软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2022,33(6):2312−2330 [doi: 10.13328/j.cnki.jos.006639]
©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

面向安全关键内存管理系统分层验证方法*

李少峰[1], 乔 磊[2], 杨孟飞[3], 张锦坤[2], 马 智[2], 刘洪标[1]

[1](西安电子科技大学 计算机科学与技术学院, 陕西 西安 710071)

[2](北京控制工程研究所, 北京 100190)

[3](中国空间技术研究院, 北京 100094)

通信作者: 乔磊, E-mail: fly2moon@aliyun.com

摘 要: 安全关键系统的失败会造成很严重的后果, 确保其正确性非常重要. 空间嵌入式操作系统是一个典型的安全关键系统, 在其内存管理的设计上, 必须保障其高效的分配与回收, 同时对系统资源的占用降到最低. 在传统的软件开发过程中, 通常是在整个软件开发结束后再进行集中测试及验证, 这样势必会造成开发进展的不确定性. 因此, 将形式化验证方法和软件工程领域内的“需求-设计-实现”的 3 层开发框架相结合, 通过性质分层传递验证的方法, 保证了各个层次间的一致性. 首先, 从需求层面的需求分析开始, 引入形式化证明的思路, 证明对需求层逻辑的正确性, 从而可以更好地指导程序的设计. 其次, 在设计层面的验证可以极大地减少开发代码的错误率, 证明设计算法和需要实现的函数之间调用逻辑的正确性. 最后, 在实现层, 证明所实现代码与函数设计的一致性, 并且证明代码实现的正确性. 使用交互式定理证明辅助工具 Coq, 以某一国产空间嵌入式操作系统的内存管理模块为例, 证明了其内存管理算法的正确性以及需求、设计、实现的一致性.

关键词: 程序设计; 内存管理; 形式化验证; 精化验证; 嵌入式系统

中图法分类号: TP311

中文引用格式: 李少峰, 乔磊, 杨孟飞, 张锦坤, 马智, 刘洪标. 面向安全关键内存管理系统分层验证方法. 软件学报, 2022, 33(6): 2312–2330. http://www.jos.org.cn/1000-9825/6639.htm

英文引用格式: Li SF, Qiao L, Yang MF, Zhang JK, Ma Z, Liu HB. Verification Method of Hierarchical for Safety-critical Memory Management Systems. Ruan Jian Xue Bao/Journal of Software, 2022, 33(6): 2312–2330 (in Chinese). http://www.jos.org.cn/1000-9825/6639.htm

Verification Method of Hierarchical for Safety-critical Memory Management Systems

LI Shao-Feng[1], QIAO Lei[2], YANG Meng-Fei[3], ZHANG Jin-Kun[2], MA Zhi[2], LIU Hong-Biao[1]

[1](School of Computer Science and Technology, Xidian University, Xi’an 710071, China)

[2](Beijing Institute of Control Engineering, Beijing 100190, China)

[3](China Academy of Space Technology, Beijing 100094, China)

Abstract: The failure of a safety-critical system can cause serious consequences, and it is very important to ensure its correctness. The space embedded operating system is a typical safety-critical system. In the design of its memory management, it must ensure its efficient allocation and deallocation, and the occupancy of system resources is minimizedat the same time. In the traditional software development process, centralized testing and verification are usually carried out after the entire software development is completed, which will inevitably cause uncertaindevelopment. Therefore, this study combines the formal verification method with the three-tier development framework of “demand-design-implementation” in the field of software engineering, and ensures the consistency of each level through the method of layered transfer verification. First, starting from the demand analysis of the demand level, the idea of formal proof is introduced to prove the correctness of the logic of the demand level, which can better guide the design of the program. Second,

* 基金项目: 国家自然科学基金(61632005, 62032004, 61802017) ; 中国科学院软件研究所计算机科学国家重点实验室开放课题基金(SYSKF1804)

收稿时间: 2021-06-06; 修改时间: 2021-09-15, 2021-11-28; 采用时间: 2021-12-28; jos 在线出版时间: 2022-01-28

verification at the design level can greatly reduce the error rate of the development code, and prove the correctness of the call logic between the design algorithm and the function that needs to be implemented. Third, at the code level, it is needed to prove the consistency of the implemented code and the functional design, and prove the correctness of code. Using the interactive theorem proving auxiliary tool Coq, this study takes the memory management module of a domestic space embedded operating system as an example, to prove the correctness of the memory management algorithm and the consistency of demand, design, and code.
Key words: programming; memory management; formal verification; refined verification; embedded system

在实时嵌入式系统中, 必须保障其嵌入式设备在一定时间内完成特定功能. 当外界事件或数据产生时, 必须以足够快的速度进行处理, 调度资源完成实时任务. 其中, 针对内存的高效管理是十分重要的一部分. 动态内存分配(dynamic memory allocators, DMA)算法可以提高应用程序的灵活性, 已经进行了广泛的研究. 但是由于在实时嵌入式系统中, 对于内存分配和释放的时间有严格的限制, 在嵌入式系统中采用 DMA 是一件比较困难的事情, 在没有充分验证和测试的情况下, 很容易在设备运行一段时间后产生难以估量的问题. 在研究航天实时嵌入式系统时, 对于实时性的要求将更加严格. 因此从内存管理的需求分析开始, 就需要保证其正确性, 并证明其满足空间应用的强实时性要求.

DMA 管理有两种通用方法: (1) 显式分配和释放内存, 应用程序在创建时, 调用系统内存管理中的分配函数, 根据其请求的内存大小为其分配合适大小的内存区域供应用程序使用. 应用程序结束运行后, 也需要通知系统, 释放其在使用期间占用的内存. (2) 隐式内存释放(也称为垃圾回收), 应用程序创建时是显式地分配内存区域, 但在应用程序运行结束后, 不进行内存释放, 而是 DMA 在合适的时候检查之前分配的内存, 将不再使用的内存释放掉, 重新分配给新的应用程序.

在强实时嵌入式系统中, 并没有足够的计算资源供内存管理模块来计算内存的使用情况, 所以只考虑显式分配与释放内存的方法. 显式分配和释放的一个简单的实现是基于区域的内存分配与回收, 每个分配的对象都是内存中一段连续的区域, 并且通过销毁区域, 释放其中的所有对象来回收内存.

在强实时嵌入式系统中, 对于 DMA 的选择主要考虑以下几个方面的要求.

- 确定执行时间: 内存分配和释放在最坏情况下的执行时间(WCET)必须是确定的.
- 快速响应时间: 内存的分配和释放必须有一个快速的响应时间.
- 最小化内存池: 减少内存浪费, 在保证响应时间与执行时间最小化的同时最大限度地降低内存的消耗, 内存的碎片化率需要降低.

在影响航天嵌入式软件可信的 10 个核心问题中[1], 内存使用问题、编码问题、各阶段一致性问题这 3 个问题是本文涉及到的内容. 在 SpaceOS[2]的基础上, 本文研究在开发内存管理模块的过程中, 一个可行的经过验证的开发架构, 完成内存管理需求、设计、实现的分层形式化验证, 保证内存管理模块的正确性.

目前关于内存管理算法的验证都是在代码的层次上, 使用不同的验证工具, 采用不同的方式抽象内存, 来验证代码实现的正确性. 也没有一个可行的自动化验证工具被使用, 还是以人工为主, 使用辅助定理证明工具来完成验证, 这会产生大量的验证工作, 验证程序和源程序的代码量会达到十几倍. 因此, 本文提出性质分层传递验证方法, 使得代码层的部分性质在需求层和设计层被验证, 因为需求层和设计层的抽象层次更高, 一些代码层的实现细节被隐藏, 也就简化了验证过程.

需要注意的是, 在性质传递的过程中, 原来代码层的性质可能分解到需求层或者设计层, 所以虽然最后总的验证的性质数量大于单独验证代码层的性质数量, 但每一层验证的性质数量一定是小于单独验证代码层的性质数量. 假设不使用本文的验证方法, 单独验证代码层的性质数量为 Ps. 采用本文提出的方法后, 需求、设计、代码分别验证的数量为 Pr、Pd、Pc, 则 $Pr<Ps \wedge Pd<Ps \wedge Pc<Ps$, $Pr+Pd+Pc>Ps$. 虽然增加了总的验证性质的数量, 但是因为需求层和设计层的抽象层次更高, 验证过程更容易, 最终总的验证代码的行数是减少的. 本文的主要贡献有:

(1) 将形式化验证方法引入到软件工程领域的“需求-设计-实现”3 层开发框架中, 通过性质的逐层传递, 对内存管理模块进行形式化证明, 所有证明工作均在交互式定理证明工具 Coq 中完成.

© 中国科学院软件研究所 http://www.jos.org.cn

(2) 在需求层, 通过逐层精化方法提炼内存管理的性质, 使用有限状态机的方式抽象内存状态, 并在状态机的基础上抽象内存性质形成内存管理程序所需要满足的全局不变式. 然后, 证明这些全局性质的正确性.

(3) 在设计层, 流程图是软件设计的重要输出, 我们对内存管理所涉及的操作进行形式化描述, 以流程图为对象证明我们设计算法的正确性.

(4) 在实现层, 结合设计层函数操作的形式化描述, 以实现的函数代码为基础, 基于霍尔逻辑的方式证明代码实现的正确性.

本文在开发及验证嵌入式系统中的内存管理模块时, 其整体框架如图 1 所示.

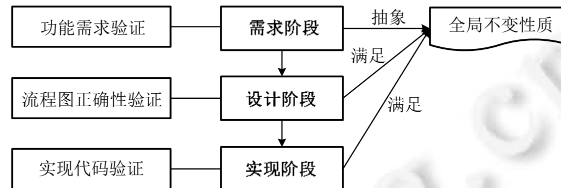

图 1 分层验证方法框架

1 相关工作

形式化建模和验证在计算机领域的很多方面都得到了应用. 法国计算机科学家 Leroy 领导的 CompCert 项目[3]验证了一个 C 编译器, 使用辅助定理证明器 Coq 证明了 CompCert 编译器在整个编译过程中保持了代码的语义. 德国教育研究部资助的 VeriSoft 项目[4]对整个计算机系统自底向上进行形式化验证. 将整个操作系统分为 5 层, 每一层都使用定理证明工具 Isabelle/HOL 经过了人工验证. Shao 等人[5,6]将自动化定理证明的优点与基于 VeriML/OCAP 的认证编程框架模块化表达相结合, 对 CertiKOS 的内核安全性进行了研究.

