOC(包括. c 及. h 文件共计 29 个，涉及内部函数 57 个，API 函数 60 个，数据结构定义 27 个)，相关的 VCC 注解代码约 15.5KLOC(包括数据不变量、函数契约、ghost 代码和断言)。在一个 Intel(R) Core(TM) i5 2.50GHz 双核 CPU 机器上的验证时间约为 2.78h。系统服务建模、VCC 注解开发及其他验证相关工作的总体工作量约为 2 个人每年。图 7 为 eAutoOS 的核心关键数据结构之一 T_OSEK_TASK_ReadyTaskTableItem 的验证实例图，输出结果中的方括号内为验证所消耗的机器时间(1.56s)。

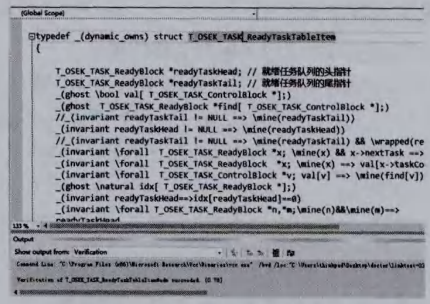


图 7　eAutoOS 数据结构在 VCC 中的验证实例

eAutoOS 顺序内核相关代码涉及 20 个 C 及汇编实现文件，总代码行数约为 2.9KLOC，主要包括 105 个函数及 2 个结构定义。在 Isabelle/HOL 中对应的模型、规范的描述，以及实现的转化代码合计约为 16KLOC，证明代码约为 55KLOC，约 3 个人每年的工作量。因此从上述数据对比中可以分析得出，针对相同规模的实现代码，在 VCC 中进行验证的总体工作量要低于在 Isabelle/HOL 环境中的工作量。

**结束语**　通过本文的工作，我们获得了一个经形式化验证其正确性的、面向汽车工业应用需求的、具备安全相关特性的嵌入式实时操作系统。更为重要的是，在面向此类嵌入式操作系统的形式化工作方面，取得了如下的经验和成果：(1)分层、分部件构建 OS 模型的技术。面向操作系统处理对象的不同方面，建立不同的配置状态模型及其配套的变迁函数，尤其通过扩展 C-IL/CC-IL 的配置状态模型，使得目标 OS 满足存储隔离保护的需求从而在抽象模型中被体现和关注。(2)根据操作系统不同层次或部件的模型特性以及有关实现语言的情况，选择适宜的工具进行形式化验证工作，积累了定理证明器 Isabelle/HOL 及程序验证工具 VCC 的组合应用经验：利用了 VCC 支持并发 C 程序验证的天然特性，以及 Isa-

belle/HOL 逻辑系统的通用性，同时被验证对象两个部分的配置状态的分离又解耦了两个工具环境之间的逻辑连接，降低了工作量。本文工作的结果表明，这样的工具组合使用是可行且有成效的。

## 参 考 文 献


[1] ISO. ISO 26262-6 Road vehicles—Functional Safety—Part 6: Product development at the software level [S/OL]. http://www.iso.org/iso/: ISO, 2011

[2] AUTOSAR GbR. Specification of Operating System V4.1.0 R4.0 Rev 2 [S/OL]. http://www.autosar.org/: AUTOSAR GbR, 2011

[3] Klein G. Operating system verification—an overview[J]. Sadhana, 2009, 34(1): 27-69

[4] 钱振江，刘苇，黄皓. 操作系统形式化设计与验证综述[J]. 计算机工程，2012，38(11)：234-238
Qian Zhen-jiang, Liu Wei, Huang Hao. Survey of formal design and verification for operating system [J]. Computer Engineering, 2012, 38(11): 234-238

[5] Elkaduwe D, Klein G, Elphinstone K. Verified protection model of the seL4 microkernel [M] // Shankar N, Woodcock J, eds. Verified Software: Theories, Tools, Experiments. Springer Berlin Heidelberg, 2008: 99-114

[6] Heise G, Elphinstone K, Kuz I, et al. Towards trustworthy computing systems: taking microkernels to the next level[J]. ACM SIGOPS Operating Systems Review, 2007, 41(4): 3-11