苏黎世联邦理工学院的 Abril[7]声称使用形式化方法可以产生“无错误的系统”, 虽然其在文章中对这个声明做出了很多的限制, 但是也让我们了解到应用形式化方法开发正确系统的可行性. 英国约克大学的 Woodcock[8]讨论了一些已经成功应用形式化方法的工业应用, 例如列车控制系统、刷卡系统等, 表明形式化方法在诸多领域都具有了广泛的应用, 使用形式化方法交付的软件产生的错误更少. 但是其中大多数的项目都是系统状态确定的静态验证, 在状态复杂的场景下, 对于形式化的使用还很少.

近年来, 形式化验证在工业领域的应用越来越广. 无论是轻量级的形式化与主流方法的结合, 还是重量级的形式化方法在工业级软件上的应用, 都取得了较大的进步和成功[9]. Bolton 等人[10]将形式化验证应用在人机交互的软件中, 以证明潜在的意外错误行为是否会导致系统故障. Trindade 等人[11]首次应用模型检查来验证独立太阳能光伏系统的设计, 从而在购买设备和部署之前, 确保预期行为的安全. 王尚等人[12]面向嵌入式控制软件的需求, 将形式化方法与量纲分析相结合, 并以轨道交通列车控制软件为例来验证其需求.

操作系统内核中的内存管理模块是一个逻辑复杂的模块, 目前并没有一个统一的验证方式, 各个不同的项目都是根据不同操作系统的特点做出验证. seL4[13]采用了边设计边验证的方式, 利用函数式程序语言 Haskell 来开发内核原型, 并在 Isabelle/HOL 定理证明器中得到可执行规范, 最后再将 Haskell 原型手动转换为高性能的 C 语言代码. 针对内存管理部分的验证, 其将内存分配的权利部分授权给应用程序, 降低了内核验证的复杂性, 而在应用层对内存的验证依赖于内核验证的属性. Yu 等人曾对一个 C 标准库中的 malloc/free 简化算法进行了严格的形式化验证[14]. 引入了一种低级语言 CAP, 用于构建认证程序并提供认证用于动态内存分配的库.

内存管理模块是操作系统的基础, 且是一个运行状态复杂的模块. 提供最佳且形式化证明正确的 DMA 是一项具有挑战性的任务. 首先, 在管理内存区域提供低开销和满足内存请求的高速度的 DMA, 还没有最佳的通用解决方案. 针对不同的操作系统, 内存管理模块的设计应考虑其具体用途, 并采用多种技术组合以满足该应用场景. 这导致要采用多种多样的 DMA 算法进行实现并分别证明各个算法是正确的. 其次, 这些实

© 中国科学院软件研究所 http://www.jos.org.cn

现通常将低级代码(例如, 指针算术、位字段)与有效率的高级别数据结构(例如, 具有双向链表的哈希表)结合在一起, 用于证明这种优化实现正确性的形式方法已经被几个项目证明了[15,16]. 但是, 在这些项目中使用的技术并不能很好地扩展以验证不同 DMA 的实现正确性.

对于实时系统的内存管理模块的一个有趣特性是其动态内存分配/重新分配机制的复杂度是 O(1)(没有循环), 正确地构建这种系统的内存管理模块是一个严峻的挑战. 同时, DMA 的现有实现采用了大量策略和技术, 这些技术即使在单独进行形式规范时也非常复杂, 更不用说在组合使用下的验证. Fang[17]通过提供形式化模型来解决这个问题, 使用建模框架 Event-B 在定理证明平台 Rodin 上证明了模型的完善性和精化关系. Su[18]详细说明了一些算法细节及相关的复杂数据结构, 遵循 Event-B 的精化原则, 从一些初始需求中逐步构造经过验证的可执行代码. Qiao[19]基于工具 Event-B 形式化证明了航天器的内存管理系统, 但是没有很好地解决需求到代码的一致性问题, 也没有涉及到内存管理中复杂的数据结构. Li 等人[20]使用有限状态机来抽象系统内存状态, 提出内存管理模块所需要满足的性质, 并使用归纳演绎的推理方法完成了验证.

TLSF (two level segregated fit)[21,22]动态内存算法内存分配和释放时间的复杂度为常数, 具有内存自动合并、灵活性强、内存碎片少等特点, 在很多系统中得到了应用. 本文在国内自主研发并经过空间飞行的空间飞行器嵌入式实时操作系统 SpaceOS 的基础上, 将形式化验证方法与软件工程领域的“需求-设计-实现”3 层开发框架相结合, 通过性质分层传递的验证方法, 研究其内存管理算法的正确性.

2 需求层验证

本文研究的内存管理模块是在安全攸关的嵌入式系统上使用的, 因此内存管理模块需要有严格的限制. 因为嵌入式系统的内存资源有限, 必须高效地组织内存的分配与回收, 需要一个低碎片率的内存算法; 同时, 为了准确评估系统任务的执行时间, 并保证任务在其截止期前执行完成, 需要内存管理算法对内存的分配与回收是确定执行时间的. TLSF 算法正好符合我们的应用要求, 因此我们首先对该算法的需求层进行验证.

2.1 需求层逐层精化

针对空间嵌入式系统的内存管理模块, 如图 2 所示, 我们通过 5 个层次来逐层分析与精化其需求, 并在每一层提出其所对应的功能需求.

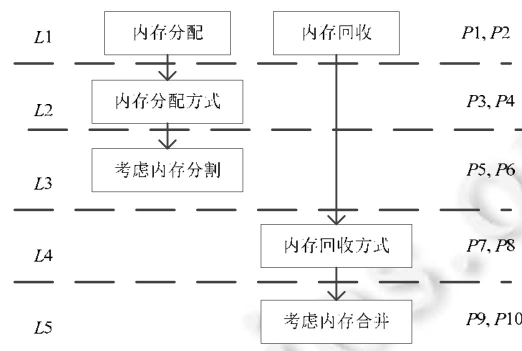

图 2 内存管理需求分层精化

*L*1: 动态内存管理最基本的需求是内存的分配与回收. 其应满足性质 1 与性质 2.

性质 1. 如果内存区中存在大小为 *N* 个字节的连续地址空间, 对任意一个申请内存大小小于等于 *N* 个字节的分配请求都是可被满足的.

性质 2. 任意已被分配的内存空间都是可被释放的.

*L*2: 考虑内存分配的方式. 对于一整块内存如何将其分配给应用程序使用. 应该满足性质 3、性质 4.

性质 3. 状态为占用的内存块不能被再分配.

性质 4. 任意两块毗邻的内存块不能重叠.

*L*3: 在进行内存分配时, 还需要考虑内存块的分割, 系统中剩余的是一个比较大的内存块, 而应用程序需要的内存比较小时需要对系统中的内存块进行分割, 使其满足应用程序的需求, 且不会浪费系统的内存空间.

性质 5. 分配给应用使用的内存块与应用要求的内存大小的差值不会超过某固定值.

性质 6. 内存最小空闲块的大小大于等于某固定值.

*L*4: 考虑内存释放的方式, 释放应用程序不在使用的内存块, 让内存管理模块可以重新分配这部分内存, 增加内存的利用率.

性质 7. 任意已被释放的内存空间都是可被分配的.

性质 8. 状态为空闲的内存块不能再被释放.

*L*5: 在内存释放时, 如果有两个相邻的空闲内存块, 则需要对内存块进行合并, 降低内存的碎片, 使得内存块可以被应用于更大的应用程序需求.

性质 9. 两块空闲块之间至少有 1 块非空闲块.

性质 10. 没有相邻的空闲块.

2.2 需求层性质规范

前一小节通过对内存管理模块的分层精化, 归纳出了 10 条全局性质, 本小节对这 10 条全局性质进行形式化规范. 如图 3 所示, 其中, b 和 c 指的是内存块, s 指的是应用程序的内存需求, $size(b)$指的是该内存块的大小, $alloc(s)$指的是为应用程序分配满足需求 s 的内存块, $free(b)$指的是释放内存块 b, $right(b)$指的是内存块 b 的下一个内存块, $left(b)$指的是内存块 b 的上一个内存块, $sta(b)$指的是内存块的状态(占用或空闲), M 指的是系统中最小内存块的值.

$$
\begin{aligned}
&p1: && \forall s\cdot\exists b\cdot size(b)>s \Rightarrow alloc(s)\neq b\\
&p2: && \forall b\cdot sta(b)=1\Rightarrow free(b)=\text{true}\\
&p3: && \forall b,s\cdot sta(b)=1\Rightarrow alloc(s)\neq b\\
&p4: && \forall b\cdot[b,b+size(b))\cap[right(b),right(b)+size(right(b)))=\varnothing\\
&p5: && \forall s,b\cdot\exists M\cdot b=alloc(s)\Rightarrow s\leqslant right(b)-b<s+M\\
&p6: && \forall b\cdot\exists M\cdot size(b)\geqslant M\\
&p7: && \forall b\cdot\exists s\cdot sta(b)=0\Rightarrow alloc(s)=b\\
&p8: && \forall b\cdot sta(b)=0\Rightarrow free(b)=\text{false}\\
&p9: && \forall b,c\cdot sta(b)=sta(c)=0\wedge b<c\Rightarrow right(b)<c\wedge sta(right(b))=1\\
&p10: && \forall b\cdot sta(b)=0\Rightarrow sta(right(b))=1\wedge sta(left(b))=1
\end{aligned}
$$

图 3 内存管理需求的全局性质规范

2.3 需求层建模

基于需求层的逐层精化, 内存管理模块对外提供的函数见表 1 中的 *init*、*alloc*、*free*. 函数 *init* 初始化内存, 初始化内存块的索引结构. 函数 $alloc(n)$在内存块索引结构中搜索块主体大小至少为 n 的空闲块, 并返回给应用程序. 函数 $free(p)$释放内存块 p, 将其内存块状态标记为空闲, 放入内存块的索引结构中. 另外, 为了降低内存的碎片率, 还需要两个内部函数 *split* 和 *merge*. 函数 *split* 发生在 $alloc(n)$后, 如果分配的内存块远大于应用程序的要求时, 则对内存块进行分割. 函数 *merge* 发生在 $free(p)$后, 如果内存块 p 有相邻的空闲内存块, 则对其进行合并操作.

表 1 DMA 函数

void init()
void* alloc(size_tsz)
bool free(void* p)
void split(size_tsz,void* p)
bool merge(void* p)

采用有限状态机模型对内存管理的需求层进行建模, 见表 2, 定义了一些基本的变量与常量, 通过它们来

© 中国科学院软件研究所 http://www.jos.org.cn

规范表示系统的状态. 使用 $s\triangleq\langle H,clt,msize\rangle$表示系统的内存状态, *clt* 表示内存块列表的尾地址. *H* 表示内存块列表, *msize* 表示内存的大小. *H* 是由内存块组成的内存块列表, 每一个内存块使用 $mc\triangleq\langle bst,bad,bsi\rangle$表示系统分配给程序使用的内存区域, 其中, *bst* 代表着每一个内存块的状态, 使用或空闲, *bad* 代表着内存块的初始地址, *bsi* 代表着内存块的大小.

表 2　内存状态的抽象

符号	描述
$clh, clt\in\mathbb{N}$	内存块列表的初始和尾地址
msize	整个系统内存的大小
bst: {0,1}	内存块状态, 1-使用, 0-空闲
$bad, bsi\in\mathbb{N}$	内存块的首地址与内存块大小
$mb\triangleq\langle bst,bad,bsi\rangle$	内存块
H: *list mb*	内存块列表
$s\triangleq\langle H,clt,msize\rangle$	系统的内存状态

在内存的初始状态, 内存块列表为空, $clt=0$, $msize=M$, 内存状态是已知的, 系统结束后, 释放所有占用的内存块后, 内存块列表重新变为空, $clt=0$, $msize=M$, 内存状态已知并和初始状态一样. 因此, $s_0=s_f=id$, 其中, s_0是初始状态, s_f 是最终状态, 初始状态和终止状态相等, 记为 *id*. 随着系统任务不断的创建、执行、终止, 系统内存的状态会不断发生变化, 任意时刻的内存状态都是随机的, 如果证明每一个内存状态都满足第 2.1 节中提出的 10 条性质, 将具有十分巨大的工作量. 因此, 我们首先对这 10 条性质作进一步精化, 并结合模型的抽象方式, 提出了 6 条不变式, 见表 3. 表中有一些涉及到内存块的操作, *bst* 代表内存块的状态, *bad* 代表内存块的首地址, *bsi* 代表内存块的大小, *bnc* 代表下一个内存块, *bpr* 代表上一个内存块. 内存状态在内存函数的变换下永远满足这 6 条性质, 从而证明我们需求层的规范是正确的.

表 3　内存状态的不变式

规范表达	描述
I_1: $clh=0$	内存块列表的起始位置为 0
I_2: $clt<mz$	尾地址的值小于内存大小 *mz*
I_3: $\forall b\in H\wedge bst(b)free\Rightarrow bst(bnx(b))=busy$	当前内存块状态为空闲, 则下一块状态为占用
I_4: $\forall b\in H\wedge bst(b)free\Rightarrow bst(bpr(b))=busy$	当前内存块状态为空闲, 则上一块状态为占用
I_5: $\forall b\in H\Rightarrow bsi(b)>0$	块的大小永远大于 0
I_6: $\forall b\in H\Rightarrow bad(b)+bsi(b)=bad(bnx(b))$	内存的每个元素都属于一个块

2.4 需求层证明

我们在 Coq 中采用形式化方法描述了这些函数, 并采用归纳法的形式演绎证明出内存状态在内存管理的所有环节都满足这些全局性质.

I_1: 在内存状态变换函数中, 并没有任何函数会改变 *clh* 的值, 其值只会在系统开始的阶段, 赋值为 0. 因为它的值并不会改变, 所以在抽象系统状态时, 也并没有用其来描述内存. 所以在系统的任何时刻, *clh* 值恒为 0. I_1 得证. □

I_2: 内存状态使用 $s\triangleq\langle H,clt,msize\rangle$来描述, 在内存状态变换函数中, 只有函数 *alloc* 会改变内存状态的 *clt* 的值. 函数 *alloc* 改变 *clt* 值的条件是 $clt+N<mz$, 所以函数 *alloc* 改变 *clt* 值, 内存状态改变为 $H\cup c\rightarrow H$; $clt+N\rightarrow clt$, 即 *clt* 变为 $clt+N$ 的条件是 $clt+N<mz$, 所以 $clt<mz$ 是恒成立的. I_2 得证. □

I_3 和 I_4 是相对应的, 在证明时可以合并为一个性质来进行. $I_{3,4}$: $\forall b\in H\wedge bst(b)free\Rightarrow bst(bnx(b))=busy\wedge bst(bpr(b))=busy$. 函数 *free* 后的函数 *merge* 会涉及到不变式 $I_{3,4}$. 证明过程在 Coq 中如下给出:

```
Theorem release_free_surround_busy: forall (l: blocklist)(b: Block),
   free_surround_busy l=true->free_surround_busy B (release b l)=true.
Proof.
   intros.unfold free_surround_busy; simpl.Induction l.
   - unfold allocate; unfold b; simpl; try (intros; reflexivity).
   - assert (H1: free_surround_busy (release b(b1:: l))=free_surround_busy (release bl)).
      {apply(lng_relate_allocate l b b1).assumption.}
```

© 中国科学院软件研究所 http://www.jos.org.cn

```
        rewrite H1.case(block_is_end b l)); simpl.
    - fold release; simpl; reflexivity.
    - fold free_surround_busy; fold release; assumption.
Qed.
```

在证明过程中引入了一些引理，不是这里介绍的重点，就不再详细加以说明了．至此完成了对内存管理阶段内存状态满足不变式 I_3 和 I_4 的证明．□

I_5：在内存管理程序中，涉及到内存块大小改变的函数有 *merge*、*alloc*、*split*．函数 *merge* 改变内存块大小一定是增加的．函数 *alloc* 则是根据 *sz* 的大小，生成内存块，但内存块的大小一定是大于 *sz* 的．*Split* 是对内存块进行分割，分割后产生的内存块也一定大于系统的最小内存块．所以 I_5 是永远被满足的．□

I_6：在内存管理函数中涉及到内存块的大小或首地址变化的规范函数有 *alloc*．I_6 在 Coq 中定义为 *list_not_gap*，类似于性质 $I_{3,4}$ 的证明．在 Coq 中证明性质 *alloc_list_not_gap* 如下所示：

```
Theorem alloc_list_not_gap: forall (l: blocklist)(r: nat),
    list_not_gap l=true->list_not_gap (alloc r l)=true.
Proof.
    intros.induction l.
    - unfold alloc; simpl.case r; simpl; try (intros; reflexivity).
    - assert (H1: list_not_gap (alloc r(b:: l))=list_not_gap (alloc r l)).
    {apply(lng_relate_alloc l r b).assumption.}
    rewrite H1.apply lng_equal_minus in H.apply IHl in H.assumption.
Qed.
```

通过对定理的证明，完成了对不变式 I_6 的证明．□

3 设计层验证

在空间嵌入式操作系统中，为了保证任务的实时性要求，需要保证内存分配与回收的时间是确定的，为 O(1)时间级．我们以 TLSF 算法为基础，设计空间实时嵌入式系统内存管理模块．

3.1 TLSF算法模型抽象

TLSF 算法使用分组空闲链表(segregated list)的形式组织系统内的所有空闲块，并且通过位图匹配(bitmaps fit)的机制使得在分配内存块时可以快速定位到相应的空闲块链表，从而实现 O(1)时间级的分配内存块．所以，在设计层基于需求层对内存块和 TLSF 的索引结构进行抽象．

3.1.1 内存块的抽象

内存块结构：内存块的块头充分包含这个内存块的信息，综合考虑内存分配的相关操作，内存块的块头需要包含以下信息：内存块的状态、内存块物理上相邻的块、在空闲块链表上的相邻块、内存块的大小．如图 4 所示，是内存块的基本结构．

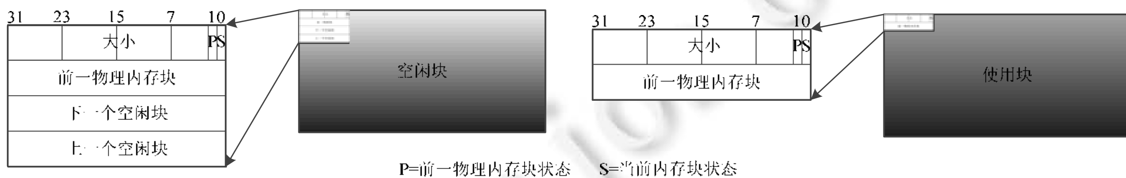

图4 空闲块和使用块的块头

在需求层对内存块抽象的基础上，进一步描述 TLSF 相关的内存块属性，见表 4．每一个内存块使用一个五元组进行表示，$mb \triangleq \langle bst, bad, bsi, preB, FBAdd \rangle$ 表示系统分配给程序使用的内存区域，其中，*bst* 代表着每一个内存块的状态(使用或者空闲)，*bad* 代表着内存块的开始地址，*bsi* 代表着内存块的大小，*preB* 代表着内存块在物理上前一个内存块及其状态，*FBAdd* 指的是内存块空闲时，其在空闲内存块链表上的位置．*preB* 由 *preBA* 和 *preBS* 构成，分别指的是内存块在物理内存上的前一个内存块地址及其状态．*FBAdd* 由 *preFB* 和 *nextFB* 构成，分别指的是其在空闲内存块链表上的前一个空闲内存块和后一个空闲内存块．

© 中国科学院软件研究所 http://www.jos.org.cn

表 4 内存块的抽象

符号	描述
bst, *preBS*: {0,1}	内存块状态, 1-使用, 0-空闲
bad, *bsi*, *preBA*∈ℕ	内存块的首地址、大小、前一个内存块
preFB, *nextFB*∈ℕ	空闲块链表上的前一个和后一个块
FBAdd≙⟨*preFB*,*nextFB*⟩	空闲块位置
preB≙⟨*preBA*,*preBS*⟩	前一个物理内存的位置和状态
mb≙⟨*bst*,*bad*,*bsi*,*preB*,*FBAdd*⟩	内存块

3.1.2 TLSF 索引结构的抽象

TLSF 算法的索引结构采用二级位图的方式快速查找合适的内存块. 首先, 第 1 级索引将内存块按照 2 的 *n* 次幂大小进行划分, 例如 32、64、128 等. 第 2 级索引再将第 1 级索引的内存块大小区间进行线性划分, 划分的个数是 2^{SLI}. 一般 *SLI* 为 4, 但不能超过 5, 从而可以使用 32 位的位图确定某一范围内是否有空闲内存块. 如图 5 所示, 是 TLSF 算法索引结构的一个示例, 其中, *SLI*=2.