[7] Klein G, Elphinstone K, Heiser G, et al. seL4: Formal verification of an OS kernel [C] // Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, 2009. Big Sky, MT, USA: ACM, 2009: 207-220

[8] Schirmer N. Verification of sequential imperative programs in Isabelle-HOL[D]. Munich: Technical University Munich, 2006

[9] Rieden T I D. Verified linking for modular kernel verification [D]. Saarbrucken: Saarland University, 2009

[10] Gargano M, Hillebrand M, Leinenbach D, et al. On the correctness of operating system kernels[M] // Hurd J, Melham T, eds. Theorem Proving in Higher Order Logics. Springer Berlin Heidelberg, 2005: 1-16

[11] Rieden T I D, Tsyban A. CVM—a verified framework for microkernel programmers[J]. Electronic Notes in Theoretical Computer Science, 2008, 217: 151-168

[12] Tsyban A. Formal Verification of a Framework for Microkernel Programmers[D]. Saarbrucken: Saarland University, 2009

[13] Daum M, Dörrenbächer J, Wolff B. Proving fairness and implementation correctness of a microkernel scheduler[J]. Journal of Automated Reasoning, 2009, 42(2-4): 349-388

[14] Dörrenbächer J. Formal specification and verification of a microkernel[D]. Saarbrucken: Saarland University, 2010

[15] Schmidt M. Formal verification of a small real-time operating system[D]. Saarbrucken: Saarland University, 2011

[16] Bogan S. Formal specification of a simple operating system[D]. Saarbrucken: Saarland University, 2008

[17] Cohen E, Paul W, Schmaltz S. Theory of multi core hypervisor verification[M] // van Emde Boas P, Groen F, Italiano G, et al, eds. SOFSEM 2013: Theory and Practice of Computer Science. Springer Berlin Heidelberg, 2013: 1-27

[18] Schmaltz S. Towards the pervasive formal verification of multi-core operating systems and hypervisors implemented in C [D]. Saarbrucken: Saarland University, 2013

[19] Baumann C, Bormer T, Blasum H, et al. Proving memory separation in a microkernel by code level verification[C] // 2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2011. IEEE, 2011: 25-32

[20] 陈超超，曾庆凯. 采用 SPIN 的 L4 内存管理形式化验证[J]. 计算机工程，2009，35(11)：131-133
Chen Chao-chao, Zeng Qing-kai. Formal verification of L4 memory management using SPIN [J]. Computer Engineering, 2009, 35(11): 131-133

[21] 钱振江，刘苇，黄皓. 操作系统对象语义模型(OSOSM)及形式化验证[J]. 计算机研究与发展，2012，49(12)：2702-2712
Qian Zhen-jiang, Liu Wei, Huang Hao. OSOSM: operating system object semantics model and formal verification [J]. Journal of Computer Research and Development, 2012, 49(12): 2702-2712

[22] 李康杰，钱振江，黄皓. 微内核中断机制的形式化设计与验证[J]. 计算机科学，2013，40(3)：197-200，205
Li Kang-jie, Qian Zhen-jiang, Huang Hao. Formal design and verification of interrupt mechanism based on microkernel [J]. Computer Science, 2013, 40(3): 197-200, 205

[23] 钱振江. 安全操作系统形式化设计与验证方法研究[D]. 南京：南京大学，2013
Qian Zhen-jiang. Research on methodology of Formal design and verification for security operating system [D]. Nanjing: Nanjing University, 2013

[24] 程亮. 基于模型检测的安全操作系统验证方法研究[D]. 合肥：中国科学技术大学，2009
Cheng Liang. Research on verification of secure operating system based on model checking [D]. Hefei: University of Science and Technology of China, 2009

[25] 吴丹，刘芳，戴葵，等. Linux 中 System V 进程通信机制安全性形式化验证[J]. 计算机工程与科学，2002，24(2)：13-18
Wu Dan, Liu Fang, Dai Kui, et al. The formal verification of safety in System V of Linux [J]. Computer Engineering & Science, 2002, 24(2): 13-18