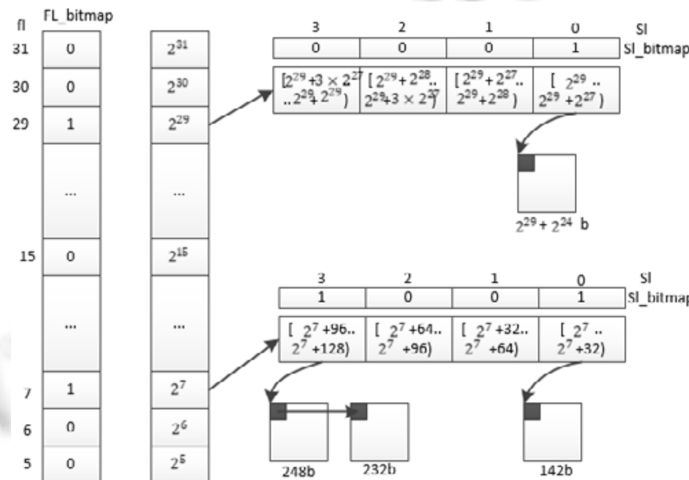

图5 TLSF算法索引结构

首先在一级位图 FL_bitmap 每一位上的数字表示某个范围内是否有空闲块, 1 表示有, 0 表示没有. 如图中一级索引 7 和 29 对应的位是 1, 则意味着在$[2^7,2^8]$和$[2^{29},2^{30}]$这个区间内有空闲内存块. 其次, 二级位图对这个范围再进行线性划分, 如图中范围$[2^7,2^8]$下为 1 的位表明[128,160]和[224,256]这个区间内有空闲内存块. 最后, TLSF 索引结构通过一个二维数组, 其两个维度的值为 *fl* 和 *sl*, 可以定位到一个空闲块的链表.

在进行内存块查找时, 根据程序内存需求计算出其对应的 *fl* 和 *sl*, 然后通过二级位图判断在该内存大小范围下有没有空闲内存块. 没有的话, 则根据二级位图找到内存大小更大范围的 *fl* 和 *sl*, 最后返回 *fl* 和 *sl* 定位的空闲块链表的首个内存块. 在空闲内存块查找过程只有位图的判断操作, 其时间复杂度是 O(1)时间级的.

TLSF 算法的索引结构主要有 3 个参数, 一级索引、二级索引、空闲块链表. 对其抽象见表 5.

表 5 TLSF 索引结构的抽象

符号	描述
FL, *SL*∈*BitU*32	TLSF 索引的一级和二级索引
MSL∈*list SL*	二级索引列表
BL∈*list mb*	空闲内存块列表
MBL∈*list BL*	多级空闲块列表
TS≙⟨*FL*,*MSL*,*MBL*⟩	TLSF 的索引结构

其中, *BitU*32 是在 Coq 中定义的一个 32 位的整数, 一级索引 *FL* 和二级索引 *SL* 都是这个数据类型的. 在 Coq 中使用列表来模拟链表的结构, *MSL* 是多个二级索引组成的二级索引列表. *BL* 是空闲内存块的列表, *MBL*

多个空闲内存块列表组成了多级空闲块列表. 最后对 TLSF 索引结构的抽象就变为了使用一个三元组 $TS \triangleq \langle FL, MSL, MBL \rangle$. 索引结构中空闲块列表 *MBL* 的个数 $Lnum = REAL_FLI * MAX_SLI$. *REAL_FLI* 是一级索引实际上将内存分开的区域数, MAX_*SLI* 是二级索引将一级索引下继续细分的区域数.

3.1.3 内存状态的抽象

在需求层, 系统的内存状态被抽象为 $s \triangleq \langle H, clt, msize \rangle$, 其中, *clt* 在 TLSF 索引结构抽象中可以得出, *msize* 是一个常量, 表示系统内存总的大小. *H* 表示系统中所有内存块的列表, 包含空闲内存块和使用内存块. 而在 TLSF 索引结构抽象中, 有系统中空闲内存块的抽象, 所以为了表示系统的内存状态, 这里还需要表示系统中被使用的内存块列表 *UL*, 因此将设计层的系统抽象为 $s^D \triangleq \langle TS, UL, msize \rangle$.

3.2 设计层与需求层的一致性

在需求层完成了对内存管理模块功能需求的证明, 证明了我们对空间嵌入式系统中内存管理模块所提出的功能需求是正确的. 而在设计层, 我们针对需求层的功能进行相应的设计, 所以需要证明设计层和需求层是一致的. 在我们的方法中, 这个一致性指的是需求层的性质在设计层仍然是被满足的.

对内存管理模块建模, 最主要的两个方面是内存块的抽象和系统内存状态的抽象. 对于内存块, 需求层和设计层的抽象分别为 $mb \triangleq \langle bst, bad, bsi \rangle$和 $mb^D \triangleq \langle bst, bad, bsi, preB, FBAdd \rangle$, 设计层的抽象包含了需求层的抽象, 是对需求层抽象的拓展. 需求层和设计层的内存块是一一对应的关系, 满足如下二元关系.

$$\forall b \in mb, \exists b' \in mb^D. bst(b') = bst(b) \wedge bad(b') = bad(b) \wedge bsi(b') = bsi(b).$$

所以, 需求层抽象的内存块所满足的性质, 也一定会被设计层抽象的内存块满足. 使用 *P* 表示内存块在需求层所满足的性质的集合, P''表示内存块在设计层所满足的性质的集合, 那么 $P \subset P''$.

对于系统内存状态, 需求层和设计层的抽象分别为 $s^R \triangleq \langle H, clt, msize \rangle$和 $s^D \triangleq \langle TS, UL, msize \rangle$. *clt* 包含在设计层抽象的 *TS* 中, 设计层的 *TS* 和 *UL* 一起可以转换为需求层的 *H*, 也就是说, 对于任意一个设计层的内存状态, 可以通过函数 *SDetoRe* 转换为需求层的抽象状态. 即$\forall s \in s^D$, $\exists s' \in s^R. s' = SDetoRe(s)$. 所以需求层已经证明过的内存状态满足的 6 个不变式对于设计层也是满足的. 因此, 我们将需求层模型满足的不变式传递到了设计层, 从而实现了需求到设计的一致性证明.

定义 1. 需求层到设计层的性质传递.

对于状态 $s \in s^R$, 令 $I(s)$表示 6 个不变式对于状态 *s* 成立, 所以,

$$\forall s \in s^D, I(SDetoRe(s)).$$

3.3 设计层函数

如表 6 所示, TLSF 算法基本函数接口, 需要在需求层函数的基础上, 添加函数 *remove*(*void* p*), 从对应的空闲块链表中移除内存块; 函数 *insert*(*void* p*), 将内存块插入到对应的空闲块链表中. 接下来我们对每一个函数进行详细的设计与建模. 设计指的是流程图, 建模指的是对函数进行形式化语义描述.

表 6 TLSF 基本函数接口

void init()
void* alloc(size_tsz)
void* search(size_tsz)
void remove(void* p)
void* split(size_tsz,void* p)
void insert(void* p)
bool free(void* p)
bool merge(void* p)

3.3.1 初始化 Init

在系统的一开始, 就需要先执行 *Init*()函数, 完成对内存管理模块中快速搜索表、动态内存区基址、内存池的管理块初始化等步骤. 其中内存池的管理块就是我们在第 3.1.2 节中分析的 TLSF 索引结构的抽象. 初始化后, 整个内存池被认为是一个大的内存块, 将其插入到内存的索引结构 *TS* 中, 从而使得在索引结构中查找到对应的内存块. 如图 6 所示.

© 中国科学院软件研究所 http://www.jos.org.cn

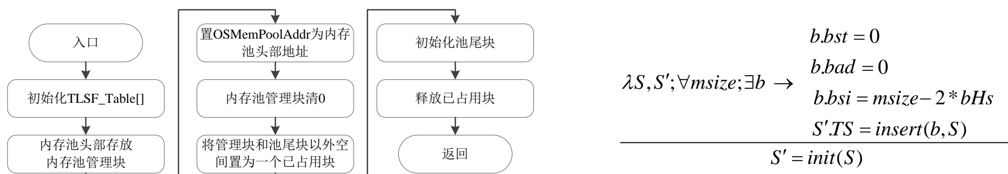

图 6 初始化流程图和形式化语义

从初始化函数中提炼的性质如下. P1: 初始化后内存中存在一个极大的空闲块; P2: 内存的总大小为内存池管理块、被管理的空闲块、池尾块的大小之和.

3.3.2 内存块查找 search

内存管理系统根据任务所需的内存大小更新后的值 *size* 和 TLSF 的索引结构进行内存块的查找, 根据 *size* 计算出对应的 *fl* 和 *sl*, 找到其中对应的内存块链表. 此时, 被查找到的内存块链表中每一个内存块的大小都大于任务需要的内存大小 *size*. 在实时系统中, 需要确定的执行时间, 因此在 TLSF 算法中, 将该内存块链表中的首个内存块返回给任务, 在该内存块链表中移除该内存块. 如果没有找到, 则需要更新 *fl*、*sl* 的值, 在内存大小更大范围的内存块链表中查找, 如果还是没有找到, 则返回空. 如图 7 所示.