[26] 李斌. 基于 B 的安全操作系统原型的形式化分析和验证[D]. 昆明：云南大学，2011
Li Bin. Formal analysis and verification of secure operating system prototype based on B [D]. Kunming: Yunnan University, 2011

[27] 张忠秋，董云卫，张雨，等. 基于 Coq 的微内核操作系统程序验证方法研究[J]. 计算机测量与控制，2011，19(8)：1939-1942
Zhang Zhong-qiu, Dong Yun-wei, Zhang Yu, et al. Research on microkernel operating system program verification based on Coq [J]. Computer Measurement & Control, 2011, 19(8): 1939-1942

[28] 顾飞. 共享内存 IPC 机制的形式化验证与实现[D]. 兰州：兰州大学，2012
Gu Fei. Formal verification and implementation of shared memory IPC mechanism [D]. Lanzhou: Lanzhou University, 2012

[29] Shi Jian-qi, Zhu Hui-biao, He Ji-feng, et al. ORIENTAIS: Formal verified OSEK/VDX real-time operating system[C] // IEEE International Conference on Engineering of Complex Computer Systems, 2012. Los Alamitos, CA, USA: IEEE Computer Society, 2012: 293-301

[30] 彭云辉. 基于 AUTOSAR 的汽车电子操作系统及其应用的建模与分析[D]. 上海:华东师范大学,2014
Peng Yun-hui. Modeling and analysis of AUTOSAR OS and application [D]. Shanghai: East China Normal University, 2014

[31] Alkassar E, Böhme S, Mehlhorn K, et al. Verification of certifying computations[M]// Gopalakrishnan G, Qadeer S, eds. Computer Aided Verification. Springer Berlin Heidelberg, 2011: 67-82

[32] Alkassar E, Hillebrand M A, Paul W J, et al. Automated verification of a small hypervisor[M]// Leavens G, O'Hearn P, Rajamani S, eds. Verified Software: Theories, Tools, Experiments. Springer Berlin Heidelberg, 2010: 40-54

[33] Shadrin A. Mixed low-and high level programming language semantics and automated verification of a small hypervisor[D]. Saarbrucken: Saarland University, 2012

[34] The OSEK/VDX Group. OSEK/VDX Operating System specification Version 2. 2. 3[S/OL]. http://www. osek-vdx. org/: The OSEK/VDX Group, 2005

[35] AUTOSAR GbR. Technical Safety Concept Status Report V1. 0. 0 R4. 0 Rev 1[S/OL]. http://www. autosar. org/: AUTOSAR GbR, 2009

[36] 陈丽蓉,燕立明,罗蕾. 汽车电子嵌入式操作系统的隔离保护机制[J]. 电子科技大学学报,2014,43(3):450-456
Chen Li-rong, Yan Li-ming, Luo Lei. An isolation and protection mechanism of automotive electronic embedded operating system [J]. Journal of University of Electronic Science and Technology of China, 2014, 43(3): 450-456

[37] Alkassar E, Böhme S, Mehlhorn K, et al. A Framework for the Verification of Certifying Computations[J]. Journal of Automated Reasoning, 2014, 52(3): 241-273

(上接第 193 页)

存其中一个因子,KMC 保留另一个因子,主密钥的更新依赖 KMC 保留因子的更新,使得当有节点加入或退出时,合法成员的秘密解密密钥保持不变,减少了密钥更新时延。在安全性上,IKMS 方案支持前向/后向安全性,且无需安全信道的支持,具有比 OMEDP 更高的安全性。综合上述,IKMS 方案适合对密钥更新延时要求严格的动态网络。

## 参 考 文 献

[1] Akyildiz I F, Xudong W. A survey on wireless mesh networks [J]. IEEE Communications Magazine, 2005, 43(9): 23-30

[2] Yihchun H, Perrig A. A survey of secure wireless ad hoc routing [J]. IEEE Security and Privacy, 2004, 2(3): 28-39