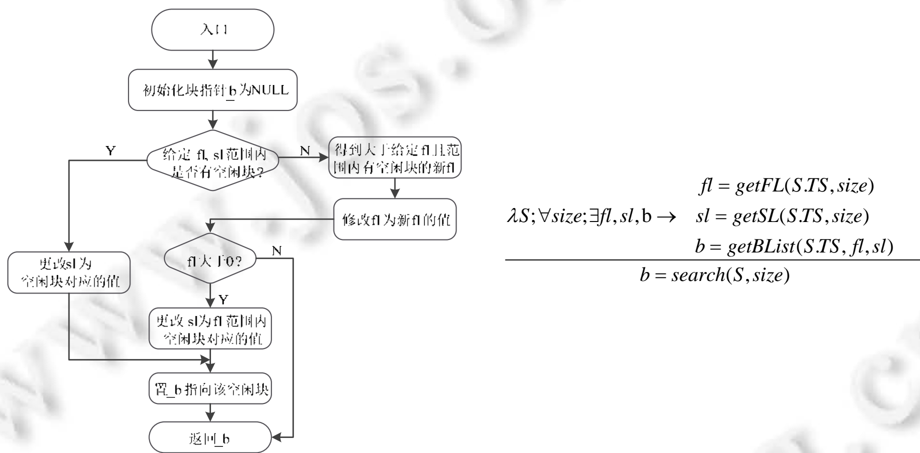

图 7 内存块查找流程图和形式化语义

从内存块查找函数中提炼的性质如下. P3: 合适内存块范围 *fl* 不变时, *sl* 是增加的; P4: 合适内存块范围 *fl* 改变时, *fl* 是增加的; P5: 找到的内存块的状态是空闲的; P6: 找到的空闲块的大小大于任务的内存需求.

3.3.3 内存块分配 *alloc*

内存管理系统根据任务所需的内存大小进行内存分配. 首先, 根据所需内存的大小 *size* 和系统的最小内存块大小 *MBS* 确定实际应该分配的内存块, *size*=(*size*<*MBS*) ? *MBS*: *size*, 同时还需要对内存进行双字对齐, 就是操作语义中 *update*(*size*)中的执行过程. 其次, 根据此时内存块的索引结构 *TS* 和更新后的内存需求 *size′* 查找系统中合适的内存块, 就是操作语义中的 *search*(*S*,*size′*)的执行过程. 最后, 对找到的内存块进行空闲链表的移除操作, 并将该内存块的下一个内存块的 *FBAdd* 进行更新, 亦即其 *preFB* 和 *nextFB* 这两个属性. 分配内存时对找到的内存块可能需要执行拆分的操作, 将在后续小节进行分析. 以上过程的操作语义如下. 其中, *b′* 是内存块 *b* 所分割出的内存块, 而 *b″* 是内存块 *b* 在实际物理设备上的下一个内存块, 需要对相关属性进行更新. 如图 8 所示.

从内存块查找函数中提炼的性质如下. P7: 内存请求更新后的内存请求大小大于原始的内存请求; P8: 内存请求更新后的内存请求大小不小于最小内存块大小; P9: 根据内存请求计算出的所有 *fl* 和 *sl* 都在其合法范围内.

© 中国科学院软件研究所 http://www.jos.org.cn

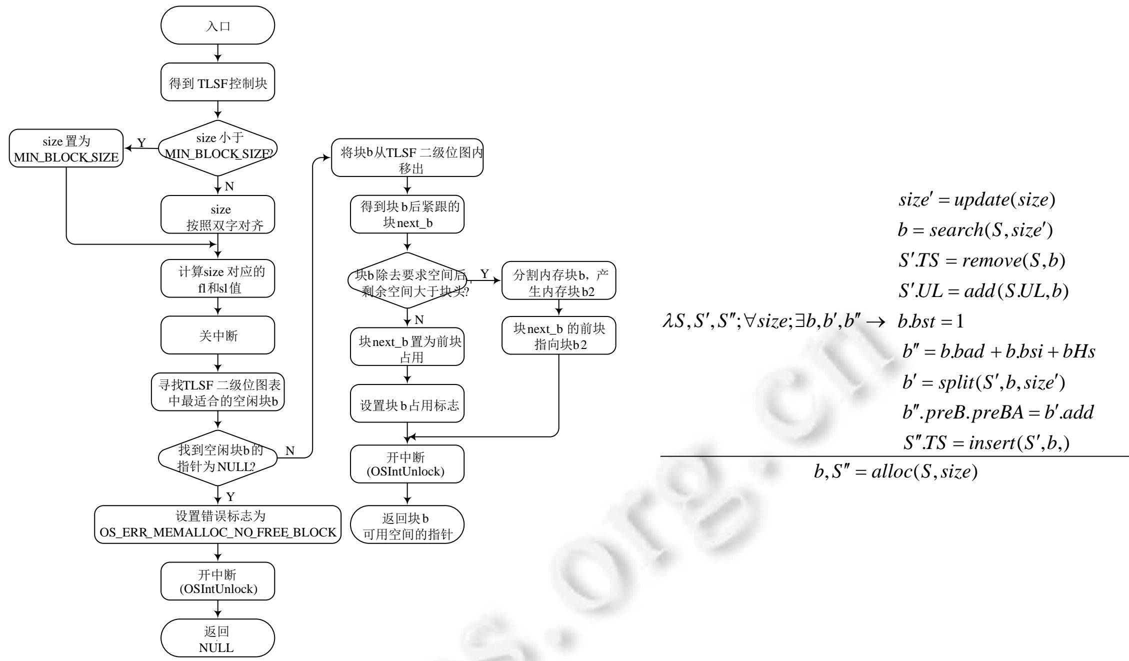

图 8 内存块分配流程图和形式化语义

3.3.4 内存块移除 remove

在内存管理系统找到应用所需要的内存块 *b* 后要将其从系统的内存索引结构 *TS* 中移除，完成内存索引结构的更新. 首先，根据该内存块属性 *FBAdd* 中的 *preFB* 和 *nextFB*，通过更改这两个数值将一个内存块从索引结构中移除，并将这个内存块的状态更改为使用. 然后，根据判断该内存块后面是否还有内存块进行下一步操作. 如果没有，则更新 *TS* 中的 *fl* 和 *sl*，否则，更改其 *FBAdd* 中的 *preFB* 和 *nextFB* 属性，分别称为 $remove^L$ 和 $remove^B$. $remove^L$ 指的是内存块链表中还存在内存块，只需要对该内存块更新即可，$remove^B$ 指的是内存块链表中没有内存块了，则需要更新内存索引中的二级索引. 如图 9 所示.

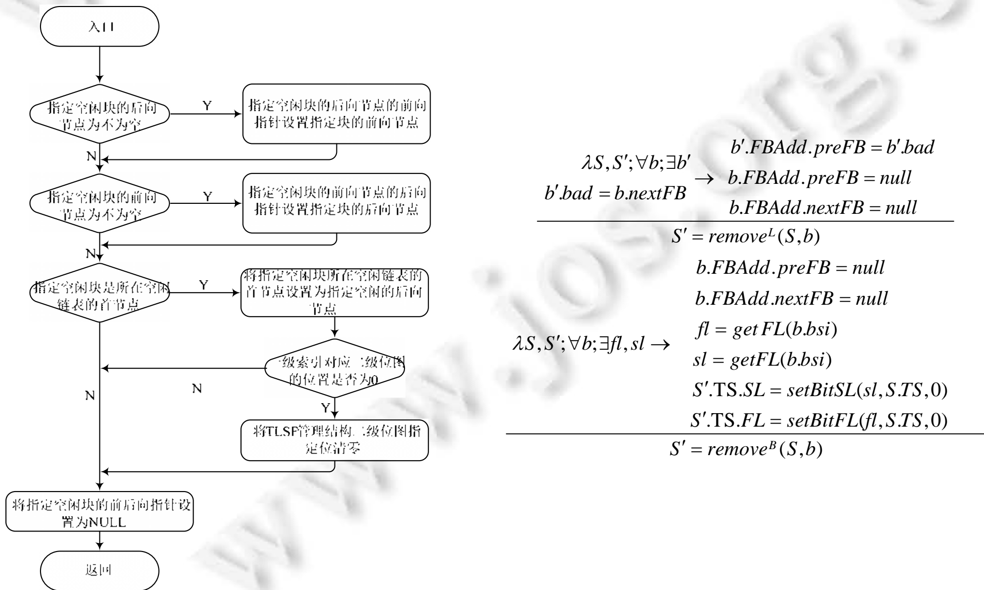

图 9 内存块移除流程图和形式化语义

从内存块移除函数中提炼的性质如下. P10: 该空闲块链表的节点数小于原链表的节点数; P11: 如果开始空闲块链表的节点数为 1, 则经过移除函数后节点数为 0.

3.3.5 内存块分割 *split*

在内存管理系统为应用找到合适的内存后, 如果找到的内存块 b 的大小大于应用所要求的内存大小 $size$, 且剩余部分大于系统要求最小内存块, 则执行内存块的分割操作. 其将内存块 b 的大小设为 $size$, 将剩余的内存组成一个新的内存块 b'并插入到内存的索引结构中. 需要注意的是, 在考虑分割内存块时, 应该考虑到一个内存块头的大小 bHs, 计算的时候减去块头. 如图 10 所示.

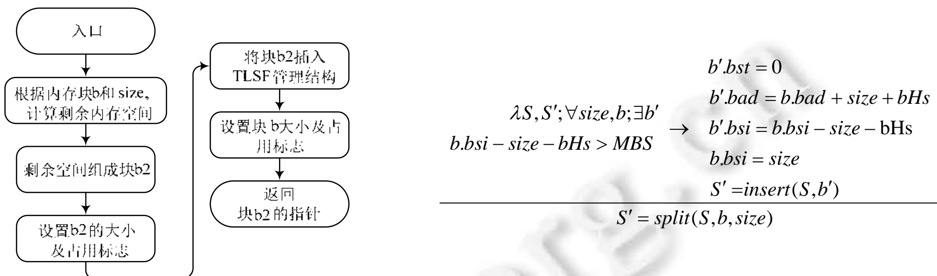

图 10 内存块分割流程图和形式化语义

从内存块分割函数中提炼的性质如下. P12: 分割出的内存块的内存范围在原先的内存块 b 内; P13: 分割出的内存块的状态为空闲.

3.3.6 内存块插入 insert

当内存管理系统对块进行分割并释放一个使用块后, 都需要将这个块插入到内存的索引结构中, 完成内存索引结构的更新. 首先, 根据该内存块的大小计算相应的 fl 和 sl, 对相应的位索引进行更新. 然后, 将所插入内存块的 *FBAdd* 属性中的 *preFB* 和 *nextFB* 进行更新, 将其连接到对应的空闲内存块链表上. 如图 11 所示.

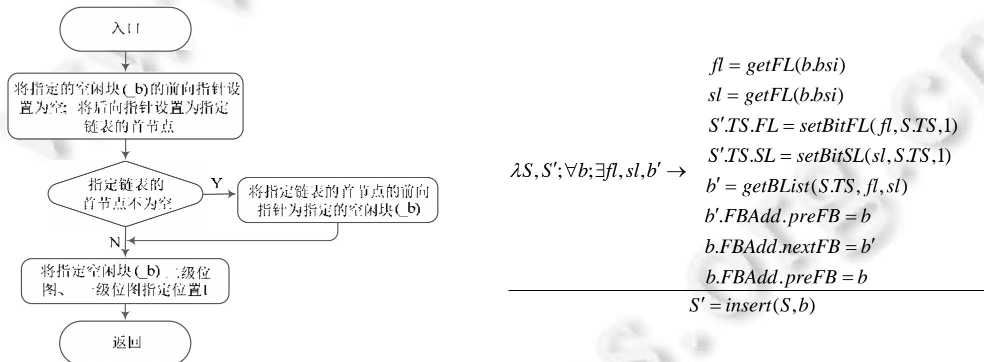

图 11 内存块插入流程图和形式化语义

从内存块分割函数中提炼的性质如下. P14: 该内存块是插入空闲链表的首节点; P15: 插入空闲块后空闲链表的节点数加 1.

3.3.7 内存块释放 *free*

系统中应用运行结束后, 内存管理系统释放该应用使用的内存块, 根据其内存块大小, 将其插入到内存管理结构中合适的空闲列表上, 并将其内存块的状态更改为空闲, 使得系统下一次分配可以使用该内存块. 如图 12 所示.

从内存块释放函数中提炼的性质如下. P16: 被释放的内存块的状态从占用变为空闲.

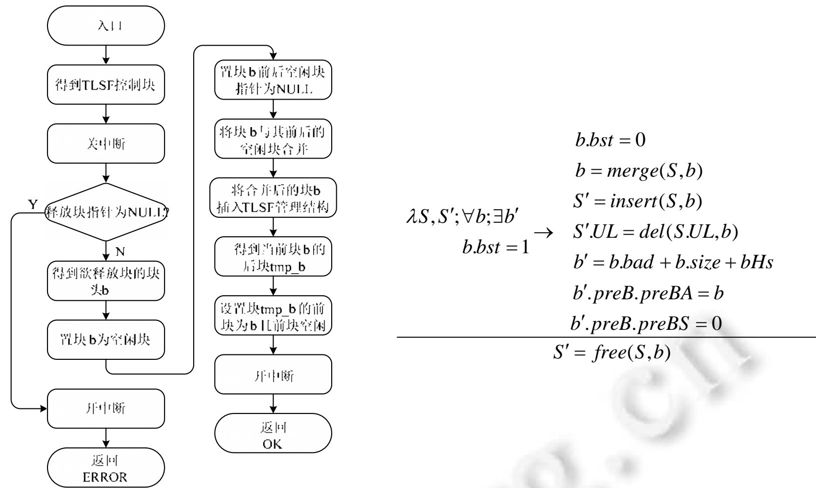

图 12　内存块释放流程图和形式化语义

3.3.8　内存块合并 *merge*

在对应用的内存块释放后，需要找到该内存块物理内存上的前后内存块，判断其状态，如果是占用的，无需执行任务操作，如果是空闲的，则需要进行合并操作. 因为其前后内存块状态的不同，这里需要分为左合并、右合并、全和并，其操作语义表示为 $merge^L$、$merge^R$、$merge^A$. 如图 13 所示.

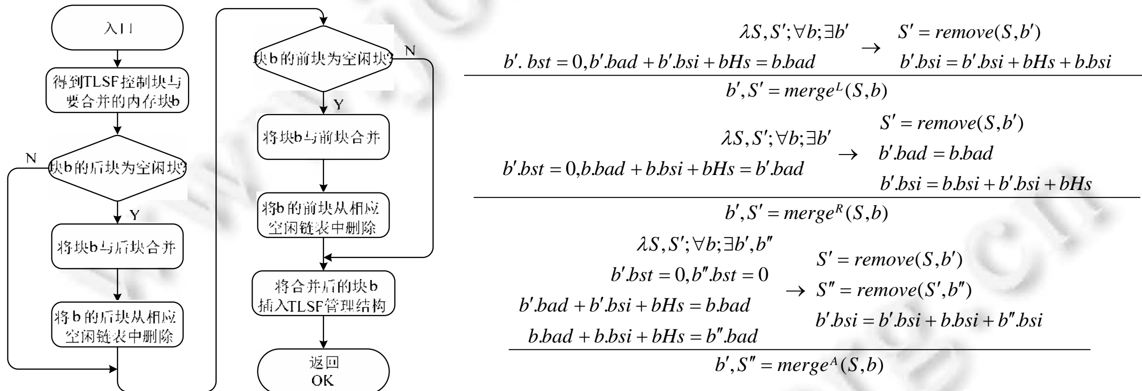

图 13　内存块合并流程图和形式化语义

从内存块合并函数中提炼的性质如下. P17: 合并后的内存块大小大于原来的内存块; P18: 被释放的内存块合并之后，其相邻块的状态一定为占用.

3.3.9　其他函数

在对内存管理函数进行形式化规范时，也会使用一些辅助函数，比如 *setBitFL*、*setBitSL*、*getFL*、*getSL*、*add*、*del*. 其中，*setBitFL* 是更新内存块索引中的一级索引 *FL*, *setBitSL* 是更新内存块索引中的二级索引 *SL*. *getFL* 和 *getSL* 则相反，是根据一个 *size* 获得其 *fl* 和 *sl* 值. *add* 和 *del* 是对占用块列表中块的添加和删除.

3.4　设计层证明

上一节对要实现的 TLSF 算法的主要函数进行了形式化的规范，并且通过流程图提炼出设计层所应该满足的 18 条性质. 结合设计层的抽象，在 Coq 中证明这些性质的正确性. 这 18 个性质在 Coq 中描述见表 7.

© 中国科学院软件研究所 http://www.jos.org.cn

表 7　内存管理模块设计层性质描述

P1	Theorem extMBlock: forall (m: nat), getBst(firstB(getBL(getMap(init m))))=free.
P2	Theorem extMSize: forall (m: nat), getBsi(firstB(getBL(getMap(init m))))=m-tlsfS-BlockInfoS.
P3	Theorem slGrow: forall (fl sl: nat), updatefl fl sl=fl->updatesl fl sl>=sl.
P4	Theorem flGrow: forall (fl sl: nat), updatefl fl sl<>fl->updatefl fl sl>fl.
P5	Theorem searchBFr: forall (size: nat), exists (tlsf: TLSF), getBst(searchB size tlsf)=free.
P6	Theorem searchBsiG: forall (size: nat), exists tlsf: TLSF, getBsi(searchB size tlsf)>size.
P7	Theorem updateGe: forall (size: nat), update size>size.
P8	Theorem updateGeMS: forall (size: nat), update size>=MIN_BLOCK_SIZE.
P9	Theorem checkFL: forall (size: nat), size>MBS∧size<mSize-> inFLRange(getFL size)=true∧inSLRange(getSL size)=true.
P10	Theorem rmblsize: forall (bl: BL), (blsize(rm bl))<(blsize bl).
P11	Theorem rmbl2nil: forall bl: BL, blsize bl=1->rm(bl)=nil.
P12	Theorem splitBin: forall (b b1: mb)(size: nat), size<(getBsi b)->b1=(split b size)-> (getBad b1)>(getBad b)∧getBsi b1+getBad b1<(getBad b)+(getBad b).
P13	Theorem splitBiF: forall (b: mb)(size: N), size<(getBsi b)-> b1=(split b size)->(getBst b1)=free.
P14	Theorem insertBFi: forall (b: mb)(bl: BL), firstB(insert b bl)=b.
P15	Theorem insertBadd: forall (b: mb)(bl: BL), blsize(insert b bl)=(blsize bl)+1.
P16	Theorem BusyB2F: forall (b: mb), getBst b=busy->getBst(free b)=free.
P17	Theorem mergeSizeGe: forall (b: mb), exists (b': mb), b'=merge b->(getBsi b')>=(getBsi b).
P18	Theorem mergelrBlockBusy: forall (b: mb), exists (b': mb), b'=merge b-> getBst(getNextB b')=busy∧getBst(getPreB b')=busy

4　实现层验证

在设计层, 我们对内存管理函数的每一个函数进行设计与建模, 并基于流程图证明了各个函数设计逻辑的正确性, 本节证明内存管理模块实现层代码的正确性.

4.1　实现层模型

在内存管理模块的代码实现中, 数据的传递通过指针的方式直接传递给相关的函数, 会在形式化验证过程中造成函数间逻辑过于复杂, 验证十分困难. 但是, 因为我们采用了“需求-设计-实现”的证明方法, 对于函数间的逻辑关系在设计层已经完成了证明, 在实现层只需要证明所实现的代码和设计层的设计是一致的即可. 相比于设计层的抽象, 代码层需要将内存管理阶段所涉及到的所有变量进行描述, 对每一行代码执行涉及的变量变化都需要清晰的描述.