[3] 李先贤,怀进鹏,刘旭东. 群密钥分配的动态安全性及其方案[J]. 计算机学报,2002,25(4):337-336
Li Xian-xian, Huai Jin-peng, Liu Xu-dong. Dynamic Security of group key Distribution and its Solutions[J]. Chinese Journal of Computers, 2002, 25(4): 337-345

[4] Johann M, Dawoud D, Stephen M. A survey on peer-to-peer key management for mobile ad hoc networks [J]. ACM Computing Surveys, 2007, 39(1): 1-46

[5] Yacine C, Hamida S. Group Key Management Protocols: A Novel Taxonomy [J]. International Journal of Information Technology, 2005, 2(2): 105-119

[6] Steiner M, Tsudik G, Waidner M. Diffie-Hellman Key Agreement Protocol with Key Confirmation [C]// Proceedings of Indocrypt 2000. LNCS 1977, Springer-Verlag, 2000: 237-249

[7] Burmester M, Desmedt Y. A Secure and Efficient Conference Key Distribution System [C]// Proceedings of Eurocrypt 1994. LNCS 950, Spring-Verlag, 1995: 275-286

[8] Steer D, Strawczynski L L, Diffie W, et al. A Secure Audio Teleconference System[C]// CRYPTO'88. 1988: 520-528

[9] Kim Y, Perrig A, Tsudik G. Communication-Efficient group Key Agreement [C]// IFIP SEC. June 2001

[10] Kim Y, Perrig A, Tsudik G. Tree-based group key agreement [J]. ACM Transactions on Information System Security, 2004, 7(1): 60-96

[11] L Li-jun, Manulis M. Tree-based group key agreement framework for mobile Ad-Hoc networks [J]. Future Generation Computer Systems, 2007, 23(16): 787-803

[12] Kim Y, Perrig A, Tsudik G. Simple and fault-tolerant Key Agreement for Dynamic Collaborative Groups [C]// 7th ACM Conference on Computer and Communications Security. 2000: 235-244

[13] Abdullatif S, Melek Ö, Refik M. Local key management in opportunistic networks[J]. International Journal of Communication Networks and Distributed Systems, 2012, 9(1/2): 97-116

[14] Chiou G H, Chen W T. Secure Broadcast using Secure Lock [J]. IEEE Transactions on Software Engineering, 1989, 15(8): 929-934

[15] Kurosawa K. Multi-recipient public-key encryption with shortened ciphetext[C]// Proceedings of 5th International Workshop on Practice and Theory in Public Key Cryptosystem. Paris, France, 2002: 48-63

[16] Liao P, Hui X L, P Qing-qi, et al. A Public Key Encryption Scheme with One-Encryption and Multi-Decryption[J]. Chinese Journal of Computers, 2012, 35(5): 1059-1067

[17] Abdel A K. Cryptanalysis of a Polynomial-based Key Management Scheme for Secure Group Communication[J]. International Journal of Network Security, 2013, 15(1): 68-70

[18] W Qian-hong, Yi M, Willy S, et al. Asymmetric Group Key Agreement[C]// Proceedings of the 28th Annual International Conference on Advances in Cryptology: the Theory and Applications of Cryptographic Techniques (EUROCRYPT'09). 2009: 153-170

[19] Lei Z, W Qian-hong, Bo Q, et al. Asymmetric group key agreement protocol for open networks and its application to broadcast encryption[J]. Computer Networks, 2011, 65(15): 3246-3255

[20] Alkalai L. An overview of flight computer technologies for future NASA space exploration missions[J]. Acta Astronautica, 2003, 52(9-12): 857-867

[21] Cassady R J, Frisbee R H, Gilland J H, et al. Recent advances in nuclear powered electric propulsion for space exploration[J]. Energy Conversion and Management, 2008, 49(3): 412-435

[22] Davarian F, Popken L. Technical advances in deep space communications and tracking[J]. Proceedings of the IEEE, 2007, 95 (11): 2108-2110

[23] 熊永平,孙利民,牛建伟,等. 机会网络[J]. 软件学报,2009,20(1):124-137
Xiong Yong-ping, Sun Li-min, Niu Jian-wei, et al. Opportunistic Networks[J]. Journal of Software, 2009, 20(1): 124-137