在代码实现中, 最主要的结构是空闲内存块的索引结构. 因为分配函数会根据空闲内存块的索引结构和应用程序的内存需求查找到合适的内存块, 并将这个内存块返回给应用程序. 而释放函数会将一个使用的内存块状态更改为空闲, 并重新插入到空闲块索引结构中. 所以, 在代码层的模型中, 重点关注空闲内存块的索引结构. 根据图 4 内存块的结构, 我们描述内存块的结构体如下.

```
struct free_block{                          /*  空闲块指针结构  */
        struct BlockHead *prev;             /*  指向前一个空闲块的块头  */
        struct BlockHead *next;             /*  指向后一个空闲块的块头  */
};
struct BlockHead {                          /*  块头结构  */
        int size;                           /*  块大小, 单位字节, 4 对齐*/
        struct BlockHead *phys_prev;        /*  指向前一块的块头  */
        union{                              /*  前后空闲块指针, 仅空闲块有意义  */
                FREE_PTR_T free_ptr;
                U8 buffer[1];
        } ptr;
};
```

所以, 空闲内存块在 Coq 中的定义如下.

```
Inductive FlagB: Type: =|busy| free.
Inductive Block: Type: =|block (flag: FlagB)(first size preBApreBSpreFBnextFB: nat).
```

© 中国科学院软件研究所 http://www.jos.org.cn

TLSF算法在代码层的索引结构如下所示. 其中, *tlsf_signature*是一个标志, 在TLSF的函数运行期间是不会更改的. *fl_bitmap* 是索引结构的一级索引, *sl_bitmap*[*REAL_FLI*]是二级索引组成的数组, **matrix* [*REAL_FLI*][MAX_*SLI*]是一个内存块的链表数组, 数组的每一个元素都是前面介绍的内存块的指针或者是空指针.

```
struct TLSF_struct{
    U32 tlsf_signature;                    /* 内存池初始化标志 */
    U32 fl_bitmap;                         /* 一级索引 fl */
    U32 sl_bitmap[REAL_FLI];               /* 二级索引 sl */
    BHDR *matrix[REAL_FLI][MAX_SLI];       /* 索引表指针 */
};
```

在Coq中对该空闲块索引结构建模时不需要关注 *tlsf_signature* 标志, 而是重点关注其他元素的建模. 在设计层, 我们将TLSF算法的索引结构抽象为 $TS\triangleq\langle FL,MSL,MBL\rangle$. 在代码层的建模阶段, 需要在其基础上进行展开, *matrix*[*REAL_FLI*][MAX_*SLI*]的数组长度十分大, 接近 2^{10}, 所以我们定义的TLSF实际上是根据 *fl* 和 *sl* 已经查找到的隔离列表. 在Coq中的定义如下.

```
Inductive blocklist: Set: =|block_nil|block_cons(b: Block)(l: blocklist).
Inductive Map: Type: =|map(fl: nat)(sl: nat)(bl: blocklist).
Inductive TLSF: Type: =|tlsf(fl: U32)(sl: U32)(dmap: Map).
```

其中, U32是对一个32位的二进制数的定义, 为了直观地反映位的操作, 在Coq使用了针对每一个位的列表进行了定义, 对于32位数的定义如下.

```
Inductive bitC: Set: =|xl|xo.
Inductive bitl: Set: =|l|o.
Inductive U32: Type: =|u32(b1: bitl)(b2 b3 b4 b5 b6 b7 b8 b9 b10 b11 b12 b13 b14 b15
b16 b17 b18 b19 b20 b21 b22 b23 b24 b25 b26 b27 b28 b29 b30 b31 b32: bitC).
```

对于32位二进制数的使用, 在Coq中也同样定义了很多操作, 比如相加、递增、与、或、和、左移、右移等操作. 接下来我们通过代码层的模型, 来验证代码实现的正确性. 分配函数和释放函数是内存管理模块最重要的两个函数, 这里便以这两个函数为例.

实现层的验证对象应该是具体实现的代码, 需要验证代码中每一条语句对数据的改变是否符合设计层的规范, 所以从伪代码中构建代码所需要满足的性质, 并在Coq中完成性质的验证. 伪代码的建立依据的是设计层的流程图和函数形式化规范, 其一致性是由代码的实现者确认的, 我们不进行其一致性验证, 仅针对代码层的性质进行验证.

4.2 分配函数

根据分配函数的设计流程图, 对分配函数进行代码实现, 其伪代码如下.

算法. void *MemAlloc(U32 size).
输入: 应用程序需求内存大小size;
输出: 分配给应用程序的内存块b.

```
if size<MIN_BLOCK_SIZE
1.   then size=MIN_BLOCK_SIZE
2.   else size=ROUNDUP_SIZE(size)
3. MappingSearch(&size, &fl, &sl)   //①
4. Block b=FindSuitableBlock(tlsf, &fl, &sl)   //②
 if (b==(BHDR *)NULL)
       then return (void *)NULL   //③
5. OSMemExtractBlockHDR(b, tlsf, fl, sl)
6. next_b=GET_NEXT_BLOCK(b->ptr.buffer, b->size)
```

```
 if (b->size-size>MIN_BLOCK_SIZE)
7.   then splitBlock(b, size, next_b)
8.   else setsta(next_b)
 return (void *) b->ptr.buffer   //④
```

其中, ①是根据 *size* 的大小来匹配 *fl* 和 *sl*; ②是索引结构中查找合适的内存块; ③是没有找到, 返回空块; ④是返回分配的内存块.

在伪代码的描述中, 为了简单, 省略了一些变量的定义. *tlsf* 是空闲内存块索引结构的定义. *fl* 和 *sl* 分别指的是某一大小的内存块在 *tlsf* 索引结构中的位置, *fl* 是一级链表, *sl* 是二级链表. 在内存管理模块的代码实现中, 通过指针传递的方式调用其他的函数, 我们看到, 有函数 *MappingSearch*、*FindSuitableBlock*、*GET_NEXT_BLOCK*、*OSMemExtractBlockHDR*、*splitBlock*、*setsta*. 所以在抽象内存状态时就需要考虑到所有被传递指针值的变化程度, 从而完成函数代码实现的证明. 分配函数的程序中, 涉及到的变量有 *size*、*fl*、*sl*、*tlsf*、*b*、*next_b*, 然后通过这些变量的变化建立程序逻辑描述并最终证明. 除了返回语句和条件语句外, 将 8 条执行语句定义为 $S_A^1, S_A^2, ..., S_A^8$, 建立这 8 条程序的逻辑如下.

(A1) $\{size>0 \wedge size<\mathrm{MIN}B_SIZE\}S_A^1\{size=\mathrm{MIN}B_SIZE\}$

(A2) $\{size>\mathrm{MIN}B_SIZE\}S_A^2\{size'>size\}$

(A3) $\{size\geqslant \mathrm{MIN}B_SIZE \wedge fl=0 \wedge sl=0\}S_A^3\{(fl=0\wedge sl>0)\vee(fl>0\wedge sl=0)\vee(fl>0\wedge sl>0)\}$

(A4) $\{b=null\wedge Q_3\}S_A^4\{b=null\vee(b\neq null\wedge((fl'=fl\wedge sl'>sl')\vee fl'>fl))\}$

(A5) $\{tlsf\wedge Q_4\}S_A^5\{tlsf'\}$

(A6) $\{next_b=null\}S_A^6\{next_b\neq bull\}$

(A7) $\{bsi(b)-size>\mathrm{MIN}B_SIZE\wedge tlsf\}S_A^7\{bsi(b)=size\wedge tlsf'\}$

(A8) $\{bsi(b)-size\leqslant \mathrm{MIN}B_SIZE\wedge next_b\}S_A^8\{next_b'\}$

逻辑语句中的 Q_3、Q_4 是指语句 3 和语句 4 的后置状态, 直接使用其作为语句(4)、语句(5)的前置状态. *Size*′、*fl*′、*sl*′、*tlsf*′、*next_b*′都指这些变量发生了改变. 其具体改变的情况将根据相关的 *S* 进行操作. 在实际证明过程中, 在 Coq 中进行了实际的描述, 证明了这些逻辑的正确性. $S_A^1, S_A^2, ..., S_A^8$, 实际上就是一些内存管理的函数, 或者一些程序块, 这些都在 Coq 中进行了定义. 这 8 条程序逻辑的定义如下.

```
Theorem A1: forall (size: nat), size>0∧size<MINB_SIZE->update(size)=MINB_SIZE.
Theorem A2: forall (size: nat), size>MINB_SIZE->update(size)>size.
Theorem A3: forall (size: nat), size>MINB_SIZE->
   (MappingSearchfl(size)=0∧MappingSearchsl(size)>0)∨(MappingSearchfl(size)>0∧
   MappingSearchsl(size)=0)∨(MappingSearchfl(size)>0∧MappingSearchsl(size)>0).
Theorem A4: forall (b: Blockt), exists (fl sl: nat), b=null∧(fl=0∧sl>0)∨(fl>0∧sl=0)∨(fl>0∧sl>0)–>
   b=null∨(b<>null∧((FindSuitableBlockfl(fl,sl)=fl∧(FindSuitableBlocksl(fl,sl)>sl)∨(FindSuitableBlockfl(fl,sl)>fl).
Theorem A5: forall (tlsf1: TLSF), exists (tlsf2: TLSF), tlsf2=MemExtractBlock(tlsf1).
Theorem A6: forall (b: Block), getNextBlock(b)<>null.
Theorem A7: forall (b: Block), forall (size: nat), bsi(b)-szie>MINB_SIZE->bsi(split(b,size))=size.
Theorem A8: forall (b next_b: Block), forall (size: nat), exists (nb: Block),
   bsi(b)-szie<=MINB_SIZE∧next_b=getNextBlock(b)->nb=setsta(next_b).
```

4.3 释放函数

根据释放函数的设计流程图, 对释放函数进行代码实现,其伪代码如下.

算法. STATUS MemFree(void *ptr).

输入: 需要释放的内存块;

输出: 释放是否成功的状态.

```
if ptr==NULL
  then return OSERROR
Block b=(BHDR *)(ptr)   //①
1.  merge(b)
```

© 中国科学院软件研究所 http://www.jos.org.cn

2. MappingInsert(b, &fl, &sl)
3. OSMemInsertBlock(b, tlsf, fl, sl)
4. next_b=GET_NEXT_BLOCK(b->ptr.buffer, b->size)
5. setsta(next_b)

return OSOK

其中, ①是将指针 *ptr* 转化为指向内存块的指针.

类似于分配函数, 通过变量的变化建立程序逻辑描述并最终证明. 除了返回语句和条件语句外, 将5条执行语句定义为 S_F^1, S_F^2, ..., S_F^5, 建立这 5 条程序的逻辑如下.

$(F1)$ $\{b\neq null\}S_F^1\{b'\}$

$(F2)$ $\{b\neq null\wedge fl=0\wedge sl=0\}S_F^2\{(fl=0\wedge sl>0)\vee(fl>0\wedge sl=0)\vee(fl>0\wedge sl>0)\}$

$(F3)$ $\{tlsf\wedge Q_F^2\}S_F^3\{tlsf'\}$

$(F4)$ $\{next_b=null\}S_F^4\{next_b\neq null\}$

$(F5)$ $\{next_b\neq null\}S_F^5\{next_b'\}$

逻辑语句中的 Q_F^2 是指语句 $F3$ 的后置状态, 直接使用其作为语句 $F3$ 的前置状态. *next_b′*和 *tlsf′*都指的是这些变量发生了改变. 其具体改变的情况将根据相关的 S 进行操作. 在实际证明过程中, 在 Coq 中进行了实际的描述, 证明了这些逻辑的正确性. 这些逻辑在 Coq 中的定义如下.

```
Theorem F1: forall (b: Block), exists (b′: Block), b<>null->b′=merge(block).
Theorem F2: forall (b: Block), b<>null->(MappingInsertfl(b)=0∧MappingInsertsl(b)>0)
   ∨(MappingInsertfl(b)>0∧MappingInsertsl(b)=0)∨(MappingInsertfl(b)>0∧MappingInsertsl(b)>0).
Theorem F3: forall (b: Block), exists (tlsf: TLSF), tlsf=MemInsertBlock(b).
Theorem F4: forall (b: Block), exists (next_b: Block), next_b=next_b=getNextBlock(b).
Theorem F5: forall (b: Block), exists (b′: Block), b′=setsta(b).
```

5 结论和未来的工作

本文针对安全关键的嵌入式操作系统, 从空间领域的实际需求出发, 将形式化方法与软件工程领域内的需求、设计、实现这 3 层开发验证相结合. 以一个国产空间嵌入式操作系统的内存管理模块的开发过程为例, 在需求层对内存管理模块的功能需求进行验证, 在设计层针对函数流程图的设计进行正确性验证, 在实现层对代码实现符合需求的规范和设计进行验证, 从系统实现的一开始就保证该系统模块的正确性. 在验证过程中都采用交互式定理证明辅助工具 Coq 完成形式化验证工作.

本文提出的验证开发方法具有通用性, 对于一个安全关键的系统来说, 必须严格保证系统的正确性, 采用本方法对系统进行开发, 可以从源头减少系统错误的发生, 保障系统的可靠性.

未来, 考虑将这个验证开发方法进行更广泛的推广, 应用到其他安全关键领域内系统的开发, 并将相关验证策略进行抽象, 提取出验证的共性部分封装为验证框架, 方便验证其他领域软件时使用.

References:

[1] Yang MF, Gu B, Guo X Y, *et al*. Aerospace embedded software dependability guarantee technology and application. SCIENTIA SINICA Technologica, 2015, 45(2): 198–203 (in Chinese with English abstract). [doi: 10.1360/N092014-00485]

[2] Qiao L, Yang MF, Gu B, Yang H, Liu B. An embedded operating system design for the lunar exploration rover. In: Proc. of the 5th Int'l Conf. on Secure Software Integration and Reliability Improvement—Companion. Jeju: IEEE, 2011. 160–165. [doi: 10.1109/SSIRI-C.2011.39]

[3] Krebbers R, Leroy X, Wiedijk F. Formal C semantics: CompCert and the C standard. In: Proc. of the Int'l Conf. on Interactive Theorem Proving. Cham: Springer, 2014. 543–548. [doi: 10.1007/978-3-319-08970-6_36]

[4] Baumann C, Bormer T. Verifying the PikeOS microkernel: First results in the verisoft XT avionics project. In: Proc. of the Doctoral Symp. on Systems Software Verification (DS SSV 2009) Real Software, Real Problems, Real Solutions. RWTH Aachen University, 2009. 20–22.

© 中国科学院软件研究所 http://www.jos.org.cn

[5] Vaynberg A, Shao Z. Compositional verication of a baby virtual memory manager. In: Proc. of the 2nd Int'l Conf. on Certied Programs and Proofs (CPP). Berlin, Heidelberg: Springer-Verlag, 2012. 143−159. [doi: 10.1007/978-3-642-35308-6_13]

[6] Gu RH, Shao Z, Chen H, Wu XN, Kim J, Sjioberg V, Costanzo D. CertiKOS: An extensible architecture for building certied concurrent OS kernels. In: Proc. of the 12th USENIX Conf. on Operating Systems Design and Implementation (OSDI). Savannah: USENIX Association, 2016. 653−669.

[7] Abrial J. Faultless systems: Yes we can! Computer, 2008, 42(9): 30−36. [doi: 10.1109/MC.2009.283]

[8] Woodcock J, Larsen PG, Bicarregui J, Fitzgerald J. Formal methods: Practice and experience. ACM Computing Surveys, 2009, 41(4): 1−40. [doi:10.1145/1592434.1592436]

[9] Klein G, Andronick J, Fernandez M, Kuz I, Murray T, Heiser G. Fromally verified software in the real world. Communications ofthe ACM, 2018, 61(10): 68−77. [doi: 10.1145/3230627]

[10] Bolton ML, Molinaro KA, Houser AM. A formal method for assessing the impact of task-based erroneous human behavior on system safety. Reliability Engineering and System Safety, 2019, 188(AUG): 168−180. [doi: 10.1016/j.ress.2019.03.010]

[11] Trindade AB, Cordeiro L. Automated formal verification of stand-alone solar photovoltaic systems. Solar Energy, 2019, 193: 684-691. [doi: 10.1016/j.solener.2019.09.093]

[12] Wang S, Feng JC, Zhu JY, Huang GH, Zheng HY, Xu XR, Miu WK, Zhang K, Pu GG. A dimensional analysis method for the requirements model of railway control software. Chinese Journal of Computers, 2020, 43(11): 136−149 (in Chinese with English abstract). [doi:10.11897//SP.J.1016.2020.02152]

[13] Klein G, Elphinstone K, Heiser G, Andronick J, Cock D, Derrin P, Elkaduwe D, Engelhardt K, Kolanski R, Norrish M, Sewell T, Tuch H, Winwood S. seL4: Formal verication f an OS kernel. In: Proc. of the 22nd ACM SIGOPS Symp. Operating Systems Principles (SOSP). NY: Association for Computing Machinery, 2009. 207−220. [doi:10.1145/1629575.1629596]

[14] Yu DC, Hamid NA, Shao Z. Building Certified Libraries for PCC: Dynamic Storage Allocation. Berlin, Heidelberg: Springer-Verlag, 2003. 363−379. [doi: 10.5555/1765712.1765739]

[15] Adam C. Mostly-automated verification of low-level programs in computational separation logic. In: Proc. of the 32nd ACM SIGPLAN Conf. on Programming Language Design and Implementation. NY: Association for Computing Machinery, 2011. 234−245. [doi: 10.1145/1993498.1993526]

[16] Nicolas M, Reynald A, Akinori Y. Formal verification of the heap manager of an operating system using separation logic. In: Proc. of the 8th Int' l Conf. on Formal Methods and Software Engineering. Berlin, Heidelberg: Springer-Verlag, 2006. 400−419. [doi: 10.1007/11901433_22]

[17] Fang B, Sighireanu M, Geguang PU, Su W, Abrial JR, Yang MF, Qiao L. Formal modelling of list based dynamic memory allocators. Science China (Information Sciences), 2018, 61(12): 81−96. [doi: 10.1007/s11432-017-9280-9]

[18] Su W, Abrial JR, Pu GG, Fang B. Formal development of a real-time operating system memory manager. In: Proc. of the 20th Int'l Conf. on Engineering of Complex Computer Systems (ICECCS). QLD: IEEE, 2015. 130−139. [doi: 10.1109/ICECCS.2015.2]

[19] Qiao L, Yang MF, Tan YL, Pu GG, Yang H. Formal verification of memory management system in spacecraftusing Event-B. Ruan Jian Xue Bao/Journal of Software, 2017, 28(5): 1204−1220 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/ 5218.htm [doi: 10.13328/j.cnki.jos.005218]

[20] Li SF, Qiao L, Yang MF. Memory state verification based on inductiveand deductive reasoning. IEEE Trans. on Reliability, 2021, 70(3): 1026−1039. [doi: 10.1109/TR.2021.3074709]

[21] Masmano M, Ripoll I, Balbastre P, Crespo A. A constant-time dynamic storage allocator for real-time systems. Real Time Systems, 2008, 40(2): 149−179. [doi: 10.1007/s11241-008-9052-7]

[22] MasmanoM, Ripoll I, Crespo A, RealJ. TLSF: A new dynamic memory allocator for real-time systems. In: Proc. of the Euromicro Conf. on Real-time Systems. Catania: IEEE, 2004. 79−88. [doi: 10.1109/EMRTS.2004.1311009]

附中文参考文献:

[1] 杨孟飞, 顾斌, 郭向英, 等. 航天嵌入式软件可信性保障技术及应用研究. 中国科学: 技术科学, 2015, 45(2): 198−203.

[12] 王尚, 冯劲草, 诸嘉逸, 黄怿豪, 郑寒月, 徐想容, 缪炜恺, 张翔, 蒲戈光. 面向轨交控制软件需求模型的量纲分析方法. 计算机学报, 2020, 43(11): 136−149.

[19] 乔磊, 杨孟飞, 谭彦亮, 蒲戈光, 杨桦. 基于 Event-B 的航天器内存管理系统形式化验证. 软件学报, 2017, 28(5): 1204−1220. http://www.jos.org.cn/1000-9825/5218.htm [doi: 10.13328/j.cnki.jos.005218]

李少峰(1992－), 男, 博士生, 主要研究领域为嵌入式操作系统 ,内存管理, 文件系统, 形式化验证.

张锦坤(1993－), 男, 硕士, CCF 专业会员, 主要研究领域为嵌入式操作系统, 任务管理, 形式化验证.

乔磊(1982－), 男, 博士, 研究员, CCF 高级会员, 主要研究操作系统模型设计, 任务调度策略, 存储管理.

马智(1994－), 男, 博士生, 主要研究领域为嵌入式操作系统, 中断管理, 形式化验证.

杨孟飞(1962－), 男, 博士, 研究员, CCF 高级会员, 主要研究领域为空间飞行器嵌入式系统, 控制系统, 总体技术.

刘洪标(1995－), 男, 博士生, 主要研究领域为嵌入式操作系统, 任务调度, 形式化验证.

www.jos.org.cn